Auto-Tuning and Performance Portability on Heterogeneous Hardware

Von der Fakultät 5 Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
David Pfander
aus Herrenberg

Hauptberichter:	Prof. Dr. rer. nat. Dirk Pflüger
Mitberichter:	Prof. Dr. rer. nat. Hans-Joachim Bungartz
Tag der mündlichen Prüfung:	28.11.2019

Institut für Parallele und Verteilte Systeme der Universität Stuttgart

2019

Abstract

In high-performance computing, excellent node-level performance is required for the efficient use of supercomputers. However, manual optimization is a tedious process that commonly needs to be repeated for every hardware platform targeted. Auto-tuning has been developed as an approach to partially automate the optimization process, generally by tuning parameterized compute kernels. Through auto-tuning, both high performance on a single hardware platform and performance portability can be achieved.

In this work, we present the auto-tuning framework AutoTuneTMP that leverages two features: just-in-time (JIT) compilation and C++ template metaprogramming. JIT compilation enables a tight integration of auto-tuning into an application and reduces the time required for auto-tuning. Template metaprogramming enables a design focused on ease-of-integration, maintainability and extensibility. It further forms the basis for optimization templates that support the development of auto-tunable compute kernels. Additionally, our framework provides search strategies for performing parameter tuning.

To demonstrate the applicability and usefulness of our framework, we use AutoTuneTMP to auto-tune three algorithms that require excellent performance. The first algorithm is matrix multiplication. Whereas many implementations are manually optimized or even use assembly, we rely on template metaprogramming for a high-level approach. Across four hardware platforms, we achieved up to 91% of the peak performance and are, therefore, competitive with vendor libraries such as Intel's MKL.

Sparse grid regression is well-suited for machine learning in moderate-dimensional big data scenarios. However, large datasets necessitate high-performance algorithms. We introduce two auto-tuned high-performance algorithms for this application: the unified streaming algorithm and the subspace algorithm. These algorithms serve as the second and third example for auto-tuning with AutoTuneTMP. The unified streaming algorithm is written in OpenCL and targets a wide range of architectures including GPUs. The subspace algorithm was developed for processor platforms only, but has a lower time complexity. Due to these auto-tuned algorithms and a new approach for the spatial adaptivity of sparse grids, speedups of up to 13x were measured compared to the state-of-the-art surplus-refined masked streaming approach. Furthermore, we demonstrate that both new algorithms are performance-portable.

Apart from auto-tuning, we investigate the performance portability of a new distributed variant of the sparse grid clustering method. As a density-based clustering method, it relies on a sparse grid density estimation to compute density functions for

large datasets of moderate dimensionality in linear complexity in the size of the dataset. The algorithm consists of four major components: two density compute kernels and two compute kernels for creating and pruning a k-nearest-neighbor graph. All components were written using OpenCL and MPI. On the node-level, we reached on average 79% of the achievable peak performance of one processor and four GPUs. In distributed experiments on two supercomputers, Hazel Hen and Piz Daint, we measured up to 352 TFLOPS using 128 nodes. As a similar fraction of peak performance was achieved on both supercomputers and because of the high node-level efficiency, we can demonstrate the performance portability of the algorithm.

Our work shows that performance portability is a realistic goal for scientific applications on modern hardware. By using JIT compilation and template metaprogramming, the tightly-integrated auto-tuning approach presented reduces the effort required for optimization without compromising on performance.

Acknowledgments

This work would not have been possible without many helpful suggestions, technical discussions and friendly conversations. Firstly, I want to thank my supervisor Dirk Pflüger for his valuable input on all aspects of the research project that became this thesis. Furthermore, at the Simulation of Large Systems chair and, later on, the Simulation Software Engineering chair, I was fortunate to have many great colleagues that pointed me to fruitful new technical approaches, often over a cup of coffee (Gregor, Steffen, Fabian, Florian, Michael, Caro, Theresa, Miriam and others). Additionally, I was given the opportunity to spend three enlightening months at Louisiana State University with Hartmut Kaiser and his amazing team (Adrian, Zack and many more). There, Hartmut taught me proper C++. Finally, my parents and, of course, Anna supported me in facing all the challenges that awaited me. They helped me to stay balanced and motivated—and enabled me to focus during the months of writing.

Zusammenfassung (German Abstract)

Im wissenschaftlichen Hochleistungsrechnen ist eine exzellente Nutzung der Rechenleistung einzelner Rechenknoten notwendig, um Supercomputer effizient nutzen zu können. Allerdings ist die manuelle Leistungsoptimierung ein mühsamer Prozess, der gewöhnlich für jede unterstützte Hardwareplattform wiederholt werden muss. Auto-Tuning wurde als ein Ansatz für die teilweise Automatisierung des Optimierungsprozesses entwickelt. Verbreitete Ansätze nutzen eine automatisierte Bestimmung der Leistungsparameter von leistungskritischen Programmabschnitten. Durch Auto-Tuning kann eine hohe Leistung auf einzelnen Hardwareplattformen und auch Leistungsportabilität erreicht werden.

In dieser Arbeit präsentieren wir das Framework AutoTuneTMP. AutoTuneTMP besitzt zwei zentrale Eigenschaften, die effizientes Auto-Tuning ermöglichen: just-in-time (JIT) Kompilation und C++ Template-Metaprogrammierung. JIT Kompilation ermöglicht eine enge Integration von Auto-Tuning in eine Applikation und reduziert die für das Auto-Tuning benötigte Zeit. Template-Metaprogrammierung verbessert die Integrierbarkeit, Wartbarkeit und Erweiterbarkeit. Weiterhin ist es die Basis für Optimierungstemplates, welche die Entwicklung von Algorithmen unterstützen, die mittels Auto-Tuning optimiert werden können. Zudem stellt AutoTuneTMP Suchstrategien zur Verfügung, die zur automatischen Parameterwahl genutzt werden können.

Um die Anwendbarkeit und den Nutzen von AutoTuneTMP nachzuweisen, zeigen wir, dass Auto-Tuning die Leistung von drei Algorithmen verbessert. Bei allen drei Algorithmen ist das Erreichen einer hohen Leistung für akzeptable Laufzeiten notwendig. Als ersten Algorithmus betrachten wir die Matrixmultiplikation. Hochleistungs-Implementierungen für diesen Algorithmus werden oft manuell optimiert oder in Assembler programmiert. Anstatt ebenfalls auf niedriger Abstraktionsebene zu arbeiten, nutzen wir Template-Metaprogrammierung für eine Implementierung auf einer höheren Abstraktionsebene – und natürlich Auto-Tuning. In einer Evaluation über vier Hardwareplattformen hinweg konnten bis zu 91% der theoretischen Maximalleistung der Hardware gemessen werden. Damit erreicht der vorgestellte Ansatz das Leistungsniveau von hochoptimierten Herstellerbibliotheken wie Intels MKL.

Dünne Gitter sind gut geeignet für das maschinelle Lernen von moderat-dimensionalen Datensätzen im Umfeld von Big Data. Allerdings erfordern große Datensätze entsprechende Hochleistungsalgorithmen. Wir stellen zwei derartige Algorithmen für dünngitterbasierte Regression vor: den Unified-Streaming-Algorithmus und den Subspace-Algorithmus. Da beide Algorithmen Auto-Tuning mittels AutoTuneTMP unterstützen,

dienen sie als zweites und drittes Beispiel für die Verwendung von AutoTuneTMP. Der Unified-Streaming-Algorithmus wurde in OpenCL geschrieben und unterstützt eine Vielzahl von Hardwareplattformen inklusive Grafikprozessoren. Der Subspace-Algorithmus wurde ausschließlich für Prozessoren entwickelt, er hat dafür jedoch eine bessere Zeitkomplexität. Durch diese beiden Algorithmen und einem neuen Ansatz für die räumliche Adaptivität von dünnen Gittern konnte eine bis zu 13-fache Beschleunigung erreicht werden. Als Vergleich diente der moderne Masked-Streaming-Algorithmus mit einem überschussverfeinerten dünnen Gitter. Die Ergebnisse zeigen weiterhin, dass beide neuen Algorithmen leistungsportabel sind.

Zusätzlich zu den Auto-Tuning-Beiträgen wird die Leistungsportabilität einer neuen verteilten Variante des Clusteringalgorithmus für dünne Gitter untersucht. Diese dichtebasierte Clusteringmethode nutzt eine Dünngitter-Dichteschätzung, um Dichtefunktionen für große Datensätze von moderater Dimensionalität zu bestimmen. Im Vergleich zu anderen Algorithmen besitzt die dünngitterbasierte Dichteschätzung eine lineare Komplexität in der Größe des Datensatzes. Dünngitterbasiertes Clustering besteht aus vier Unteralgorithmen: zwei für die Dichteschätzung und zwei für das Erstellen und Reduzieren eines k-nächste-Nachbarn Graphs. Für die Implementierung wurde auf OpenCL und MPI zurückgegriffen. Auf der Ebene einzelner Rechenknoten erreichten wir durchschnittlich 79% der erreichbaren Maximalleistung in Experimenten auf einem Prozessor und vier Grafikkarten. In verteilten Experimenten auf zwei Supercomputern, Piz Daint und Hazel Hen, maßen wir bis zu 352 TFLOPS bei Nutzung von 128 Rechenknoten. Die ähnliche hohe Effizienz hinsichtlich der Maximalleistung auf beiden Supercomputern und die hohe Effizienz auf einzelnen Rechenknoten zeigt die Leistungsportabilität des Algorithmus.

Diese Arbeit zeigt, dass Leistungsportabilität ein realistisches Ziel für wissenschaftliche Anwendungen auf moderner Hardware ist. Der präsentierte eng-verzahnte Auto-Tuning-Ansatz ermöglicht durch JIT Kompilation und Template-Metaprogrammierung einen geringeren Aufwand für die Optimierung, ohne dabei Kompromisse bei der erreichbaren Leistung zu machen.

Contents

1. Introduction

In earlier days of high-performance computing (HPC), less sophisticated compilers and hardware with comparably low performance necessitated labor-intensive manual performance optimization. However, as compilers and hardware platforms have become more sophisticated, certain types of optimizations are no longer or only rarely needed. For example, compiler heuristics for loop unrolling perform loop unrolling automatically. Additionally, superscalar processors can overlap instructions of the loop with instructions in the loop body. Thus, manual loop unrolling is rarely beneficial.

However, despite clear progress, efficiently using the resources of modern hardware platforms still entails tedious manual optimization. This issue is most pressing with regard to the multi-layer cache hierarchy implemented by basically all modern high-performance architectures. Despite caches being invisible in an application code, data locality is required for the cache to effectively reduce the reads and writes to the main memory. However, rewriting an algorithm into a form that is semantically equivalent but expresses sufficient data locality is generally beyond the scope of modern compilers.

Apart from high performance on a single platform, performance portability is often a highly-desirable property. We preliminarily define it as achieving high performance across a set of hardware platforms considered. Performance portability is important, as most scientific applications are run on different hardware platforms. For example, applications are usually run on future generations of server processors. Furthermore, graphics processors (GPUs) and other types of accelerators are now widely used and, therefore, need to be efficiently supported by applications. However, the microarchitecture of GPUs is more different from a processor than, for example, a processor and its next-generation successor. Consequently, performance portability across standard server processors and GPUs has proven to be a challenging requirement.

There is a tension between high node-level performance and performance portability. Applications that achieve excellent node-level performance are usually heavily optimized. To be effective, these optimizations target specific low-level features of the underlying hardware platform. However, performance portability entails high-performance across

hardware platforms with different low-level characteristics. To address this issue, a common approach is the creation of multiple code paths for each hardware platform. While it is clear that with enough effort performance portability can be achieved with this approach, the disadvantages are rather obvious as well: the resulting application is more difficult to maintain, more difficult to extend and each additional hardware platform might require yet another implementation. For these reasons, a single code path is generally preferable.

In this work, we show that high node-level performance and performance portability can both be achieved through auto-tuning. Through auto-tuning, applications can adapt to the underlying hardware platforms automatically in order to maximize performance. Again, this definition should be considered preliminary, we introduce auto-tuning in more detail in Sec. 2.2. Our work extends a well-known basic approach for auto-tuning: the writing of portable compute kernels that expose parameters which are tuned by parameter tuners that implement search strategies. By auto-tuning parameterized compute kernels, only a single code path is needed. As the parameterization makes a compute kernel more adaptable, supporting additional hardware platforms might only require an automated run of the parameter tuners. Or, if the compute kernel needs to be adjusted, the improved compute kernels might exhibit improved performance for already supported other platforms as well. Therefore, auto-tuning enables high-performance and performance portability at reduced development effort, especially with regard to the effort for supporting additional hardware platforms and, more general, maintainability and extensibility.

Having argued that auto-tuning provides significant advantages, this raises the question of how to enable auto-tuning in applications. To that end, we introduce the auto-tuning framework AutoTuneTMP. AutoTuneTMP supports two types of auto-tuning which are both investigated in this thesis. A just-in-time-compiled (JIT-compiled) C++-based approach that not only tunes compute kernels, but also provides an approach for writing parameterized compute kernels using template metaprogramming. As writing parameterized compute kernels is a non-trivial task, this addresses a major obstacle towards a wide-spread adoption of auto-tuning. The second type of compute kernel considered in this thesis are compute kernels written in OpenCL. For this compute kernel type, AutoTuneTMP primarily acts as a parameter tuner. By implementing compute kernels in OpenCL, we can write compute kernels with arguably the widest-possible portability compared to other runtimes and languages available. Thereby, we can show the usefulness of auto-tuning across a wide range of hardware platforms that include pro-

cessors as well as graphics cards. While we differentiate between these approaches, we remark that the OpenCL 2.2 standard includes C++ as a compute kernel language [147]. With the availability of implementations of this version of the OpenCL standard, it will be possible to merge both types of auto-tuning. Compared to prior approaches, AutoTuneTMP provides a further benefit. Thanks to C++ template metaprogramming and JIT compilation, it allows for a seamless integration of auto-tuning into an application. This is a major advantage especially compared to prior approaches that rely on external tuning programs. The tight integration of auto-tuning with an application contributes to improved developer productivity, maintainability and extensibility. Finally, as we employ JIT compilation, compute kernels can be tuned at application runtime, i.e., AutoTuneTMP supports online auto-tuning. Online auto-tuning is supported by few auto-tuning frameworks, as we discuss in Sec. 2.2.

We present three auto-tuned algorithms that achieve excellent performance relative to their competitors or a performance model. Further, we show that all of them are performance-portable. These algorithms are intended to demonstrate the usefulness and applicability of AutoTuneTMP. As the first algorithm, we consider an auto-tuned dense matrix multiplication. The auto-tuned matrix multiplication has the most challenging parameter space of the algorithms considered. Thus, we use it to evaluate different search strategies built into AutoTuneTMP.

As further auto-tuned algorithms, we present one extended and one new algorithm for sparse grid regression. The sparse grid method is a grid-based approach for spatial discretization that mitigates the curse of dimensionality. While sparse grid regression is well-suited for learning large moderate-dimensional datasets, achieving near-optimal performance is still paramount, as the ever-increasing sizes of datasets are challenging even for methods with competitive time complexity. We show that auto-tuning enables the desired level of performance. Both algorithms we present not only serve as examples for auto-tuning, we show that they improve on the state-of-the-art in sparse grid regression as well. As the first algorithm, we present an extended variant of the sparse grid streaming algorithm. Due to its design and the use of OpenCL, this algorithm is suited for processors and GPUs. The second algorithm is called the subspace algorithm and was designed for processor platforms. Its lower complexity enables higher performance compared to the streaming algorithm, but for a more limited range of hardware platforms.

In addition to showing the value of AutoTuneTMP, we further present the distributed sparse grid clustering algorithm as a study in performance portability. This algorithm

is a new distributed high-performance variant of the sparse grid clustering algorithm by Peherstorfer et al [114]. Algorithmically, it is a density-based clustering method that employs a sparse grid density estimation for efficiently targeting higher-dimensional datasets. By using OpenCL and MPI, we demonstrate performance portability across two supercomputers and five node-level platforms.

Beyond optimization and performance portability, we introduce a new datadriven approach for constructing spatially-adaptive sparse grids called support refinement. Commonly, sparse grids are refined by iteratively extending an initial grid. These approaches solve the targeted problem in each step of the refinement process. With support refinement the problem only needs to be solved once, as it is a solverless refinement approach. We show that support refinement accelerates both sparse grid regression and clustering compared to the state-of-the-art surplus refinement approach.

As the last step of this introduction, we briefly outline the structure of this thesis. In Part I, we define performance portability (Chapter 2.1) and auto-tuning (Chapter 2.2); and we describe approaches to both of these concepts. As achieving high-performance is a goal of all the algorithms discussed in this thesis, we give a brief overview of recent trends in node-level hardware and discuss concepts important in node-level optimization on modern processor and graphics card platforms (Chapter 3).

In Part II, we introduce our auto-tuning framework AutoTuneTMP (Chapter 4). As part of its introduction, we compare it to other auto-tuning frameworks. The auto-tuned matrix multiplication is introduced and evaluated thereafter (Chapter 5).

Part III brings together auto-tuning, performance portability and sparse grids. We first introduce the basic sparse grid theory and spatially-adaptive sparse grids (Chapter 6). Then, we consider sparse grid regression and the two auto-tuned regression algorithms (Chapter 7) Finally, we investigate the performance portability of the distributed sparse grid clustering algorithm (Chapter 8). At the very end, in Chapter 9, we summarize the results of this thesis and discuss directions of future research.

Part I.

Performance Portability and Auto-Tuning for Modern Hardware Platforms

In the following two chapters, we lay the foundation for the remainder of this thesis. We define important terms and describe the state-of-the-art in both performance portability and auto-tuning. As auto-tuning enables a partial automation of performance optimization, we further describe modern hardware platforms and how high performance can be achieved on these platforms.

Chapter 2 introduces both performance portability and auto-tuning. We first define performance portability and describe approaches for enabling performance portability in applications. Furthermore, we report on some aspects of software and hardware that limit performance portability. We proceed by defining auto-tuning and explain why auto-tuning is useful—especially in HPC. Then, we give an overview of the approaches to auto-tuning that are described in the literature.

In Chapter 3, we describe the architecture of modern hardware platforms with a focus on aspects that are relevant for performance optimization. We consider trends in hardware in the last two decades and their implications for performance optimization. Afterwards, we discuss two specific microarchitectures relevant in HPC: the Intel Skylake architecture and the Nvidia Pascal architecture. Finally, we describe the hardware platforms we used for evaluating our algorithms.

2. Performance Portability and Auto-Tuning

2.1. Performance Portability

An intuitive definition of performance portability is the ability of an application to perform well on a wide range of hardware platforms. However, performance portability is rarely defined concisely and definitions vary in the literature. Even in research on performance-portable software and performance portability frameworks, either the intuitive definition is used or a definition is provided with little discussion [133, 42, 117, 130, 52, 107, 43, 80, 115, 165, 125, 43]. Only few papers discuss the concept of performance portability in detail [116, 167]. In the following, we present the definition of performance portability by Pennycook, Sewall and Lee in slightly rephrased form [116], as their attempt at defining the basic concept seems to be the most advanced. Then, we describe our position towards performance portability.

For performance portability to be possible, functional portability is a prerequisite. Pennycook et al. define this software property as follows:

Definition 2.1.1 (Functional Portability)**.** The ability of an application a to execute a problem p correctly on a given set of platforms H.

Throughout this work, we will refer to functional portability simply as portability. Other definitions in the literature stress the effort required to make an application run correctly on a different hardware platform or operating system, cf. Tanenbaum et al [142]. The definition by Pennycook et al. instead requires the specification of a set of platforms on which an application actually runs.

Apart from portability, we further need to define performance.

Definition 2.1.2 (Performance)**.** A measurement of an application's correct execution of a problem p on a platform h using a given performance metric.

A performance metric can be anything measurable about a program, including standard HPC performance metrics like application runtime, floating-point operations rate and achieved memory bandwidth.

The following definition of performance portability attempts to capture the general use of performance portability.

Definition 2.1.3 (Performance Portability)**.** For a given application a portable across platforms H for problem p, a performance is achieved on all platforms that satisfies stated performance goals or is useful for a given purpose.

This definition restricts the scope of portability claims to what can actually be evaluated, e.g., a set of problems and a specific set of platforms. It thereby highlights that abstract performance portability claims for applications or across an unspecified range of devices are extrapolations. Because performance portability has been used for different purposes, the criterion to accept a level of performance as "good enough" is intentionally left to be defined in the specific work. This reflects that acceptable performance can vary widely. It can be required to reach a similar time-to-solution, a similar floating-point rate or a similar memory bandwidth. For a given metric, what constitutes a similar performance must be further specified. In other contexts, acceptable performance might be just barely fast enough for an application scientist to consider the application applicable to a scientific problem.

Because this definition in itself is not measurable and in part subjective, Pennycook et al. further introduce a measurable and objective performance portability score.

Definition 2.1.4 (Performance Portability Score)**.** For a given application a portable across platforms H for problem p and a performance efficiency map e, the performance portability score $\mathcal{P}_{a,p,H,e}$ is given by

$$\mathcal{P}_{a,p,H,e} := \frac{|H|}{\sum_{h \in H} \frac{1}{e(a,p,h)}}. \tag{2.1}$$

This definition calculates the harmonic mean of an achieved performance efficiency. A performance efficiency map e is a performance metric normalized to $[0, 1]$.

Definition 2.1.4 uses the harmonic mean in contrast to the arithmetic mean for averaging across H. By using the harmonic mean $\mathcal{P}$ approaches zero if the performance on a single hardware platform approaches zero. Therefore, it is more challenging to achieve performance portability scores close to one. If the arithmetic mean would be

used instead, performance portability would only be reduced by $\frac{1}{|H|}$, hiding the fact that an application it not actually performance-portable across H.

Pennycook et al. give two examples of performance efficiency maps: architectural efficiency and application efficiency. Architectural efficiency refers to the performance relative to limitations of the hardware. For example, for a maximum floating-point throughput $f_h^{\max}$ on platform h and an actual floating-point throughput $f_{a,p,h}$ the architectural efficiency can be defined as

$$e_{\text{arch}}(a, p, h) := \frac{f_{a,p,h}}{f_h^{\max}}. \tag{2.2}$$

Notably, as $f_h^{\max}$ is application-dependent, it can be much smaller than the peak floating-point throughput. To make use of architectural efficiency, a performance model that estimates an upper bound for the performance of an application is usually required.

Architectural efficiency should only be used if the application uses algorithms that fit well to the hardware architecture, i.e., the application runtime is competitive compared to other algorithmic choices on each hardware platform. Otherwise, it would be possible to generate artificially inflated performance portability scores, e.g., by doing redundant floating-point computations.

As an alternative to architectural efficiency, Pennycook et al. introduce application efficiency. Application efficiency compares the observed performance relative to the best observed performance measured on each platform. The best observed performance might have been achieved with a different application and different algorithms. Therefore, compared to architectural efficiency, application efficiency can account for different optimal algorithms across hardware platforms. To ensure that the comparison across algorithms and platforms is fair, the performance metric should only depend on the application and the measured runtime. For example, time-to-solution can be used directly and "rays per second" could be relevant for a ray-tracing application; many more examples are given by Satish et al [135].

Pennycook et al. showed that the performance portability score according to Def. 2.1.4, when retrospectively applied to published performance portability research, tracks well the performance portability claims of the authors whose published results were used [116]. This provides evidence that the proposed performance portability score actually captures performance portability as used by other researchers.

The given definition of performance portability does not pose any restrictions on the type of device for which performance portability is considered. An application can

be performance-portable for H ranging from processors to GPUs to FPGAs to different types of accelerators. Or, it can be performance-portable across different micro-architectures of processors. Both might be valid ranges of devices, as even closely-related hardware can exhibit major performance differences. For example, the Intel Haswell microarchitecture implements two vector pipelines for fused-multiply-add (FMA) instructions, doubling the vector arithmetic throughput compared to its predecessor. This can be used to obtain a 2x speedup for a well-optimized GEMM kernel. Therefore, even closely-related hardware platforms can be an interesting field of study in performance portability research.

In this work, we generally make use of the definitions presented by Pennycook et al. with, however, one exception: we do not compute the performance score $\mathcal{P}_{a,p,H,e}$. Instead, we provide per-device performance measurements, e.g., in GFLOPS or in adjusted fraction of achieved peak performance, and draw qualitative conclusions from the provided data. We recognize that a qualitative discussion has the drawback of being subjective. However, a critical analysis of the assessment is possible as the per-device results are provided. We argue that the performance portability score does not provide a clear benefit over this approach. As an example for the usefulness of this metric, Pennycook et al. themselves only mention that $\mathcal{P}_{a,p,H,e}$ can be used to assess changes to one of the variables, such as changing H. Indeed, that qualitative discussions are the state-of-the-art in analyzing performance portability might indicate that researchers so far see no benefit in the aggregate metrics proposed.

2.1.1. Why Performance Portability?

Performance portability is important whenever performance is relevant and more than one hardware platform is targeted by an application. This is generally the case in fields such as machine learning, image processing and, of course, HPC. Performance portability is important for two central reasons: to support a range of current hardware platforms efficiently and to be forward-compatible with future hardware. We address these reasons from an HPC perspective.

Supercomputers use a wide range of different hardware architectures. For example, the Top500 list of December 2018 includes standard x86 processors, PowerPC, SPARC and more exotic HPC-specific platforms like the Sunway SW26010. Many systems are heterogeneous and use accelerators, often graphics cards like Nvidia's Tesla V100, but also pure compute accelerators like the Matrix-2000 employed by the Tianhe-2A supercomputer [151]. HPC applications are written so that they are portable across a range of

hardware platforms. However, performance is the very requirement that all HPC applications share. Therefore, what is actually needed is high performance across the range of supported devices. That is to say that applications should be performance-portable.

Furthermore, computer hardware changes faster than software (argued for example by Püschel et al [126]). Software is often used productively for decades, whereas hardware is upgraded every few years. Consequently, it is a common requirement for software that it should be usable on upcoming and future hardware platforms. Again, the application needs to be portable and, for it to be useful, it needs to achieve high performance. That is, performance portability to future hardware platforms is strongly desired.

HPC applications might not be portable or performance-portable immediately. Instead, applications usually require adjustments for supporting additional hardware platform. Especially challenging is the addition of support for future hardware. This is only possible in rare cases, e.g., if an emulator is available. While minor adjustments are acceptable with respect to the effort required, rewriting major parts of an application is generally not. As developers are aware of this issue, applications are often structured so that they are not too dependent on platform specifics. This again leads to considerations relevant for performance portability. For example, programmers need to consider whether the vector width can be changed for a future architecture. Or, whether a cache blocking approach will work on a memory hierarchy that is structurally different.

2.1.2. Approaches to Performance Portability

In this section, we introduce four common strategies for achieving performance portability. First, directly implementing a performance-portable application to enable performance portability for a specific application. Second, by providing a domain-specific language (DSL) that has some performance portability across a range of platforms for a whole class of problems. Third, highly-portable languages and language extensions that provide some degree of performance portability. And fourth, performance portability frameworks and parallel runtime systems that enable application developers to create performance-portable software. Runtime systems for distributed environments can be considered an important subclass of the last category. In the remainder of this section, we discuss these approaches in more detail.

Performance portability can be achieved through direct implementation of an application using standard tools and libraries. Well-known examples for this class of approach are the BLAS library ATLAS [158] and the FFT library FFTW [57]. Both use auto-tuning to achieve optimal performance and contributed to the recognition of auto-tuning

as a useful approach in HPC. Three other examples for this type of approach are the BLAS libraries PHiPAC, clBLAS and BLIS. PHiPAC was an early pioneer in auto-tuning and featured performance portability across hardware platforms of the late 1990s [18]. As the name suggests, clBLAS is a portable BLAS library implemented in OpenCL [33]. Due to the use of OpenCL it supports a wide range of devices. BLIS is a BLAS implementation for processors that achieves performance portability through model-based parameter selection [96]. Both, clBLAS and BLIS are recommended (and seem to be partially developed) by AMD.

The performance and performance portability achieved with this type of approach is usually excellent. However, performance-portable applications that use the direct implementation approach often are technically highly-complex. As implementation strategies, some applications use multiple code paths for different architectures, possibly even using different programming languages. Other applications adjust the source code to specific hardware platform through extensive conditional compilation. These techniques can increase the effort required to maintain, extend and test performance-portable applications.

In recent year, many DSLs have been proposed for a wide range of fields ranging from source code analysis [89] to landscape dynamics modeling [48] to machine learning [139]. A number of DSLs have been developed with performance portability as an explicit goal. Many of these DSLs implement languages expressing stencils that occur when solving (partial) differential equations, examples for this type of application are OPS [130], Liszt [39], OP2 [107] (and PyOP2 [129]), PATUS [31], Pochoir [143] and STELLA [67]. Stencils occur in other applications as well. For example, Halide is a performance-portable DSL that generates stencils for image processing [127]. Of course, some performance-portable DSLs target different domains entirely. Spiral generates highly-optimized and performance-portable signal processing algorithms, newer version of Spiral use a DSL called OL [126, 55]. Others help application researchers to more generally express mathematical ideas at a higher level, so that efficient, i.e., parallel and cache-aware, code can be generated automatically. Examples of this category include Sequoia [52] and ∇ [27].

To achieve performance portability, different DSLs use different strategies. Some DSLs introduce their own languages [52, 39, 31, 127, 27]. Others appear as libraries written in a standard programming language [143, 107, 129, 130, 67]. Even of the DSLs written in a standard programming language, most are implemented with a custom compiler to generate code and perform optimization on the abstract-syntax-tree (AST) level. In

contrast, STELLA uses C++ template metaprogramming to generate efficient code [67]. Embedding a DSL into standard languages like C++ can make it easier to integrate the DSL into a project. Specifically, C++ template metaprogramming has been used to implement the API of the DSLs Pochoir [143], OPS [130] and STELLA [67].

Some mentioned DSLs provide performance portability in a distributed setting [52, 39, 107, 67, 55]. OP2 in particular addresses challenges in distributed computing like communication cost and the locality of data. However, most performance-portable DSLs focus on node-level performance challenges.

Highly-portable languages and language extensions support application developers in writing performance-portable applications. OpenMP is probably the most widely-used threading framework in scientific computing [112]. With version 4.0, OpenMP has been further extended to support vectorization through the `#pragma omp simd`-construct. The same version of the OpenMP standard added support for offloading work to accelerators. By using these features, OpenMP supports the implementation of portable, parallelized and vectorized applications. Unfortunately, more-recent OpenMP features like offloading were not fully adopted by major compilers at the time of writing.

OpenCL is likely the most portable language available, with implementations on platforms ranging from x86 standard processors to GPUs to the Intel Knights Corner Xeon Phi accelerator and even FPGAs. For this reason, OpenCL is a promising candidate for developing performance-portable application. Research has shown that OpenCL provides some degree of performance portability [49, 115, 40, 117, 165, 130, 125]. While early evaluations gave mixed result for OpenCL's performance portability, McIntosh-Smith et al. argue that OpenCL's performance portability has improved over the last years with more mature OpenCL implementations [103]. OpenCL has been used to implement many applications. Well-known examples include Bullet, GROMACS, LAMMPS and CLBlast [109].

There are few alternatives to OpenCL's portability if the target platforms include not only processors, but also accelerators. Herdman et al. argue that OpenACC is a viable alternative to OpenCL, as it offers higher productivity and has some degree of portability [78]. For example, the OpenARC compiler is an OpenACC implementation that enables the use of OpenACC across platforms of most hardware vendors relevant in HPC, though at this point it is not publicly available [94, 134].

The Intel SPMD Program Compiler (ISPC) creates a thread- and vector-parallel program from a language close to a scalar C program [123]. It is similar to OpenCL and OpenACC in its ability to leverage thread- and vector-level parallelism. ISPC supports

all major vector extensions of the x86 ISA. As ISPC only targets standard processor architectures, it can be easily integrated with programs built using standard programming languages. For example, ISPC can directly operate on memory allocated in C.

Since C++11 was introduced, performance portability frameworks and language extensions of C++ have been developed. Notable examples for this type of frameworks are RAJA [80] and Kokkos [43], In their features sets, RAJA and Kokkos are very similar. RAJA focuses on the expression of parallelism on a higher level, so that low-level thread management is handled internally by RAJA's C++ templates. These templates can be considered extensions of the C++ standard template library's (STL) `algorithm` header. The developers stress the usefulness of C++ lambdas, as they allow for the separation of a loop body and the management of the index set of a loop. RAJA supports multiple back-ends for portability with minimal code modification. It also provides abstractions for different memory layouts, including support for tiled memory access patterns. Kokkos focuses on data structures for efficient memory accesses. To that end, Kokkos implements high-performance data structures for HPC application, e.g., for efficiently iterating a multi-dimensional data structure that resides in a linear address space. Kokkos chooses the type of memory layout to be used at compile-time depending on the hardware platform. The data structures that Kokkos implements are intended to be used with parallel algorithms that are also provided by Kokkos. Similar to RAJA, these parallel algorithms extend the functionality of C++'s STL.

While RAJA and Kokkos explicitly focus on performance portability in general, several frameworks focus on portable parallelization and vectorization. Intel Cilk Plus is a C++ runtime and language extension for thread- and vector-level parallelization [131]. It is built upon Cilk which only supported thread-level parallelism [20]. Intel Threading Building Blocks (TBB) is a runtime system for thread-level parallelism that provides a collection of parallel algorithms [132]. Very similar to TBB are the Parallel Pattern Library (PPL) and C++AMP, both developed by Microsoft. TBB and PPL only support processors, whereas C++AMP works on GPUs and processors [65]. A research project with a similar scope and support for GPUs and processors is SkePU [44]. Nvidia's Thrust provides host-callable algorithms that exclusively run on GPUs [15]. These frameworks either implement STL-like algorithms or use a functional style using C++'s lambdas. Furthermore, all of them appear as C++ libraries to the programmer.

As vectorization has proven challenging to address at high levels of abstractions, libraries for portable vectorization have been developed. Vc [92, 91] and Boost.SIMD [46] are both C++ libraries for explicit vectorization. Both are portable in the sense that

they can infer the native vector width from compiler settings and trigger the compiler to generate SIMD code. Both libraries expose an API that is very close to scalar C++ and both provide means for handling vector-conditionals elegantly. Vc and Boost.SIMD target processor architectures only.

Performance Portability in a Distributed Setting

In this work, we primarily consider node-level performance portability challenges. Performance portability across distributed systems leads to widely different problems to be solved: interconnect bandwidth and latency, network topology and data placements on specific nodes all have to be considered. For comparison, excellent node-level performance requires a high utilization of the available compute resources, cores and vector units, and a consideration of memory bandwidth and memory latency. Furthermore, as modern supercomputers consist of many smaller nodes with (more or less) commodity hardware, performance portability on individual nodes is foundational to performance portability across supercomputers. However, because performance portability is an important challenge in distributed computing, we briefly describe how is can be achieved.

MPI is the most commonly used library for developing distributed applications [105]. Two of the major MPI implementations, OpenMPI and MPICH, both target portability and high performance [113, 106]. Porting MPI applications to new hardware platforms tends to require some adjustments, often due to differences in interconnects and a deviating network topology. However, some scalability can be expected without major application rewrites. Therefore, MPI can generally be considered performance-portable with respect to process-level parallelization and scalability.

An increasingly important class of frameworks in distributed settings are asynchronous many task systems (AMTs). Modern AMTs like HPX [85] and CHARM++ [86, 2] can be used for portable applications that scale from small mobile systems to large distributed systems such as supercomputers [77]. AMTs create a task graph that describes dependencies between tasks. The details of the execution are determined by the runtime system. This might include the order in which tasks are processed or the selection of the node that executes the task. Another key concept of AMTs is the mostly uniform treatment of local and distributed data, thereby residing on a higher level of abstraction compared to MPI. Research on AMTs is a highly-active field. STAPL [26], LEGION [13] and StarPU [10] are all competitors of HPX and CHARM++ with similar features.

2.1.3. Limits of Performance Portability

Even high-performance application codes that utilize the hardware resources to the extent possible explicitly or implicitly make assumptions about the architecture of the hardware platforms targeted. If these assumptions are invalid on an additional hardware platform, even an application that displayed excellent performance portability might be significantly less efficient. As an example, we consider the porting of an astrophysical application from the Intel Haswell to the Intel Knights Landing platform - a porting effort to which the author of this thesis contributed [121]. The application, called Octo-Tiger, simulates the merger of binary star systems. In astrophysics, these mergers are highly important, as they lead to supernovae of type Ia, luminous red novae and the formation of R Coronae Borealis stars [156, 32, 152]. The computationally most intensive components of this application are a set of three compute kernels that are part of the fast-multipole method (FMM) that Octo-Tiger uses.

Before our porting effort, the execution of these kernels consisted of three steps: a gather step that aggregated data from memory into arrays, many arithmetic instructions on these arrays and, finally, a scatter step for writing the data back to memory. The compute kernels ran efficiently on Intel's Haswell platform, despite the potentially expensive gather and scatter steps. On this platform, the kernels were primarily bound by the arithmetic instructions and achieved a significant fraction of the peak performance. The Knights Landing platform is superficially similar to the Haswell platform, as it implements nearly the same x86-based instruction set. Therefore, one could assume reasonable performance on the Knights Landing platform. However, the microarchitecture of the Knights Landing differs in critical aspects. Relevant for Octo-Tiger was the much lower scalar performance of the Knights Landing architecture due to a lower frequency, smaller caches, fewer out-of-order resources and other factors. As a consequence, the original algorithm achieved less than 5% of the floating-point peak performance, despite the compute kernels being compute bound. On this platform, the scatter and gather steps turned out to be the primary cause for the low performance, as these were implemented as many scalar `MOV` instructions. Note that address indirections precluded the use of the more efficient AVX512 gather/scatter instructions that are supported by the Knights Landing architecture.

In our effort to establish performance portability, we redesigned the compute kernels. The new kernels employ a stencil approach with a very large 1074-element stencil. This stencil approach eliminates the gather and scatter steps. The new algorithmic approach enabled an approximately 5x speedup for the FMM kernels compared to the

prior approach. Incidentally, the continuous memory access patterns of the revised kernels improved performance on the Haswell platform as well.

This example shows that an algorithmic approach that was efficient on a class of hardware platforms, multiple generations of Intel Xeon processors, still cannot guarantee high performance even on a related platform. However, it also shows how a modification of the algorithm again enabled performance portability across the extended range of platforms. In further work, we showed that the same stencil approach could even be reused in GPU kernels. As a result, Octo-Tiger could be run at scale on the GPU-based supercomputer Piz Daint [35].

2.2. Auto-Tuning

Auto-tuning, sometimes known as empirical optimization or self-tuning, can be defined as follows.

Definition 2.2.1 (Auto-Tuning)**.** Auto-tuning is the ability of an application to adjust itself automatically to the underlying hardware platform or input features in order to maximize some metric, usually performance.

While there are different approaches to auto-tuning, there is a common basic strategy. Generally, variants of applications or compute kernels of applications are considered. These variants are either specified through a set of parameters or they are given by a set of (code) transformation rules and an initial problem statement. The goal of the auto-tuning process is to discover the variants that maximize performance. As the performance characteristics of the variants often cannot be inferred analytically, all or a subset of variants are investigated empirically. The set of variants can be considered as a search space and the discovery of the highest-performance variants can be formulated as an optimization problem. As the search spaces can be vast, search strategies are employed that, if effective, consider only few relevant variants.

Performance, of course, can refer to different quantities of interest. It, however, usually refers to time-to-solution or a metric related to time-to-solution such as GFLOPS (for a fixed hardware platform and input). While rare in the literature, goals other than performance have been considered. For example, Anzt et al. performed auto-tuning to maximize power efficiency [8].

In the remainder of this section, we first describe the two issues that auto-tuning addresses. We explain how auto-tuning can improve performance and reduce the effort

for performance optimization. Then, we consider auto-tuning as a means for developing performance-portable algorithms. Having stated our goals, we proceed by introducing the most important approaches to auto-tuning. Finally, we characterize the search spaces and search strategies commonly used.

2.2.1. Auto-Tuning for Productivity and Performance

There are two widely-used strategies to create high-performance implementations: relying on an optimizing compiler and optimizing manually. We argue that these strategies either do not yield optimal performance or lead to an often unacceptably high effort. By using auto-tuning techniques, performance can be improved with reduced effort.

Optimizing compilers have made significant progress in the last decades. As a result, programmers are often advised not to manually optimize their programs, as this step is handled by the compiler. However, especially in high-performance computing the performance obtained by simply using an optimizing compiler is not sufficient. Compiler are subject to constraints when deciding whether performance optimizations can be applied.

- Compiler optimizations need to preserve semantics as specified in the language description.
- Compilers have to be conservative if the correctness or benefit of an optimization cannot be proven.
- Optimizations have to be effective, i.e., actually improve performance, even for unexpected or unusual input.
- In standard programming languages, performance-critical knowledge might not be represented in the program.

These constraints limit what compilers can achieve and, as a result, certain critical optimizations are generally not performed.

Pointer aliasing in C++ highlights some of the issues compiler face because of the constraints stated above. In C++, pointers are allowed to point to the same memory address (though some restrictions apply) [162]. As an example, we assume two pointers that point to the same address and the value pointed to is changed through the first pointer by a store instruction. If the value pointed to is held in a processor register because of a read access through the second pointer, the value needs to be reloaded,

as the register now contains the incorrect former value. Generally, whether a reload is required needs to be inferred by the compiler through alias analysis. Ruling out pointer aliasing can lead to significantly faster code because of fewer generated load and store instructions. However, if the absence of aliasing cannot be proven, the compiler has to be conservative and the generated code might exhibit lower performance, i.e., it misses an optimization.

C99 introduced the `restrict` keyword, which enables programmers to explicitly state that pointers do not alias [161]. However, there is no comparable feature in the C++ standard [162]. Thus, pointer aliasing in C++ is one of the cases where details relevant for optimal performance cannot be represented in the programming language.

In addition to the stated constraints, optimizing compilers are limited by more practical requirements:

- Languages that are compiled ahead-of-time only allow for static analysis.
- The compiler might only see parts of the program and therefore only optimizes the program locally.
- The compilation itself should be fast. This is especially important for just-in-time compiled languages.

These requirements all make it more difficult for a compiler to create near-optimal code and therefore help to explain why performance might be lower than expected. However, research and implementation efforts for techniques such as profile-guided optimization and link-time optimization might resolve or mitigate these issues in future compilers.

The costs of missed optimizations depend on the optimizations considered and can be very high, especially for HPC compute kernels. For example, matrix multiplication on modern x86 processors can be implemented so that the innermost loop uses vector fused-multiply-add (FMA) instructions. Modern compilers can in some cases perform auto-vectorization. However, a compiler might not auto-vectorize a matrix multiplication implementation and instead use scalar instructions. On modern processors such as Intel's Skylake-SP series, performance can be reduced by a factor of up to 32 (double precision) or 64 (single precision) [93].

Missing memory optimizations can be even worse, as an efficient use of the memory often is a prerequisite for other optimizations like vectorization to be effective. As an example, we consider memory latency. Memory latency seems to no longer decrease, in fact it even increases by $\approx 4\%$ per year [102]. If a value can be cached in the L1

cache of a Intel Skylake processor, it requires four cycles to move this value into a (vector) register [54]. However, if the value instead gets loaded from memory more than a hundred cycles can be needed, resulting in significantly lower performance.

Because of the impact of important missed optimizations like cache blocking or vectorization, manual optimization is still commonplace in HPC. However, the downsides of manual optimization are rather obvious. The effort to optimize compute kernels can be very high. Expert knowledge on the hardware platform and high-performance programming might be needed for the optimizations to be effective. Moreover, the next generation of hardware can require the optimization process to be repeated.

In performance engineering, it is often clear that a compute kernel benefits from a certain optimization. Still, it is a time-consuming task to figure out how to exactly implement an optimized algorithm, as optimizations often expose performance-critical parameters that need to be tuned. For example, cache blocking requires the dimensions of the blocks to chosen carefully, as they strongly affect whether the optimization actually improves performance. Too small dimensions might lead to too little reuse of data loaded into the cache for overall reduced performance. If the dimensions are too large, data that is still needed might get evicted from the cache, affecting performance as well.

In case of manual optimization, a developer adjusts one or multiple parameters, recompiles the application and then evaluates whether the change led to improved performance. With auto-tuning, the optimization loop can be partially automated by providing search strategies and means to integrate auto-tuning into an application. Thereby, the amount of work needed is reduced.

Because experimenting with parameter values can be highly time-consuming, performance engineers often cannot explore all relevant or technically-plausible parameterizations. Furthermore, while an optimization expert might find parameter values close in performance to the optimal values, it is possible that a systematic search approach might find less intuitive, but higher-performance parameter values. Both these factors can explain why manually-optimized code might make use of subpar parameter choices and, therefore, delivers lower-than-possible performance. Therefore, an auto-tuning approach that systematically explores the search space with an appropriate search strategy can discover higher performance parameter values.

Furthermore, auto-tuning can bridge the knowledge gap between a programmer and an optimizing compiler or a second programmer. The compiler is generally not aware of the input of a function, e.g., a function might be called with matrices with widely varying dimensions. Consequently, the compiler might generate low-performance code

for a specific input. Similarly, the developer of a library might not know how a library routine is actually used. This can lead to a highly-optimized implementations that still displays low performance for a given input. With auto-tuning, the parameterization can be deferred to when the application is actually used. At this point, input representative of the actual use of the application can be made available to the auto-tuner. This enables the auto-tuner to adapt the application to the given input for improved performance. Some auto-tuning frameworks are even able to perform tuning during application runtime, i.e., they perform online auto-tuning. Thereby, even widely different input characteristics during an application run can be considered without the need to manually providing many variants of a compute kernel.

2.2.2. Auto-Tuning for Performance Portability

In Sec. 2.1, we introduced what performance portability is, why it is important and how it can be achieved. In this section, we explain how auto-tuning can be used to achieve performance portability across sets of different and even future hardware platforms.

We argued in the previous section that optimizations expose parameters that require tuning for optimal performance on a given hardware platform. This parameter tuning approach for optimization can also be used to support related hardware platforms. If two hardware platforms are similar enough, selecting slightly different parameter values through auto-tuning can already be sufficient to enable performance portability. For example, between the Intel's Broadwell and Skylake generation, the size of the L2 cache was increased from 256 kB to 1 MB. If a compute kernel implements parameterized cache blocking, adjusting the related parameters for larger caches is feasible purely by changing parameters. Thus, generational changes of processors can be taken into consideration.

With auto-tuning it is even possible to implement applications that achieve high performance on platforms of different vendors. Auto-tuned parameters that target common architectural features might immediately apply to a different platform. Because of trends of convergence in HPC hardware is has become more likely that if an optimization is needed on a specific platform, it is needed on a different platform as well. For example, modern x86 architectures like Intel Skylake and AMD Zen share most performance critical architectural features. They implement superscalar out-of-order cores with wide vector units, simultaneous multithreading (SMT) and a multi-layer cache hierarchy. Both platforms support (nearly) identical instruction sets as well. There are, however, many quantitative differences, e.g., differently-sized caches or differences in the width of the vector units. Fortunately, quantitative differences can be addressed by auto-tuning

parameter values. As we elaborate in Chapter 3, even processors and GPUs have many shared characteristics. A compute kernel that is portable across these architectures can therefore benefit from the auto-tuning of its parameters as well.

2.2.3. Approaches to Auto-Tuning

Auto-tuning was originally introduced as a feature of applications. Early examples from the late nineties include the BLAS implementation ATLAS [157] and the FFT library FFTW [57]. A slightly later well-known auto-tuned application is sparse matrix-vector multiplication library OSKI [155]. Because of the success of the earlier auto-tuned applications, the generalization of auto-tuning has become a topic of research.

Apart from applications, DSLs have been proposed that implement auto-tuning, following the example set by Spiral [126]. Some DSLs target the generation of efficient algorithms for stencil applications for solving partial differential equations, in computational fluid dynamics and in image processing [87, 31, 166, 98, 127]. Other DLSs more generally target the development of high performance and performance-portable compute kernels [52, 163, 7, 153]. Related to general compute DSLs are auto-tuning-enabled language extensions like HMPP [64] and OpenARC [134]. As auto-tuning can be used to enable performance portability there is a strong overlap between the DSLs mentioned here and those presented in Chapter 2.1.2.

As most implementations of auto-tuning rely on parameter tuning, several auto-tuning frameworks provide search strategies and abstractions for managing search spaces. Active Harmony possibly was the first parameter tuning framework created [144]. A major feature of Active Harmony is its ability to perform distributed auto-tuning, i.e., evaluating different parameterizations of an application on different nodes simultaneously. Another notable parameter tuner is OpenTuner, which is able to deal with search spaces with more than 200 dimensions [6]. Other parameter tuning frameworks are the Periscope Tuning Framework (PTF) and BEAST. The Periscope Tuning Framework (PTF) is notable for its extensibility [12], whereas BEAST can be used to optimize for energy efficiency [8, 97]. There are other frameworks that have slightly different goals. For example, SkePU focuses on selecting an appropriate parallelization backend from a list of predefined backends [44, 36].

Some auto-tuning frameworks not only tune parameters, but provide a code transformation approach to support the development of auto-tunable compute kernels. Active Harmony has been used together with CHiLL, which is a source-to-source compiler that acts as the code transformation component [29, 148, 150]. CHiLL implement loop trans-

formations like tiling and loop unrolling. Insieme is an auto-tuning framework and uses a source-to-source compiler to enable parameterized code transformations similar to those of CHiLL [84]. Apollo is an auto-tuning extension of the performance portability layer RAJA and uses machine learning instead of search heuristics to select the algorithm best-suited for a given input [14].

The Kokkos framework abstracts from the underlying hardware and promises to enable performance portability. In recent work, it has been extended with an auto-tuning component [62]. Kokkos uses C++ template metaprogramming for enabling the implementation of portable high-performance applications.

OpenCL has been combined with auto-tuning in many instances [109, 98, 153, 134, 64, 36, 138, 125, 90, 42, 133, 40, 117, 165]. In part because OpenCL exposes parameters (the structure of the grid, the size of the work group) that affect performance and need to be chosen with care. Furthermore, though OpenCL achieves some performance portability immediately, auto-tuning has also been used to bridge the remaining performance gap. To that end, three different strategies have emerged. Either a direct implementation strategy where parameters control conditional compilation, e.g., implemented with C preprocessor statements. This methodology is used by the CLBlast project [109]. Alternatively, code generators are employed that emit OpenCL source code, e.g., by Du et al [42]. These code generators expose parameters that can be tuned. A technically more sophisticated approach is employed by some DSLs and language extensions that support OpenCL as a back-end language. These DSLs use source-to-source compilation and therefore a compiler-level approach to create OpenCL kernels [98, 153, 134, 64, 138, 40].

Parameterized Types and Auto-Tuning Objectives

As the next step, we give an overview of the types of parameters that are targeted by auto-tuning. We discuss three types of parameters that are related to optimization. Additionally, we describe an auto-tuning approach that cannot be categorized as parameter tuning, but is relevant, as successful auto-tuned frameworks use it.

The first and most common parameter type controls performance optimizations. Examples for auto-tuned optimizations are

- cache blocking [109, 157, 158, 87, 31, 165, 98, 52, 7, 163, 134, 64, 149, 150, 148, 29, 138, 38, 125, 42, 95, 159, 166, 8, 68],
- register blocking [109, 157, 158, 87, 149, 38, 159, 68] and software pipelining [158,

159],

- loop unrolling [109, 157, 158, 31, 163, 134, 64, 149, 150, 148, 29, 133, 159, 68],
- loop interchange [163, 64, 149, 150, 29, 125, 159],
- selecting the data layout (array-of-structs/struct-of-arrays [165, 98], sparse matrix formats [155, 159], row-major/column-major matrix layout [165] and others [7, 125, 117]) and
- selecting the vector width, e.g., for OpenCL kernels [109, 138, 133, 42, 153], or choosing whether to vectorize a dimension [127].

For cache blocking, the dimensions of the block can be tuned. Register blocking requires parameter values for the block sizes as well, but has a different goal from cache blocking. By keeping data in the registers as much as possible the number of memory references can be reduced for improved performance.

Parameterized loop unrolling allows for the unrolling factor to be tuned or in a simplified version whether a loop should be unrolled fully or not at all. Multiple loops are perfectly-nested if the loop body of an outer loop only consists of the next inner loop. Perfectly-nested loops with independent iterations allow changing the order of the loops without affecting correctness. Interchanging loops can improve performance. For example, memory references can be made more efficient if loops can be arranged so that memory addresses are accessed linearly.

When designing algorithms, selecting an appropriate data layout is critical to performance. Compute kernels have been implemented with usually binary parameters that enable switching between (two) different types of data layout.

OpenCL offers vector types with different widths. Compute kernels have been written that allow for the vector width to be adjusted. Whether to vectorize at all can be a auto-tunable decision as well. The source-to-source compiler of the DSL Halide can tune whether to vectorize individual loops of a loop nest [127].

Less well-studied optimizations are: whether to parallelize a loop or a loop nest [7, 64], choosing the precision of floating-point computations [155, 117] and whether to generate prefetch instructions [159]. For GPU-based auto-tuner the use of the shared memory [165, 42] and the texture cache [42, 166] have been implemented as parameterized optimizations. While the mentioned optimizations apply to a wide range of applications, there are domain-specific optimizations as well. For example, the halo

size of stencils [98] or the permutation of matrix rows and columns for certain matrix operations [155].

The second type of parameter is exposed by runtime systems and programming languages. In OpenCL and CUDA, the dimensions of the index space (NDRange in OpenCL) of the compute kernel and the size of the work groups have been tuned in different studies [109, 90, 38, 125, 133, 95, 40, 117]. In CUDA terms these are referred to as grid and thread block, respectively. Similar runtime parameters are exposed by other languages as well. For example, Chen et al. tune parameters of the tasking system of the Chapel language [30]. The OpenACC standard specifies a parameterized tiling directive and a directive for mapping data to the shared memory [146]. This shows that optimization and runtime system parameters can overlap.

A third parameter type selects algorithms. Different algorithm might be optimal for different hardware platforms. And for a given hardware platform, different algorithms might be optimal depending on the input of a compute kernel. The frameworks Apollo and Insieme and the DSLs PetaBricks and Sequoia have been developed to manage different implementations of an algorithm and to select an appropriate implementation for a given input [52, 7, 14, 84].

A quite different objective of auto-tuning is the creation of optimal algorithms from building blocks. The goal is not only the selection of an optimal value, but the creation of a schedule that describes the final algorithm. Spiral creates algorithms for digital signal transforms where different factorization options for the matrices are explored via auto-tuning [126]. FFTW combines highly-optimized code fragments called codelets to generate FFT algorithms with different performance characteristics [57]. Steuwer et al. use rewrite-rules to replace expressions in a source-to-source compiler by equivalent expressions with potentially improved performance. Their rewrite-rules mainly target parallelization, vectorization and cache blocking [138]. The DSL Halide implements a similar approach for the optimization of image processing kernels [127].

Search spaces

The number of parameters and the value ranges of the individual parameters determine which search strategies are appropriate for the auto-tuning of an application. In general, auto-tuning has to deal with parameter spaces that are high-dimensional and contain a vast number of parameter-value tuples. Similar to the previous section, a distinction can be made between projects that do classical parameter tuning and approaches that create a schedule.

For optimization parameters, a search space with less than 10 dimensions is common, as can be seen in Tab. 2.1. Furthermore, parameterized optimizations often only involve choices between very few valid values. For example, whether a loop is to be parallelized is a binary choice. Similarly, a parameter that addresses the vectorization width might consider only powers of two between one and 16—for only five values. Runtime parameters and, even more so, algorithm-selection parameters tend to have similarly small value ranges.

Projects like Spiral, Halide and FFTW that create a schedule imply very different search spaces. These projects not only parameterize transformations, they select the type of transformation applied and the order in which transformations are applied as well. These additional degrees of freedom dramatically increase the size of the parameter spaces.

Few projects in the literature give a precise estimate of the size of the parameter space. To give an idea of parameter space size in past and current auto-tuning, we classified search spaces as large or small. As a criterion we used whether an exhaustive search seems to be feasible within a reasonable amount of time. That translates to whether the number of parameter combinations was less than a few thousands or larger. We are forced to use this rather vague criterion because of differences in the projects, their search spaces and how the search spaces are documented. As is shown in Tab. 2.1, this classification mostly mirrors whether the auto-tuning involves the creation of a schedule or not. However, there are a few outliers.

The OpenTuner project is a general auto-tuner that has been applied to widely different applications. This includes the tuning of hundreds of compiler flags and the creation of schedules for the DSL Halide. The parameter spaces involved have up to 10^{6328} possible configurations [6]. Ansel et al. state that OpenTuner was specifically designed for complex search spaces. OpenTuner seems to be unique in being able to deal with vast search spaces while being domain-independent.

Furthermore, there are three examples demonstrating that large parameter spaces in classical optimization and runtime tuning scenarios can occur. CLBlast's GEMM kernel has at least 14 parameters for millions of possible configurations [109]. Similarly, Datta et al.'s stencil code generator exposes 10 parameters for another large parameter space. And finally, Grauer-Gray et al. extend HMPP, a directive-based language for GPUs, so that it can apply five types of transformations to the loops of a kernel [64]. While the number of parameters for the individual experiments is not documented, as up to six loops are considered, it is plausible that the resulting parameter space is large.

Project/Authors	Type	Approach	Tuning Objective	Online	Search Strategies	Max. Par.	Search S.
FFTW [57]	FFT library	source-to-source	schedule	✗	dynamic prog.	high	large
CLBlast [109]	BLAS impl.	direct impl.	optimization/runtime	✗	random/exhaustive	≥14	large
ATLAS [157]	BLAS impl.	code generation	optimization	✗	exhaustive	≥4	small
ATLAS [158]	BLAS impl.	code generation	optimization	✗	exhaustive	≥ 13	small
OSKI [155]	SpMV library	direct impl.	optimization	online	N/A	N/A	N/A
Kamil et al. [87]	stencil DSL	source-to-source	optimization	✗	exhaustive	12	small
PATUS [31]	stencil DSL	source-to-source	optimization	✗	exh./genetic/...	likely low	N/A
Zhang et al. [166]	stencil DSL	source-to-source	optimization	✗	exhaustive	5	small
PARTANS [98]	stencil DSL	source-to-source	optimization	partially	exh./hill climb./binary	4	small
Halide [127]	stencil DSL	compiler	optimization/schedule	✗	genetic	high	large
Spiral [126]	DSP DSL	source-to-source	schedule	✗	gen./hill climb./...	high	large
Sequoia [52]	compute DSL	source-to-source	algorithm/optimization	✗	manual	N/A	N/A
PetaBricks [7]	compute DSL	source-to-source	algorithm/optimization	✗	n-ary/genetic	likely low	small
BOAST [153]	compute DSL	code transformation	optimization	✗	exhaustive	6	small
POET [163]	compute DSL	source-to-source	optimization	✗	simplex/random	5	small
OpenARC [134]	directives	source-to-source	optimization/runtime	✗	exhaustive	≥8	small
HMPP [64]	directives	code transformation	optimization	✗	random	>10	large
Active Harmony/CHiLL [149]	framework	tunable application dev.	optimization	✗	simplex	7	small
A. Harmony/CHiLL/Rose [150]	framework	tunable application dev.	optimization	✗	simplex	6	small
A. Harmony/CHiLL [148]	framework	tunable application dev.	optimization	online	simplex	6	small
Active Harmony [144]	framework	parameter tuning	optimization	✗	random/simplex	3	small
OpenTuner [6]	framework	parameter tuning	comp. flags/schedule	claimed	ensemble/genetic/...	hundreds	large
BEAST [8, 97]	framework	parameter tuning	optimization/runtime	✗	exhaustive	15	small
PTF [12, 104]	framework	parameter tuning	compiler flags	✗	individual	5	small
Apollo [14]	framework	algorithm selection	algorithm	partially	machine learning	2	small
Insieme [84]	framework	source-to-source	optimization	✗	genetic	3	small
CHiLL [29]	framework	source-to-source	optimization	✗	greedy w. backtrack.	≥7	small
SkePU [36]	framework	parallelism library	device selection/runtime	✗	genetic	5	small
Steuwer et al. [138]	study	source-to-source	optimization/schedule	✗	exhaustive	high	large
Datta et al. [38]	study	code generation	optimization/runtime	✗	exhaustive/random	>10	large
Price et al. [125]	study	code generation	optimization/runtime	✗	genetic	8	N/A
Komatsu et al. [90]	study	direct impl.	runtime	✗	exhaustive	1	small
Du et al. [42]	study	code generation	optimization	✗	exhaustive	10	small
Rul et al. [133]	study	code generation	optimization/runtime	✗	exhaustive	3	small
Li et al. [95]	study	code generation	optimization/runtime	✗	exhaustive	6	small
Samuel Williams [159]	study	code generation	optimization	✗	exhaustive	8	small
Dolbeau et al. [40]	study	direct impl.	runtime	✗	exhaustive	2	small
Pennycook et al. [117]	study	direct impl.	optimization/runtime	✗	exhaustive	2	small
Zhang et al. [165]	study	direct impl.	optimization	✗	exhaustive	3	small
Chen et al. [30]	study	parameter tuning	custom/runtime	✗	simplex	3	small
Tabatabaee et al. [140]	study	search strategy	application specific	✗	simplex variants	3	small
PrimeTile [68]	loop tiling tool	source-to-source	optimization	✗	exhaustive	3	small

Table 2.1.: A summary of the different approaches to auto-tuning and their most important properties.

Search strategies

The type of search strategies employed reflects the size of the search spaces. In projects with smaller search spaces, an exhaustive search strategy is used [133, 42, 40, 117, 98, 31, 134, 165, 38, 68, 18, 87, 95, 138, 159, 8, 165, 125, 90, 158, 157]. Though it might increase the duration of the auto-tuning, it guarantees the optimal result. This property makes an exhaustive search attractive where feasible.

Many projects that use exhaustive search combine it with constraints to prune the search space [38, 87, 138, 159, 8, 157, 165]. This of course accelerates the tuning process, as an exploration of the whole search space can be avoided. Additionally, pruning the search space can be necessary to ensure that only valid parameter combinations are evaluated. Tiwari et al. generalize the constraint-limited search approach by introducing a DSL to model constraints [148]. In addition to constraints, there are other approaches to pruning the search space. Hardware information can be used to rule out certain parameter values, thereby reducing the value set of individual parameters. Similarly, if highly different platforms are targeted, e.g., GPUs and processors, there are often parameters that only apply to a certain platform type (cf. Datta et al. [38] or Kamil et al [87]). Another strategy is the reduction of the dimensionality of the search space by tuning parameters in a specific order or in disjoint subsets. These two approaches have been employed to generate plausible initial parameter, before the whole search space is considered. By using these techniques, and noticing again that the value ranges of the parameters are often small, exhaustive search can be successfully applied to moderately-sized parameter spaces.

Other search strategies are needed for larger search spaces. A simple, yet popular search strategy is Monte Carlo search or random search [109, 64, 144]. While unlikely to find a global optimum in a vast parameter space, this search approach at least guarantees that the search does not get stuck in a local optimum. The Active Harmony group uses variants of Nelder-Mead simplex search [144, 31, 149]. They developed a parallel variant of the simplex search algorithm called parallel rank-ordering (PRO) [150, 148]. Another type of search strategy often used in the literature are genetic or evolutionary search strategies [7, 31, 127, 125]. Further strategies used are n-ary search [98, 7], hill climbing [98] and line search (Powell search) [31]. Individual search is employed by Bajrovic et al. and can be considered a variant of line search [12].

These search strategies do not have to be used on their own. CLBlast uses random search to generate a set of likely high-performance parameter combinations. For these candidates, a localized exhaustive search is performed [109]. OpenTuner uses a search

strategy they call ensemble search. Ensemble search is a meta-search approach where multiple search strategies run in parallel. Search strategies with promising results get more time allocated [6].

CLBlast is one of few applications that can avoid search entirely. This project maintains a database of parameter values for numerous hardware platforms. This database can be downloaded and, if results for the current hardware platform are available, tuned parameter values are available immediately [109]. Another approach for avoiding search is implemented by the auto-tuner Apollo. It uses machine learning techniques to train a model in an offline phase and then query the trained model at runtime [14].

Online vs. Offline Auto-Tuning

Auto-tuning approaches can be classified as online or offline. Online auto-tuning takes place during the runtime of the application, whereas offline auto-tuning implies that separate runs of the application are performed. Through online auto-tuning, performance can sometimes be improved beyond what is possible with offline approaches, as it enables adapting to input characteristics that are only known at runtime.

The principle requirement for online auto-tuning to be useful is the minimization of the overhead incurred on the application runtime. As for auto-tuning in general, its usefulness depends on whether the costs of tuning can be amortized by the improved performance enabled by auto-tuning. Furthermore, many auto-tuners are implemented as separate applications that execute the tuned application with varied parameter values. OpenTuner and the Periscope Tuning Framework both implement this approach [6, 104]. Other frameworks like BEAST [8, 97] might implement such an approach, because they use a different programming language for the auto-tuner and the tuned application, but the literature does not document it clearly. For online auto-tuning, it is advantageous to integrate the auto-tuner with the application. Thereby, the overhead introduced by restarting the application can be avoided. This overhead can be significant, for example if a checkpoint file or a large dataset needs to be read from disk.

A Tab. 2.1 shows, most auto-tuning approaches in the literature can be classified as offline approaches . Only four projects have at least some online auto-tuning capabilities.

Tiwari et al. showed that the auto-tuner Active Harmony can be combined with the code transformation framework CHiLL [29] for an online approach [148]. They designed a DSL for specifying constraints on parameters and implemented a JIT compilation approach to enable online search. The approach itself is language agnostic. However, it relies on two DSLs: the DSL to model the constraints and CHiLL's DSL for specifying

code transformations.

The sparse matrix-vector library OSKI has online auto-tuning capabilities [155]. OSKI adjusts parameters every time an OSKI function is called until the end of the tuning phase is reached. They argue that such an approach is necessary to adapt to the input which is only known at runtime.

Two of the auto-tuners investigated use a mixed approach that includes an online and an offline phase. As stated above, the auto-tuner Apollo uses machine learning to learn the performance characteristics of the search space. This happens in an offline phase. Then, at runtime, it uses the model to select a high-performance algorithm depending on input characteristics [14]. The stencil DSL PARTANS tunes the number and types of GPUs and how work is mapped to the GPUs in an offline phase. Then, during runtime, the halo size of the stencils is varied [98]. Thus, it splits the parameter space into parameters that are tuned offline and a parameter tuned online. OpenTuner is advertised as an online auto-tuning framework [6]. However, in their published literature they provide no evaluation of this feature.

3. Modern Hardware Platforms

A wide range of different hardware architectures have been used to build supercomputers and high-performance workstations. In the late nineties, x86 processors were adopted more and more widely. Today, x86 processors are the prevailing processor architecture in HPC, data centers and high-performance workstations. To be competitive, these processors have adopted multi-core architectures, added large vector units and generally saw multiple extensions of their instruction sets. However, especially in HPC and machine learning graphics cards and other accelerators have come more and more to the forefront.

In this chapter, we describe the most important technical aspects of modern processors and graphics cards. To that end, we first consider the global trends over the last two decades. Then, we focus on an Intel Skylake-SP processor as an example for a modern processor and Nvidia's P100 graphics card that represents a modern GPU. Both these devices and closely-related models are widely used in HPC. Finally, we summarize the hardware platforms that were used in the experiments conducted for this thesis.

3.1. Trends in Hardware

The performance of processors and similar computing architectures has improved over time. To more-deeply investigate how performance developed over the two last decades, we show trends for some performance-related metrics of compute devices in Fig. 3.1. To that end, we compiled publicly-available product information for a wide range of hardware architectures. Our database includes most processors of Intel's Xeon series, Intel's Xeon Phi series as well as AMD's Opteron and Epyc product lines. As GPUs have become more and more relevant for computationally- and memory-intensive tasks, we further added most GPUs of Nvidia's Tesla as well as AMD's FirePro and Vega series. Our dataset covers hardware released between 1998 and 2017. Especially for the older devices, documentation is sparse and minor errors are difficult to rule out. However, the overall trends are clear and consistent.

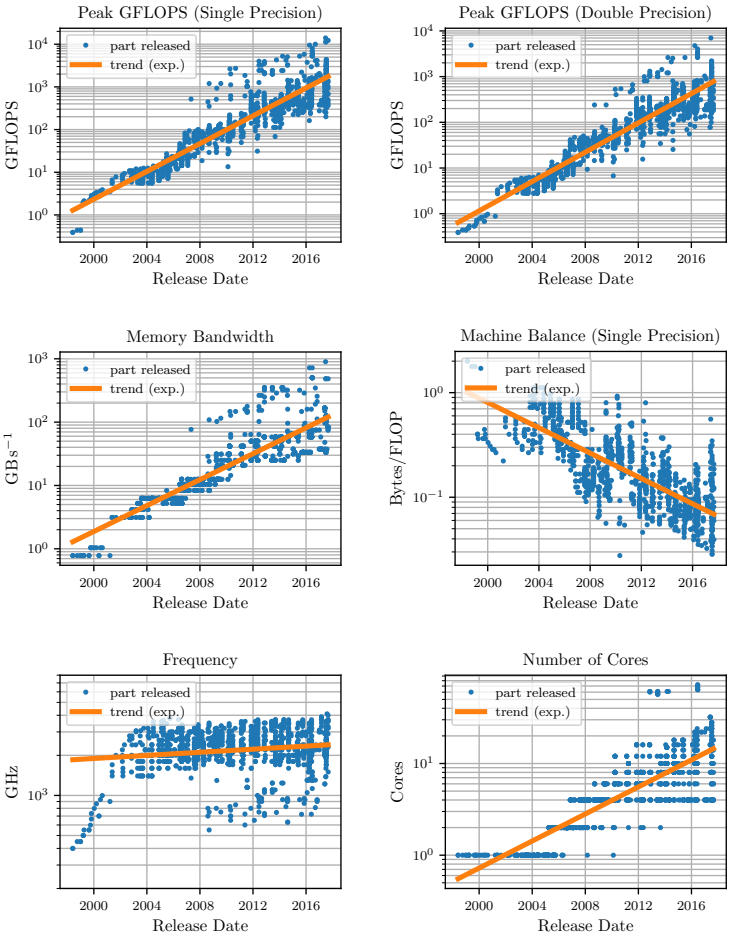


Figure 3.1.: Development of different hardware parameters over time (1998–2017). The data used encompasses Intel Xeon, Intel Xeon Phi, AMD Opteron and AMD Epyc processors. It further includes data on Nvidia Tesla, AMD FirePro and AMD Vega graphics cards. While some data points are missing, most processors and graphics processors of these series are included. There is some uncertainty about the older processors, as documentation of these products is no longer fully available. The data is based on public manufacturer product information.

A major focus of this work is performance of scientific computing. In this area, two important characteristics are peak floating-point performance and peak memory bandwidth. Both clearly improved exponentially over time. The highest-performance device in this survey is the Nvidia Tesla V100 GPU, which achieves a peak single-precision performance of 15 TFLOPS. For comparison, the highest-performance standard processor in this survey, the Intel Xeon Platinum 8180, achieves 4.1 TFLOPS. The more specialized Intel Xeon Phi processors, which were designed for HPC, achieve up to 6.9 TFLOPS. Generally, GPUs are approximately three to five years ahead in their floating-point performance compared to standard processors. Similarly, GPUs have offered significantly higher memory bandwidths since they have become available for general computations.

By dividing peak performance by the memory bandwidth, we can compute the machine balance of the devices. The machine balance is a useful metric for two reasons. First, if an algorithm does less floating-point operations per byte read from memory (writes could also be considered), peak performance is no longer achievable. Instead, performance is limited to a fraction of the peak performance. A commonly-used tool to estimate the achievable performance is the roofline model [160]. Second, it is often possible to rewrite algorithms so that they require less memory reads. A widely-used technique for this purpose is cache blocking. On machines with worse machine balance, such techniques are of greater importance. As Fig. 3.1 shows, over time machine balance got exponentially worse. A modern high-performance Xeon Platinum 8180 processor has a machine balance of 0.029 F/B. That is, an algorithm needs to perform 35 single-precision floating-point operations for every byte read from memory. A conclusion drawn from this observation is that cache-efficient algorithms are becoming ever more important. However, as cache-efficient algorithm do not exist for all relevant problems, it has generally become more difficult to achieve peak performance.

The development of processor clock speed and the number of cores show two important further trends. While clock frequency improved exponentially until about 2003, there have been only minor improvements since then. Though there are still data points that suggest small improvements in frequency, these higher frequencies are generally achieved by parts that employ fewer cores and overall offer less floating-point performance. Remarkably, GPUs achieved far lower frequencies than processors for a long time. However, their frequencies increased throughout the last years to slightly below 2 GHz, whereas the clocks of the highest-performance processors decreased to frequencies slightly above 2 GHz. For example, the Xeon Platinum 8180 has a rated turbo frequency of 2.3 GHz under heavy (AVX512) floating-point load. Its base frequency under the same condi-

tions is only 1.7 GHz. These frequencies are only relevant for programs that make heavy use of SIMD instructions. If a program only uses scalar instructions and only stresses a single core, this processor can increase the frequency of the utilized core to 3.8 GHz. Dynamic frequency adjustments according to load, power dissipation and temperature introduce further challenges for writing efficient software, modeling the performance of algorithm and accurately measuring performance.

While performance increased, the clocks roughly remained the same. This, of course, requires an explanation. In the last paragraph, we alluded to one major factor: the number of cores increased from a single core to up to 28 cores on processor platforms. The more specialized many-core Xeon Phis implement up to 72 cores. Architecturally, GPUs can be considered many-core processors as well, though they are programmed through different programming models. For instance, the Tesla V100 GPU consists of 80 cores called shader multiprocessors [111].

The implementation of increasingly wider vector units is a second major factor that explains why performance increased. For discussing the development of SIMD capabilities, we focus on a single processor core. Earlier processors in this survey had no SIMD capabilities and achieved 1 F/cycle. Intel's SSE extensions introduced 2-wide vectors, though earlier implementations did not implement the full vector size in hardware. The Intel Core 2 processor already achieved 8 FLOPS/cycle doing single-precision floating-point calculations. In 2011, Intel introduced AVX which improved performance to 16 FLOPS/cycle. Vector performance was again doubled to 32 FLOPS/cycle in 2013 when FMA instructions where added. Note that FMA instructions are commonly counted as two floating-point operations with the multiplication and the addition being counted separately. Vector capabilities were again improved to 64 FLOPS/cycle with the introduction of Skylake-SP in 2017.

GPU shader multiprocessors implement similar SIMD capabilities, though their vector units are even wider. For example, Nvidia GPUs have two 32-wide vector units per shader multiprocessor. When FMA operations are scheduled, this translates to 128 FLOPS/cycle. Compared to a decade ago, processors have nearly caught up to GPUs with respect to their SIMD capabilities. However, as processors strongly reduce their frequencies when fully using their vector units and as they still implement fewer SIMD-enabled cores, there is still a major difference in performance.

Both an increased number of cores and improved SIMD capabilities pose further challenges to developers of algorithms for high-performance hardware. To achieve a significant fraction of the peak performance of a device, it has become critical to develop

parallel and vectorized algorithms. The increasing number of cores has introduced the node-level scalability of an algorithm as a serious issue. This is especially the case for heterogeneous nodes that combine one or two processors with often multiple GPUs. Furthermore, SIMD-level parallelization is more limited than thread-level parallelization. Threads can execute different instruction stream. Using masking techniques, individual components of a SIMD vector can execute divergent code paths as well. However, masking generally leads to serial execution of the different code paths and therefore a reduction in performance. GPUs implement a form of automatic masking that allows developers to write scalar codes that are guaranteed to be vectorized. However, if masking is employed performance is still affected, a phenomenon known as warp divergence on Nvidia GPUs. On x86 processors, the AVX512 instruction set added hardware support for masking so that programming languages and frameworks can be designed that simplify the development of efficiently-vectorized code even for very wide vector units.

3.2. The Intel Skylake Architecture

The Intel Xeon Platinum 8180 is a 28-core x86 processor. It is one of the highest-performance models of the Skylake-SP series and uses the Skylake microarchitecture. This processor has 6 memory controllers that can support DDR4 clocked at up to 2666MHz, leading to a theoretical maximum memory bandwidth of 125GB/s. The cores are interconnected by a mesh, compared to one or multiple ring buses that were used in prior generations of Intel processors. This approach incurs a somewhat higher latency if further away cores are accessed compared to the ring approach. Next to each core resides a slice of the L3 cache with a size of 1.375MB [93].

On the core level, the Intel Skylake-SP processor has a L1 data cache with a size of 32kB, the L1 code cache has the same size. The L1 data cache can be accessed with a latency of 4 cycles. The size of the L2 cache was increased to 1MB, which is a major change, as the L2 cache had a size of 256kB since the introduction of the Nehalem microarchitecture in 2008. With Skylake-SP, Intel introduced a new set of vector instructions called AVX512. AVX512 extends the vector registers and vector instructions from 256 bit to 512 bit. This translated to a doubling of the vector performance of the high-end models compared to the Broadwell generation. However, the lower-end Skylake-SP models only have a single FMA unit. Because of this and because prior generations offered two FMA units, some Skylake-SP processors only achieve the same peak floating-point performance as their AVX2-implementing predecessors. As is common for

x86 processors, the prior vector instruction sets MMX, SSE, AVX and AVX2 are still supported, though for optimal performance AVX512 has to be used (except for models with one FMA unit) [93].

To achieve maximum floating-point instruction throughput on a Skylake-SP processor with two FMA units, both FMA units needs to kept busy by filling the two floating-point pipelines with a new floating-point instruction every cycle. At an AVX512 boost frequency of 2.3 GHz, the Xeon Platinum 8180 achieves a peak single-precision peak performance of 4.2 TFLOPS – and 2.1 TFLOPS for double precision.

On a Skylake-SP processor the instruction latency for addition, multiplication and FMA is 4 cycles. Assuming a model with two FMA units such as the Platinum 8180, 8 independent floating-point operations need to be in-flight for maximum throughput. Because of the number of in-flight micro-operations required, any code that attempts to achieve a large fraction of the peak performance needs to provide a high degree of instruction-level parallelism that can be leveraged by the hardware. While most codes exhibit some instruction-level parallelism, a full saturation of the floating-point vector pipelines usually needs significant optimization effort.

A processor implementing this architecture can load 2x64B per cycle, two cache lines, from the L1 cache and store up to 64B. This enables a streaming of one or two instruction operands from the L1 cache alleviating the pressure on the register file, which has 32 512 bit registers [54, 93].

As described in Sec. 3.1, cache blocking is a critical optimization due to the machine balance of modern hardware platforms. However, the memory poses further challenges. For example, outside the register level, data movement happens at cache line granularity. A Skylake-SP processor has a cache line size of 64 B. An implication of doing memory transactions at the size of cache lines is that scalar accesses that are not accompanied by further scalar accesses to the same cache lines are very expensive. To access a single 8 B variable, 64 B have to be moved, in this case up to 56 B or $\frac{7}{8} \approx 88\%$ of the bandwidth can be wasted. Therefore, for optimal throughput the data of accessed cache lines should be fully utilized by the program. This also explains why gather-scatter patterns on modern processors and GPUs are often very expensive[1].

A related issue is the necessity of efficient prefetching. As the memory latency on a modern hardware platform can be hundreds of cycles, a large fraction of pipeline slots might remain empty if memory is not properly prefetched. This precludes any memory efficiency for algorithms that randomly access memory, as such accesses cannot

[1] With the exception of accessing only data from few memory streams.

be prefetched by definition. To combine cache-line utilization with efficient prefetching, recommended access patterns are unit-stride accesses, which generally are optimal, or strided accesses with fixed, predictable strides.

The hardware features discussed are often performance-critical for HPC applications. Of course, this brief outline is in no way complete. Unfortunately, whether a feature is critical for high performance strongly depends on the application. As modern hardware is complex, we cannot exhaustively cover the many aspects of modern hardware platforms that can become relevant.

3.3. The Nvidia Pascal Architecture

At the time of writing, the Nvidia Tesla P100 GPU was one of Nvidia's strongest GPUs for scientific computing. Only the slightly faster and otherwise similar Tesla V100 offered higher performance for scientific workloads that make use of double-precision arithmetic. The architectural details described in the following make use of documentation made available by Nvidia [110, 111].

The Tesla P100 implements Nvidia's Pascal architecture and became available in 2016. This GPU consists of 56 shader multiprocessors. As mentioned, each shader multiprocessor roughly correspond to a core of a standard processor. It can execute instructions independently of other shader multiprocessors and consists of execution resources such as ALUs, a register file and caches.

The P100 GPU uses eight HBM2 memory controllers to access four HBM2 stacks with 4 GB each for a total of 16 GB of memory. In total, the eight memory controllers have a 4096 bit-wide access path to the HBM2 memory. Due to the very wide memory interface, the peak memory bandwidth of the device is 732 GB/s.

Each shader multiprocessor of a Pascal GPU has 64 so-called CUDA cores. While they process scalar threads from a programmer's point of view thanks to the CUDA programming model, the hardware actually executes 32-wide floating-point vector instructions. This explains why a shader multiprocessor can be viewed as a SIMD core with two 32-wide floating-point vector units. A shader multiprocessor additionally has 8 load-store units to transfer data from and to the register file and 8 special function units that are used to calculate less common instructions such as transcendental functions.

GPUs are often specified by their overall number of CUDA cores, computed by multiplying the number of shader multiprocessors with the SIMD-width per shader multiprocessors. Counted with this approach, the P100 GPU employs 3584 CUDA cores. At

a frequency of 1.3 GHz and one FMA operation performed every cycle by each CUDA core, this results in a peak single-precision performance of 9.5 TFLOPS. As the P100 can do double-precision arithmetic at half the rate of single-precision instructions, its peak double-precision performance is 4.8 TFLOPS.

The P100 GPU groups 32 threads into a warp which resides on a single shader multiprocessor. Instructions are executed on the warp-level with a paradigm that is called Single-Instruction-Multiple-Threads (SIMT) by Nvidia. The SIMT paradigm extends the SIMD approach by permitting every vector element to behave as if it had an individual program counter. This greatly simplifies vectorization, as it makes it possible for an individual thread (i.e. vector element) to execute an individual instruction, as if it were an actual scalar thread. The hardware implements this so that threads execute instructions jointly by default. If a branch instruction is encountered and different threads take different paths, the execution of the branches is serialized. The threads of the diverged warp now make progress at a slower rate, as masking is used to apply the now diverged instruction stream only to the appropriate elements. An important benefit of this approach is that a program is parallelized and vectorized at the same time. Thus, vectorization as an additional implementation step is not required on the P100 and other GPUs.

A major difference between CPU and GPU designs is the register file. While processors generally only expose few registers in their ISA, each shader multiprocessor on the P100 has 65536 32 bit registers. For the representation of double-precision values, two registers are used. Generally, GPUs implement smaller caches compared to competing processors. The P100 only has a 24 kB L1 cache, and the L2 cache, which acts as the last-level cache for the whole chip, only has a size of 4 MB. However, there is an additional 64 kB shared memory attached to each shader multiprocessor. The shared memory acts as a user-controlled cache and is generally used to share data between groups of threads residing on a shader multiprocessor. Such a group of cooperating threads is called a thread block in CUDA terminology.

Despite the higher memory bandwidth compared to processors, algorithms with excellent data locality are critical, as peak performance requires 13 floating-point operations performed for every byte read from or written to memory. Apart from reusing data stored in the per-thread registers, the shared memory is critical for improving data locality. We remark that modern GPUs use DRAM as main memory that is similar, in some cases even identical, to that used by processors. All considerations related to cache-line utilization and efficient memory access patterns, as described in Sec. 3.2, apply to

GPUs as well.

GPUs execute many threads simultaneously. The Tesla P100 supports up to 2048 threads per shader multiprocessor and therefore up to 114688 threads overall. On a shader multiprocessor, the high number of threads relative to the 64 CUDA cores are used to hide memory and instruction latencies. This approach is made possible by zero-latency switching between warps, which in turn is enabled by statically allocating the resources of threads such as the register space and part of the shared memory. This approach strongly contrasts to CPUs, where low instruction latencies and low-latency caches are used to address the same issues and context switches between threads (beyond SMT) are expensive.

The number of threads needed on a shader multiprocessor to hide all relevant latencies varies with the algorithm used. Generally, it is disadvantageous to use more threads than needed, as this limits the number of registers available per thread. Apart from using threading to hide latency, instruction-level parallelism can also be used [154]. The P100 warp scheduler can schedule up to 2 instructions to a warp in a single clock cycle.

Work on GPUs is scheduled in form of thread blocks which have a configurable size. As the work is internally executed by warps, the thread block size should be a multiple of the warp size. All warps of a thread block reside on the same shader multiprocessor. Within a thread block, synchronization is possible as well as sharing data through the shared memory. Synchronization between thread blocks is possible through a feature called cooperative groups, but requires additional effort by the developer. Compared to processors, the limited synchronization capabilities can be significant challenge for adapting codes for GPUs.

3.4. Hardware Platforms Overview

In this work, we use 11 different devices: five GPUs and six processors. The characteristics of these devices are summarized in Tab. 3.1. These devices implement a wide range of hardware architectures and therefore are a challenging target for studying performance portability. Most devices in our selection were considered high-performance devices when they were released. The sole exception is the AMD A10-7850K, which is a low-cost desktop processor. As this selection spans multiple generations of compute hardware, there are of course major differences in performance.

The GPU devices encompass two different hardware architectures. Both AMD devices are based on the GCN architecture, though the Vega VII uses a modernized variant of

Device	Cores/ Shaders	Frequency	Peak (SP)	Mem. Bandw.	Machine Balance	DP ratio
Graphics Processors						
Nvidia Tesla P100	3584	1.3 GHz, boost	9.5 TF	732 GB/s	13.0 F/B	1/2
Nvidia Quadro GP100	3584	1.5 GHz, boost	11.2 TF	732 GB/s	15.2 F/B	1/2
AMD Vega VII	3840	1.75 GHz, boost	13.4 TF	1024 GB/s	13.1 F/B	1/4
Nvidia GTX 1080 Ti	3584	1.8 GHz, boost	12.9 TF	484 GB/s	27 F/B	1/32
AMD FirePro W8100	2560	0.8 GHz, max	4.2 TF	320 GB/s	13.2 F/B	1/2
Processors						
AMD Epyc 7551P	32	2.5 GHz, turbo	1.3 TF	170 GB/s	7.5 F/B	1/2
AMD A10-7850K	4	3.7 GHz	0.1 TF	34 GB/s	3.5 F/B	1/2
Intel 2xXeon E5-2670	16	3.0 GHz, AVX turbo	0.8 TF	102 GB/s	7.5 F/B	1/2
Intel 2xXeon E5-2680v3	24	2.8 GHz, AVX turbo	2.2 TF	137 GB/s	15.7 F/B	1/2
Intel 2xXeon Gold 5120	28	1.6 GHz, AVX512 turbo 2.2 GHz, AVX2 turbo	1.4 TF 2.0 TF	230 GB/s	8.6 F/B 6.2 F/B	1/2
Intel Xeon Phi 7210	64	1.5 GHz, AVX512 turbo	6.1 TF	$\geq$ 400 GB/s	15 F/B	1/2

Table 3.1.: A summary of the performance characteristic of all node-level hardware platforms used in this work. Note that all platforms vary their frequency according power, temperature and workload. Floating-point performance is given in single-precision TFLOPS.

it. The FirePro W8100 is five years older than the Vega VII, which explains its lower performance. All three Nvidia GPUs implement the Pascal architecture. Apart from a minor difference in frequency, the Tesla P100 and the Quadro GP100 are identical. A P100 is built into each node of Piz Daint, one of the supercomputers used for this work. As a consumer-grade Pascal GPU, the GTX 1080 Ti has a lower memory bandwidth and double-precision arithmetic is practically disabled. Still, it offers higher single-precision floating-point performance compared to the P100. We investigate multi-GPU performance on a node with eight GTX 1080 Ti GPUs.

The AMD Epyc 7551P processor implements the Zen architecture which is a major redesign compared to its Bulldozer-derived predecessors. We included one Bulldozer derivate in our survey in form of the A10-7850K.

Each Intel device implements as different microarchitecture. The Xeon E5-2670 is part of Intel's Sandy Bridge generation, whereas the Xeon E5-2680v3 is a Haswell part. We provide results for another supercomputer, Hazel Hen, which uses the E5-2680v3 processor in its nodes. The Xeon Gold 5120 is part of Intel's more modern Skylake-SP generation. All three Xeon processors, but not the Xeon Phi, are used in dual-socket configurations in all experiments.

Finally, the Xeon Phi 7210 is a many-core processor that implements the Knights Landing architecture. While the three Xeon processors employ related architectures, the Xeon Phi has an entirely different foundation. Its cores are based on Intel's low-power low-performance Atom line-up and use an extended variant of the Silvermont architecture. The most important difference compared to these low-performance parts is the addition of two 512 bit vector units per core [82]. However, if instructions other than vectorized floating-point instructions are executed, the (scalar) performance of the Xeon Phi processor is much lower compared to all other processors in this collection. Because of this and because it has the highest number of cores, this architecture is particularly difficult to optimize for.

Part II.

AutoTuneTMP

In the following two chapters, we introduce the auto-tuning framework AutoTuneTMP and the first auto-tuned application. In Chapter 4, we present the basic optimization loop of AutoTuneTMP and two types of auto-tuned compute kernels: the CPPJIT and the generalized kernel type. For the introduction of AutoTuneTMP, we consider only minimal examples, as these are most helpful for illustrating the API.

To demonstrate the applicability and usefulness of AutoTuneTMP in general and the CPPJIT kernel type in particular, we consider matrix multiplication as a challenging application in Chapter 5. Due to the large, higher-dimensional search space, we further use this application to compare the different search strategies provided by AutoTuneTMP.

Two additional auto-tuned kernels that make use of the CPPJIT kernel type are presented in the appendix. In Sec. A.2, we describe and evaluate an auto-tuned variant of the `triad` routine which is part of the STREAM benchmark [101]. Furthermore, we present the sparse grid streaming algorithm in Sec. 7.1. An auto-tuned variant of this algorithm that uses the CPPJIT kernel type can be found in Sec. B.4.

Published Work

The following chapters extend the paper "AutoTuneTMP: Auto-Tuning in C++ With Runtime Template Metaprogramming" written by the author of this thesis [118].

4. AutoTuneTMP: Leveraging C++ for Performance and Productivity

AutoTuneTMP is an auto-tuning framework written in C++ as a header-only library. The foremost goal of AutoTuneTMP is the auto-tuning of compute kernels in order to maximize performance on a given hardware platform and for given input data. It is a general auto-tuning framework in that it is not tied to a specific application or domain.

Like many of the approaches presented in Sec. 2.2, AutoTuneTMP uses parameter tuning where a set of parameter values represents a variant of a compute kernel. A major difference to most other approaches is the use of JIT compilation to create compute kernel variants at runtimes. JIT compilation is key for reducing the time required for tuning in offline settings and for enabling online auto-tuning. Our framework improves on the state-of-the-art by introducing an approach that requires little changes to the application to be tuned and that can also be retrofitted to existing applications with only local changes. This is enabled by the use of C++ template metaprogramming, hence the name of the framework.

The architecture of AutoTuneTMP is shown in Fig. 4.1. The search strategies and the parameters together enable general parameter tuning. Search strategies are implemented against sets of parameters with shared properties. Parameter sets store arbitrarily-typed parameters compatible with the shared parameter interface of a parameter set. A search strategy explores the search space spanned by a parameter set and returns the highest-performance parameter value combination it encountered.

AutoTuneTMP currently supports two different kernel types, which are different approaches for integrating auto-tuning into an application. The CPPJIT kernel type uses JIT-compiled C++ and was designed to work with a collection of optimization templates. The optimization template collection facilitates the development of auto-tunable compute kernels, as parameters that are exposed by the templates can be tuned by the parameter tuners. We call the second type of compute kernel the generalized compute kernel type. For this kernel type, AutoTuneTMP acts as a parameter tuner and

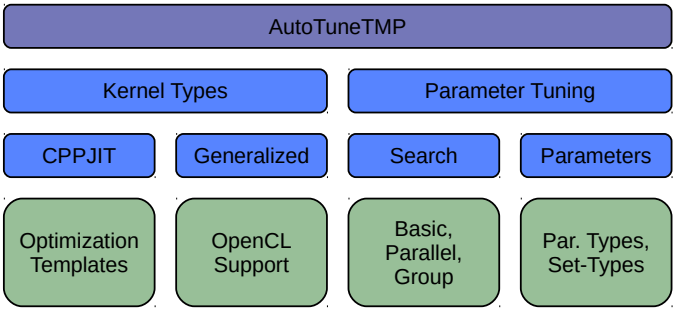


Figure 4.1.: The architecture of AutoTuneTMP. AutoTuneTMP implements two types of compute kernels and the functionality needed for the parameter tuning of these compute kernel types.

the details of the kernel need to be handled through a different framework. We use the generalized kernel type to auto-tune OpenCL kernels. For auto-tuning OpenCL kernels, AutoTuneTMP provides classes and functions that support the integration of AutoTuneTMP.

In the following, we first introduce the optimization loop that AutoTuneTMP implements and justify our runtime code generation approach in more detail. Then, we present the CPPJIT kernel type and a first example for applying AutoTuneTMP to a matrix-vector multiplication compute kernel. Afterwards, we describe the generalized kernel type and our approach for supporting OpenCL compute kernels. Having described both kernel types, we look closer at the search strategies and parameter types. Finally, we describe how we assess the contribution of auto-tuning to the performance of a compute kernel by defining a challenging baseline.

4.1. The AutoTuneTMP Optimization Process

AutoTuneTMP implements an optimization approach that requires four input parameters for a three-step optimization process. This process is displayed in Fig. 4.2 and constitutes a loop between three components: the tuner, the JIT compiler and the evaluation component. The inputs to the optimization process are the compute kernel, the description of the parameters, a search strategy and input data for the compute kernel. As the object of tuning, the compute kernel is a parameterized JIT-compiled function. The search space is spanned implicitly by the description of the parameters,

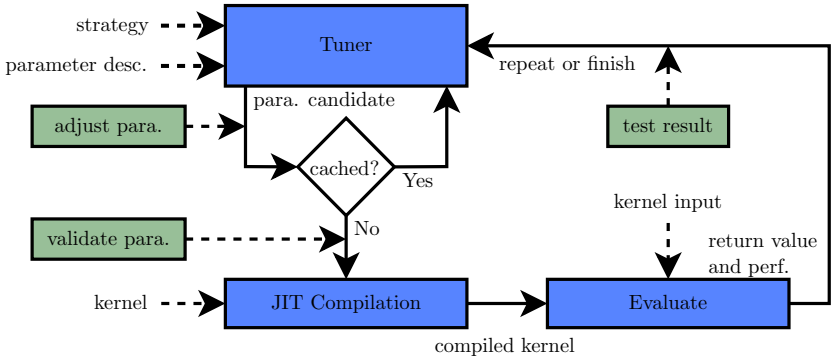


Figure 4.2.: The approach to auto-tuning that AutoTuneTMP implements. Central to the approach is the optimization loop that is controlled by the tuner component. After JIT compilation a parameter value combination gets evaluated. A cache prevents the tuner from repeatedly evaluating the same parameter combinations, as evaluations can be costly. Optional functors can be registered to more finely control the auto-tuning process (green boxes).

for which AutoTuneTMP provides parameter types. The search strategies, or tuners, of our framework enable an exploration of the search space. As AutoTuneTMP was designed for extensibility, the use of custom search strategies and parameter types is possible as well.

Whenever an auto-tuned compute kernel gets called, it is used just like a standard C++ function. Therefore, unless it has an empty argument list, input data is required to call the kernel. As the compute kernel gets called repeatedly during tuning, some important restrictions apply. Kernels that accept arguments by-reference should not modify the input data. Alternatively, if the kernel implements pass-by-value semantics, it can modify the input data, but the input data needs to be copyable. Compute kernels should generally avoid side effects.

As the next step, we describe an iteration of the optimization loop, starting at the tuner component. The configured search strategy that the tuner component manages chooses the next set of parameter values to consider. These parameter values are then applied to the compute kernel and the compute kernel gets JIT-compiled. Then, the compute kernel gets called with the supplied kernel input data and the performance of the current kernel variant is measured. Based on the internal state of the search strategy,

it either selects the next set of parameter values or finishes the tuning process returning the best parameter combination encountered during tuning.

Some search strategies might attempt to evaluate the same set of parameter values more than once. As a kernel call can be expensive, a cache component stores the measured performance of prior evaluations. These prior evaluation results are returned in case of a cache hit. Thereby, the implementation of search strategies is simplified, as duplicate evaluations are cheap.

AutoTuneTMP allows for functors to be registered that can be used to further customize the auto-tuning process, these functors are optional (green boxes in Fig. 4.2). Generally, the search space is obtained by computing the cross product of the value ranges of the parameters. Often, parameter value combinations imply kernels that would return incorrect results or not even compile successfully. However, it can be challenging to span parameters spaces that only contain valid parameter value combinations. For these cases, it can be convenient to adjust the parameter values before evaluation, so that they are as close as possible to the original parameter values and valid. By setting up an *adjust-parameters* functor, this adjustment can be implemented. How the best-fitting valid parameter values for given invalid parameter values are to be chosen, of course, depends on the application. We use this approach in most of the auto-tuned compute kernels throughout this thesis, e.g., in Sec. 7.2.

In other cases, it is more convenient to span a parameter value space with invalid parameter value sets, but skip invalid parameter combinations whenever they are encountered. The *validate-parameters* functor receives the candidate parameter values as its input and returns a Boolean that indicates whether the candidate values should be skipped. If the parameter values are discarded, control is handed back to the tuner component. This approach was chosen for the auto-tuned unified streaming compute kernel that is presented in Sec. 7.1.1. It has two parameters for which a lesser-than relation gets tested through the *validate-parameters* functor.

The search space of a parameterized compute kernel can be vast. Because of this, it is difficult to guarantee that all compute kernels with valid parameter values do actually compute a correct result. The issue of guaranteeing correctness can be addressed, or at least mitigated, by specifying a *test-result* functor. This functor gets called with the result of a compute kernel call after the compute kernel was evaluated and can test whether the computed result is in fact correct. Thereby, erroneous compute kernel variants can be detected. An error in this functor is not necessarily critical and the search strategy can continue the tuning process. For the purpose of this functor, the

result of a compute kernel is defined as the return value of the kernel. Like the two other functors, we use the *test-result* functor throughout this thesis.

4.2. JIT Compilation for Runtime Code Generation

Though we have mentioned that we use JIT compilation, so far we have not sufficiently justified why JIT compilation is a central component of AutoTuneTMP. We use a JIT compilation approach, because it is an effective means for runtime code generation and it seems to be the only common runtime code generation approach. We provide three reasons for employing JIT compilation and, more generally, runtime code generation.

Two advantages of JIT compilation are advantages relative to a common approach for auto-tuning. Some auto-tuning frameworks are implemented as external applications and are not integrated with the application that is tuned, e.g., OpenTuner [6]. These frameworks tune an application by repeated recompilation or at least application restarts. However, the compilation of large applications can be expensive. Even if the build system only rebuilds parts of the application that are affected by the changed parameterization, it can still be assumed that the compilation is significantly more expensive than locally recompiling a single compute kernel. Furthermore, as the whole application gets restarted, all time not spend in the compute kernel to tune can be considered overhead. This overhead can be significant. For example, if an application loads a dataset at start-up or if other compute kernels need to be executed before the tuned compute kernels is reached. As a runtime code generation approach does not need application restarts and only compiles compute kernel variants that get evaluated, faster auto-tuning can be expected.

A further issue is the integration of auto-tuning into the application. External auto-tuners are, by design, not tightly integrated with the compute kernel, as the compute kernel and the external tuner are two separate software components. With runtime code generation, we can more tightly integrate the auto-tuning of the compute kernel with the compute kernel. That is, the description of the parameter space, the configuration of the search strategies, the configuration of the JIT compilation and the compute kernel itself are all integrated. Thereby, we provide a straightforward path for adding auto-tuning to applications, as only local changes to a compute kernel are required. Furthermore, as the auto-tuned compute kernel can be considered a single (C++) component, maintainability is improved.

Finally, runtime code generation is required for online auto-tuning of compute kernels.

As online auto-tuning implies that auto-tuning occurs during application runtime, the application cannot be restarted or recompiled. Therefore, there are two options: either many compute kernel variants have been precompiled into the application at compile time or compute kernels variants are generated at runtime. A precompilation approach is only feasible for tens to possibly hundreds of compute kernel variants, but not for vast parameter spaces due to excessive time required for compilation and the size of the resulting binaries. For tiny parameter spaces, this approach has been explored by Beckingsale et al [14]. In contrast, runtime code generation only needs to compile kernel variants that are evaluated and the resulting size of the program is generally not an issue.

4.3. The CPPJIT Kernel Type

In the following subsections, we first describe the C++ JIT compilation framework CPPJIT. Then, we provide a first auto-tuning example where we tune a single parameter of a matrix-vector multiplication implementation to illustrate the API. As presented, the API of search strategies, parameters set types and parameters applies to the generalized kernel type as well.

4.3.1. CPPJIT: A JIT Compilation Library for C++

CPPJIT is AutoTuneTMP's component for JIT-compiling C++ compute kernels. It is not a compiler, instead it instruments standard C++ compilers such as GCC or Clang. From an implementation perspective, CPPJIT is a header-only library. Before we describe CPPJIT's internal workings, we first briefly introduce how it is used.

Listing 4.1 shows how a CPPJIT kernel is declared and called. In the global scope of the application, the macro `AUTOTUNE_KERNEL` is used to declare the JIT-enabled compute kernel. Its parameters are the signature of the kernel using C++'s standard function type syntax, the name of the kernel function to be declared and a string that specifies the location of the compute kernel source files. As a result of the macro, the namespace `autotune` gets populated with an object named according to the supplied symbol name of the kernel. The type of the object is an instantiation of an (internal) template for managing CPPJIT kernels. This wrapper object represents the kernel and has a call operator `operator()` through which the compute kernel can be called. When the kernel is called for the first time, the JIT compilation is triggered. Consecutive calls then reuse

Listing 4.1: Declaration and compilation of a JIT-compiled kernel.

```
/* global scope */
AUTOTUNE_KERNEL(vector<double>(vector<double> &), square,
  "kernel_src_dir")
/* local scope */
// JIT compile and run with given vector o
vector<double> r = autotune::square(o);
```

Listing 4.2: The implementation of a JIT-compiled kernel. JIT compilation requires the macro `AUTOTUNE_EXPORT` before the return type. The kernel language is standard C++.

```
#include "autotune_kernel.hpp"

AUTOTUNE_EXPORT vector<double> square(vector<double> &o) {
  vector<double> r(o.size());
  for (size_t i = 0 ;i < o.size(); i++)
    r[i] = o[i] * o[i];
  return r;
}
```

the result of the first JIT compilation (unless parameters of the kernel have changed). The signature of the call operator is that of the compute kernel as submitted to the `AUTOTUNE_KERNEL` macro. Thus, the wrapper object behaves as if it were a standard C++ function. We argue that this contributes to the usability of CPPJIT.

Having shown how a compute kernel can be declared and called, we illustrate the implementation of a compute kernel next. Listing 4.2 shows a compute kernel compatible with the example from List. 4.1. To mark a function for JIT compilation, the `AUTOTUNE_EXPORT` macro needs to be written before the return-type of the kernel. Furthermore, the name of the file the kernel is stored in needs to be the same as the kernel, i.e., in this case `square.cpp`. The macro `AUTOTUNE_EXPORT` is defined in the header `autotune_kernel.hpp`.

Internally, CPPJIT uses a standard C++ compiler and the functions `dlopen`, `dlsym` and `dlclose`, as defined by the POSIX standard [145]. With these functions, shared libraries can be loaded into an application, symbols can be located and, of course, the shared library can be closed after use. CPPJIT first uses the configured C++ compiler to compile and link the compute kernel into a shared library. It then locates the symbol of the compiled compute kernel using the POSIX API. Lastly, it sets up a function

Listing 4.3: Customizing the JIT compilation of a CPPJIT kernel by changing the compilations flags of the compiler and linker.

```
using cppjit::builder::gcc;
autotune::square.get_builder<gcc>().set_cpp_flags(
  "-Wall -Wextra -fopenmp -std=c++17 -O3 -march=native"
  " -mtune=native -ffast-math");
autotune::square.get_builder<gcc>().set_link_flags("-fopenmp");
```

pointer, so that the loaded symbol gets called whenever the call operator of the wrapper object is called. The use of `dlopen/dlsym/dlclose` also explains why the macro `AUTOTUNE_EXPORT` is needed. As the POSIX API used is a C API, a kernel requires C (and not C++) linkage for our approach to work. The `AUTOTUNE_EXPORT` macro ensures that the kernel has C linkage. By not using `export "C"` directly the internal implementation of CPPJIT can be changed without necessarily requiring changes to the kernels.

By default, during JIT compilation the compute kernel is compiled using GCC and the flags "-fPIC -fno-gnu-unique". The resulting object file is then linked with the flags "-shared -fno-gnu-unique". While "-fPIC" and "-shared" are standard flags for compiling and linking shared libraries, at the time of writing the "-fno-gnu-unique" flag was critical on Linux systems. This flag needs to be set to ensure that the symbol can be unloaded properly when the kernel is recompiled during auto-tuning, otherwise `dlclose` calls get ignored and the kernel is not properly replaced by a different kernel variant. The above compile flags are obviously not sufficient to generate fast code, at least some optimization flags are usually needed. CPPJIT provides an API for extending the compile and link flags that is shown in List. 4.3. The required flags mentioned above are always implicitly added. The GCC builder class provides similar functions for customizing the include path, the library paths and for specifying libraries required at link time.

Though we exclusively use the GCC compiler with this kernel type, the compiler can be exchanged with any compiler that is application binary interface (ABI) compatible with the application (and POSIX-compatible). Furthermore, for building more complex kernels CPPJIT can invoke a build system such as CMake.

4.3.2. Auto-Tuning a Matrix-Vector Multiplication Kernel

Having described how CPPJIT can be used to implement JIT-compiled kernels, we now describe our approach for auto-tuning CPPJIT compute kernels by the example of

Listing 4.4: Auto-tuning of a matrix-vector multiplication compute kernel with a single parameter for cache blocking.

```
1 AUTOTUNE_KERNEL(vector<double>(vector<double> &, vector<double> &),
2                 mv_kernel, "kernel_src_dir")
3 /* within function, the declaration of m and v is not shown */
4 enumerated_set parameters;
5 /* delegates to the parameter constructor
6    log_parameter: par. name, initial value, base, min, max */
7 parameters.emplace<log_parameter>("BLOCKING", 1, 2, 1, 16);
8 // create tuner and tune parameters
9 tuners::bruteforce tuner(autotune::mv_kernel, parameters);
10 enumerated_set optimal = tuner.tune(m, v);
11 // use optimal parameters
12 mv_kernel.set_parameter_values(optimal);
13 vector<double> result = mv_kernel(m, v);
```

a matrix-vector multiplication compute kernel. This compute kernel exposes a single parameter `BLOCKING` that, as the name suggests, controls cache blocking.

In List. 4.4, we provide a minimal example for auto-tuning this kernel. As we use CPPJIT for JIT compilation, the kernel `mv_kernel` is declared using the `AUTOTUNE_KERNEL` macro. In Line 4, a parameter set of the `enumerated_set` type is declared. It stores parameters that implement a certain interface and, through the properties of the stored parameters, implicitly spans the search space. In case of the `enumerated_set` class, the parameter interface is of the `enumerated_parameter` type. This interface manages parameters with finite enumerable value ranges. The `enumerated_set` class has a member `emplace` to add parameters to the set. This member function constructs the parameters in-place in the set. Its argument list is that of the constructor of the parameter type. In the example, a `log_parameter` type was used. The arguments of the constructor of the `log_parameter` class are the name of the parameter, the initial value, the base, the minimum value and the maximum value. This parameter type iterates the exponent for a given base in the interval limited by the supplied minimum and maximum value. If the value of the parameter is accessed, it returns the result of the exponentiation. Therefore, the parameter in the example enumerates powers of two between 1 and 16.

As the next step, a search strategy is set up to auto-tune the compute kernel. The `tuners` namespace contains templates that implement search strategies. In the example, a brute-force tuner was chosen that exhaustively iterates the search space. It gets instantiated with the compute kernel and the parameter set that spans the search space.

Listing 4.5: AutoTuneTMP kernels offer an API for exchanging the objective of the auto-tuning process.

```
1 void set_duration_functor(std::function<double()> f);
2 bool has_duration_functor(); // to check whether a functor was set
3 double get_duration(); // used to measure performance internally
```

All tuners use template metaprogramming to adapt to the signature of the compute kernel. For example, the call to the `tune` method in Line 10 has the same argument list as the compute kernel. This was derived in the instantiation of the tuner template from the type of the compute kernel.

The `tune` method performs the auto-tuning. It returns the best parameters discovered by the search strategy. During auto-tuning, the tuner calls the compute kernel whenever a combination of parameter values needs to be evaluated. However, to call the kernel its arguments need to specified. By using the same argument list as the compute kernel, input data for the compute kernel can be conveniently specified as arguments to the `tune` method. As mentioned, arguments should be either not modified during kernel execution (pass-by-reference) or they should be copyable (pass-by-value). After the auto-tuning, the returned parameter values get assigned to the compute kernel (Line 12). Further calls of the compute kernel will now use the auto-tuned parameters.

By default, AutoTuneTMP minimizes the runtime of a tuned compute kernel. To specify a different tuning objective, AutoTuneTMP uses three functions whose signatures are shown in List. 4.5. The `set_duration_functor` function can be used to set up any function that measures performance in a user-defined metric. As the function returns a double-precision floating-point value, AutoTuneTMP currently requires the objective to be represented as a double value. During kernel performance evaluations, AutoTuneTMP checks whether a duration functor was set and, if true, the search strategy minimizes its returned value. For generality, the parameter list of the duration functor is empty. Therefore, runtime information of the kernel needs to be provided as state. For example, if the duration functor is implemented as a lambda function, it can access data written by the kernel through its capture list.

Other extension points further customize the auto-tuning process. Through the `set_valid_parameter_combination_functor` method, which is an extension point of the compute kernel, the *validate-parameters* functor can be set. As the name suggests, the `set_adjust_parameters_functor` method allows for specifying the *adjust-parameters* functor, though it is a method of the tuner. Finally, the `set_test_functor` method of

Listing 4.6: A compute kernel for blocked matrix-vector multiplication. The blocking parameter allows for a reusage of the loads from **v** even if **v** is too large for the cache.

```
#include "autotune_kernel.hpp"

AUTOTUNE_EXPORT vector<double> mv_kernel(vector<double> &m,
    vector<double> &v) {
  const size_t N = v.size();
  vector<double> result(N, 0.0);
  for (size_t i = 0; i < N; i += BLOCKING)
    for (size_t j = 0; j < N; j++)
      for (size_t k = 0; k < BLOCKING; k++)
        result[i + k] += m[(i + k) * N + j] * v[j];
  return result;
}
```

the tuner enables setting the *test-result* functor.

4.3.3. Auto-Tuned Kernels and Optimization Templates

The most straightforward (but inconvenient) approach for implementing an auto-tunable kernel is shown in List. 4.6. This matrix-vector multiplication compute kernel implements a blocking approach that, instead of iterating single matrix rows, iterates a number of matrix rows simultaneously. If the vector **v** is large enough to not fit into the caches, the blocking approach can improve the performance of this memory-bound algorithm by up to 2x.

AutoTuneTMP sets up parameters as C preprocessor definitions (`#define`)[1]. These definitions are included via the `autotune_kernel.hpp` header which is updated right before JIT compilation to reflect the specified parameter values.

As the compute kernel in List. 4.6 shows, even a single parameter can increase the technical complexity of a compute kernel significantly. Because of this, AutoTuneTMP offers a collection of parameterized C++ templates that simplify the development of auto-tunable compute kernels. We refer to this approach as the optimization template approach.

The optimization template approach delegates parts of the optimization process to a library and thereby shifts parts of the technical burden from the compute kernel devel-

[1] While not currently implemented, `constexpr`-variables could be used to realize type-checked parameters.

Listing 4.7: A template for blocked loop exchange. A index range of the outer loop, controlled by the blocking parameter, is executed for each iteration of the inner loop.

```
1 struct loop_spec {
2   size_t start, stop, step;
3   loop_spec(size_t start, size_t stop, size_t step)
4     : start(start), stop(stop), step(step) {}
5   loop_spec(size_t start, size_t stop) : start(start), stop(stop),
6     step(1) {}
7 };
8 template <size_t blocking, typename F> void blocked_exchange(
9     const loop_spec &outer, const loop_spec &inner, F body) {
10   for (size_t i = outer.start; i < outer.stop; i += blocking)
11     for (size_t j = inner.start; j < inner.stop; j += inner.step)
12       for (size_t ii = 0; ii < blocking; ii += outer.step)
13         body(i + ii, j);
14 }
```

oper to the developer of the library. From a high-level perspective, we therefore use a common pattern. However, while there are many libraries that provide high-performance application-specific building blocks for applications, few target the development of compute kernels in general. AutoTuneTMP combines C++ metaprogramming for general optimization with auto-tuning. Compared to frameworks that use template metaprogramming for a similar purpose, parameters of optimization templates do not need to be chosen statically with an optimal value by the application developer. Instead, the template parameters can be tuned at runtime. Because of this, we call this a runtime template metaprogramming approach. We consider this a significant improvement over approaches that solely rely on C++ metaprogramming and a standard C++ compiler.

As an illustration of the optimization template approach, we reimplement the matrix-vector multiplication from List. 4.6 with a simple optimization template. The template, shown in List. 4.7, implements a blocked loop exchange to perform the same task as the manual loop exchange shown in List. 4.6. Its first two arguments are the boundaries of the loops. Depending on the template parameter `BLOCKING`, a number of iterations of the outer loop are moved into the inner loop. The third argument to the template is a callable, i.e., the loop body. It gets called for each index pair of the outer loops.

Listing 4.8 shows how the blocked loop exchange template can be used for reimplementing the matrix-vector multiplication kernel. The `BLOCKING` parameter of the compute kernel is now used to control the blocking as implemented by the template. Because

Listing 4.8: A matrix-vector kernel that uses the loop exchange template from List. 4.7.

```
AUTOTUNE_EXPORT vector<double> matrix_vector(vector<double> &m,
    vector<double> &v) {
  const size_t N = v.size();
  vector<double> result(N, 0.0);
  blocked_exchange<BLOCKING>({0, N}, {0, N},
    [&](size_t i, size_t j) {
      result[i] += m[i * N + j] * v[j];
    });
  return result;
}
```

of uniform initialization, the loop boundaries can be specified in curly braces. The loop body is implemented as a lambda function. By using this template, the implementation of the loop exchange is delegated to a library implementation and the intention of the optimization is explicit through the type of template chosen.

For more complex compute kernels, the template approach is often more readable, as functionality that would otherwise be implemented by the compute kernel can be replaced by library functionality. This will (hopefully) become apparent when we describe our auto-tuned matrix multiplication in Chapter 5. In general, by using a highly-optimized library the level of performance achieved can surpass that of a manually-optimized compute kernel, especially if time or node-level optimization knowledge are limited.

4.4. Generalized Kernels and OpenCL

The generalized compute kernel concept can be used to auto-tune compute kernels that do not fit the CPPJIT approach described above. As shown in List. 4.9, they are declared similarly to a CPPJIT kernel by specifying the signature of the kernel and a symbol name for it. However, to fully set up the generalized kernel, two functors need to be registered. First, through the `set_kernel_functor` method of the kernel object the implementation of the compute kernel has to be set. Any `std::function` object that implements the signature of the kernel can be registered. Second, the `set_apply_parameters_functor` method needs to be used to set up a functor that applies parameter values to the kernel. This functor gets called whenever the parameter values of the kernel are changed, e.g., right before the evaluation step during tuning. With the kernel fully set up, it can be

Listing 4.9: The macro for declaring a generalized kernel with the same signature as the CPPJIT `square` kernel from List. 4.1. Two functors of the kernel need to be set up before use.

```
1 AUTOTUNE_GENERALIZED_KERNEL(vector<double>(vector<double>), square)
2 /* functions to set up the mandatory functors */
3 void set_kernel_functor(std::function<R(Args...)> f);
4 void set_apply_parameters_functor(
5     std::function<void(parameter_value_set &)> f);
```

tuned just like a CPPJIT compute kernel.

In this work, we use the generalized compute kernel type for the auto-tuning of OpenCL kernels. OpenCL implementations both provide their own JIT compiler and, currently, do not use C++ as a kernel language. By extending auto-tuning to OpenCL kernels, we can use our framework to auto-tune GPU devices in particular, though the auto-tuning of other OpenCL devices is possible as well. The most recent version of the OpenCL standard specifies C++ as a kernel language [147]. Thus, in the near future OpenCL implementations with C++ as a kernel language should become available. This will allow for a use of optimization templates in OpenCL kernels.

In the following paragraphs, we describe the core features of OpenCL. For further details we refer to the OpenCL standard [147]. From a high-level perspective, OpenCL is very similar to Nvidia's CUDA. OpenCL is more general, as it can be used not only on GPUs but also on processors, accelerators and even FPGAs. And, of course, it is not tied to a single vendor. Because of its generality, the mapping of OpenCL abstractions to hardware features is not prescribed by the OpenCL standard.

At the beginning of the execution of an OpenCL kernel, many work-items are started that all execute the same compute kernel. A work-item can be thought of as analogous to a thread, as each work-item behaves as if it executes an individual instruction stream. However, its mapping to the hardware depends on the hardware platform. Each work-item has a unique index that can be used to assign an individual task to a work-item. The indices of the work-items are generated through a rectangular one-to-three-dimensional index space called NDRange. An NDRange needs to be specified whenever a kernel gets scheduled for execution.

Work-items are not executed individually, instead they are grouped into work-groups. Work-groups are groups of several hundreds or even thousands of work-items. During execution, work-items can collaborate by exchanging data within a work-group. Synchronization on the work-group level is possible as well. However, global synchronization

between work-groups is not supported.

Vectorizing OpenCL implementations execute multiple work-items simultaneously by mapping work-items to individual SIMD lanes. These implementations generally employ masking so that individual work-items can execute or skip instructions. All OpenCL implementations used in this work are vectorizing implementations. Thus, we can write scalar compute kernels which still get vectorized, as vectorization is handled on the framework level. For this type of implementation, avoiding branches is important, as branches affect the effectiveness of the masking approach. Generally, if work-items do not agree on whether to take a branch, both outcomes are executed with the SIMD lanes partially masked in each case. However, branches with the same outcome across work-items do not incur a significant performance penalty. Such branches occur, for example, if the condition of the branch depends on shared data only.

Having described some core features of the execution model, we briefly look at the memory model of OpenCL. The OpenCL memory model specifies multiple types of memory. Most important are the global memory, which usually is the main memory of a device, and the work-group-level local memory [147]. The local memory generally has a size in the kilobyte range and is managed manually. On devices where the local memory is not only an OpenCL abstraction but realized in hardware, its performance matches the performance of a fast cache. GPU devices physically implement a local memory and it is usually highly advantageous to use it, as some GPUs do not have L1 caches and no or only small L2 caches. Furthermore, OpenCL assumes separate address spaces for devices executing the kernels and the host, i.e., where the OpenCL framework runs. This necessitates transfers of data between the host and the devices. However, on some more recent platforms, shared virtual memory reduces the cost for copying the data. Shared virtual memory automates the transfer between the address spaces.

Abstracting from OpenCL

As part of this work, we created a set of abstractions for OpenCL that are made available through AutoTuneTMP. The most important feature is an OpenCL manager abstraction that greatly simplifies the management of multiple OpenCL devices simultaneously. Additionally, it can be used to specify the parameters of auto-tuned OpenCL kernels.

An example of the OpenCL abstraction layer is shown in List. 4.10. In this example, we configure an OpenCL kernel with a one-dimensional grid and a work-group size of 128. For selecting the OpenCL devices to be used, a configuration file is supplied to the constructor of the manager object. Then, the OpenCL kernel gets compiled. As

Listing 4.10: An example use of the OpenCL abstractions provided by AutoTuneTMP. A configuration is used to select the desired devices and a kernel is built. Then, multiple threads set up a buffer for their respective device, move the buffer data to the device and finally run a compute kernel.

```
constexpr size_t grid_1d = ...;
constexpr size_t group_size = 128;

opencl::manager_t manager("config.cfg");
// build kernel for all configured devices
vector<cl_kernel> kernels =
  manager.build_kernel(src_str, "kernel_name", cpp_flags);
// each OpenCL device is controlled by one host processor thread
#pragma omp parallel for
for (size_t i = 0; i < manager->get_devices().size(); i += 1) {
  auto &device = manager->get_device(i);
  auto &device_kernel = kernels[i];
  opencl::managed_buffer<double> buffer(device, 1000);
  buffer.to_device(host_data);
  opencl::apply_arguments(device_kernel, buffer->get());
  opencl::run_kernel_1d(device, device_kernel, grid_1d, group_size);
}
```

the manager object is aware of the configured devices, it can compile the kernel for all necessary devices. In the example, the implementation of the kernel is provided as a source string. After the compilation has finished, a managing thread for each device is started on the host. Each thread sets up an array of doubles on the devices through the `managed_buffer` abstraction. This class manages the resources of a device buffer, i.e., it performs memory allocation, reallocation and deallocation. To transfer data from the host to the device, the `to_device` method is called. In OpenCL, the arguments of the kernel need to be set before execution which is achieved by the call to the `apply_arguments` function. In the example, a kernel with a single argument was assumed. The implementation of `apply_parameters` can be found in Sec. A.1, as applying arguments with the basic OpenCL API proves to be surprisingly tedious. After the argument is applied, the kernel is executed by the call to the `run_kernel_1d` function. This function executes the kernel, halts the host thread until execution has finished and checks whether an error occurred.

As the selection of OpenCL devices and platforms is cumbersome, AutoTuneTMP simplifies the configuration through hierarchical key-value configuration files that use the JSON file format. An example is shown in List. 4.11. In the example, two OpenCL

platforms from different vendors are configured. Each platform has one type of device. The configuration specified that exactly two GTX 1080 Ti GPUs are to be used through the `COUNT` key. As the configuration of the Xeon Gold does not have this key, all available devices of this type are used. However, multi-socket systems are treated as one device by the Intel OpenCL implementation. Therefore, there can only be a single device of this type.

Each OpenCL device of each platform can be set up with different parameter values for each compute kernel. The parameters are interpreted by the kernel and their meaning therefore depends on the specific kernel. However, there are some parameters that are common to all kernels. The OpenCL manager class can be set to report on its tasks through the `VERBOSE` parameter—a feature useful for debugging. The `OPTIMIZATION_FLAGS` key is yet another example. Its value gets forwarded to the JIT compiler of the OpenCL platform when the OpenCL kernel is built. The `COUNT` key is also part of the configuration framework, though it operates on the device and not on the kernel level. There are further options, e.g., for displaying the build log after compilation.

All OpenCL kernels in this work were configured through the presented JSON-based configuration approach. The configuration shown is based on one of the actual configuration files used for evaluating the unified streaming algorithm that we describe in Sec. 7.1.1.

4.5. Parameter Spaces and Search Strategies

Having described the two types of compute kernels, we next describe the parameter types and search strategies supported by AutoTuneTMP. Note that we do not model general application parameters, but attempt to model parameters common in performance optimization, e.g., related to optimizations such as cache blocking or parallelization. Considering our auto-tuned kernels[2] and the auto-tuning literature, as discussed in Sec. 2.2.3 and summarized in Tab. 2.1, we observed some shared characteristics of parameters and search spaces. Parameters are generally discrete-valued. The value ranges are usually small, there are often less than 20 values, and the dimensionality of the search spaces is low to moderately-high. In our experiments, we tuned compute kernels with at most 11 parameters. Considering Tab. 2.1, parameter tuning approaches were used for auto-tuning problems with at most 15 dimensions. We further observed that

[2]For the parameter specifications see Tab. 5.1, Tab. 7.2 and Tab. 7.4

Listing 4.11: An example of the hierarchical configuration approach for managing OpenCL devices and setting up parameters of compute kernels. This JSON configuration file is interpreted by the manager class.

```
{
    "PLATFORMS": {
        "Intel(R) OpenCL": {
            "DEVICES": {
                "Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz": {
                    "KERNELS": {
                        "RegressionUnified": {
                            // not all kernel parameters shown
                            "KERNEL_DATA_BLOCK_SIZE": "8",
                            "KERNEL_GRID_SPLIT": "8",
                            "LOCAL_SIZE": "128",
                            "OPTIMIZATION_FLAGS": "...",
                            "VERBOSE": "true"
                        }
                    }
                }
            }
        },
        "NVIDIA CUDA": {
            "DEVICES": {
                "GeForce GTX 1080 Ti": {
                  "RegressionUnified": {
                    ...
                  }
                  // if omitted: use all devices
                  "COUNT": "2"
                }
            }
        }
    },
    "OCL_MANAGER_VERBOSE": false,
    "SHOW_BUILD_LOG": false
}
```

many parameters were independent of the other parameters. If a parameter was not independent, it often only depended on a small subset of parameters.

Based on these observations, we primarily use three discrete-valued parameter types throughout this work: enumerated parameters, log parameters and set parameters. All are implemented as classes in AutoTuneTMP. An enumerated parameter has a minimum, a maximum and a fixed step size to span a discrete value range. The log parameter, which we already encountered in Sec. 4.3.2, does the same as the enumerated parameter, but iterates the exponent of a given base. Its value is the result of the exponentiation of the base with the current exponent value. The set parameter is specified with a fixed set of values.

We use one additional parameter type that serves a special purpose: the constant parameter. Constant parameters are useful for values that are chosen at runtime, but nevertheless have a single fixed value. For example, in the regression kernels we present in Chapter 7, the dimensionality is problem-dependent, but fixed for a given problem. By describing the dimensionality as a parameter, it becomes a constant from the perspective of the JIT compiler which sometimes enables additional optimization, e.g., because a loop trip count is known.

AutoTuneTMP implements a set of heuristics as search strategies. We have already encountered brute-force search, or exhaustive search, in List. 4.3.2. However, this strategy is not an option for most compute kernels and primarily useful for testing. Further strategies implemented in AutoTuneTMP are line search, neighborhood search and Monte Carlo search. Line search considers the values in one dimension at a time and optimizes it independently of the other parameter values. As the optimal value of one parameter might change after a different value was chosen for another parameter, a dependency between the parameters, the parameter set is iterated multiple times. Due to the caching approach implemented by AutoTuneTMP, it is possible to perform repeated iterations of the parameter set until no further yet unseen parameter combination is to be tested. The search can be aborted early by specifying a maximum number of parameters to iterate. This search strategy needs to be supplied with initial values for the parameters.

Neighborhood search considers “neighboring” values of the current parameter values in all dimensions. What is considered to be a “neighbor” value depends on the type of parameter used, as compatible parameters need to implement `prev` and `next` functions. To improve the current parameter values, neighborhood search evaluates all neighbors and then selects the highest-performance parameter values for the next search iteration.

This is repeated until an improvement is no longer possible or a maximum number of search steps is reached. Similar to line search, this search strategy requires a set of initial parameter values.

AutoTuneTMP implements two variants of the basic neighborhood search algorithm. The first variant, simply called neighborhood search, considers all neighbors with exactly one parameter changed. The second variant, which we call full neighborhood search, considers the cross product of three possible values per parameter: the "previous" value, the current value and the "next" value. Full neighborhood search is better suited for parameters with interdependencies. For d parameters and a single search step, neighborhood search considers $2 \cdot d$ neighbors. Full neighborhood search, however, iterates $3^d - 1$ parameter value combinations. Consequently, full neighborhood search can only be applied to low-dimensional search problems.

A further search strategy is Monte Carlo search, which repeatedly draws random parameter combinations—as the name suggests. We primarily employ this search strategy to evaluate the directed search strategies. Furthermore, it is useful for assessing the difficulty of a search problem. For example, data collected over a tuning run allow for an approximation of the expected performance of the search space.

In our experiments, we noticed that modifying the search space proved to be a more straightforward solution than moving to more complex search strategies. AutoTuneTMP currently provides two means for reducing the complexity of the search space: the *adjust-parameters* functor and the group tuner. The *adjust-parameters* functor can be used to eliminate dependencies between parameters. We use it for that purpose in Sec. 7.2, as a padding parameter needs to be set properly depending on a parameter that controls loop unrolling. Without the parameter adjustment, neighborhood or line search could not explore relevant parts of the parameter space, as they only modify a single parameter at a time.

The group tuner is a meta search strategy. With this tuner, it is possible to split the parameters into smaller groups that are tuned independently. Thereby, irrelevant search dimensions from the perspective of a group of parameters do not need to be considered when tuning a specific parameter group. Each group of parameters gets tuned by a provided search strategy and the group tuner manages the updating of the parameters between the parameter groups. We generally use the same search strategy for all parameter groups and refer to the resulting tuner by the name of the subordinate search strategy with the prefix "split", e.g., split line search for a group-tuned line search.

To further improve auto-tuning performance, AutoTuneTMP adds variants of search

strategies that perform the compilation step in parallel. Parallel compilation can be immediately used with CPPJIT kernels and the standard provided builders. However, parallel compilation of generalized kernels requires that the compilation step is designed for parallel compilation by the application developer, as the compilation of a kernel of this type is not controlled by AutoTuneTMP. Currently, there are parallel line search, parallel neighborhood search and parallel full neighborhood search. All address the same issue. With our standard GCC builder the compilation is a single call to GCC and therefore a serial operation. As the compilation of a single C++ kernel source file can take up to multiple seconds, the time required for compilation can be significant. The parallel tuners bundle all compute kernel variants scheduled for evaluation by the search strategy and compile them simultaneously. Note that the evaluation step is still performed serially, as the compute kernels might use parallelization internally.

Adding Parameter Types and Search Strategies

To handle applications that require parameter types not covered by the types described above, AutoTuneTMP can be easily extended. Search strategies are implemented against parameter interfaces. Listing 4.12 shows the `enumerated_parameter` interface. The core components of this parameter interface are the methods for moving from the current value to the next and the previous value, i.e., `next` and `prev`. These methods return `false` if the current value already is the maximum or minimum value, respectively. Furthermore, the `set_min` method resets a parameter to a value that can be used to enumerate the whole value range using the `next` method. The `count_values` method returns the number of values of the parameter. All parameter classes are furthermore required to implement copy constructors.

Because the parameter sets use type-erasure, essentially duck-typing through template instantiation, parameter types do not need to inherit the parameter interface types. This improves maintainability, as otherwise parameters would need to inherit the interface types of all parameter sets they are to be used with. Extensibility is improved as well, as a new parameter type only needs to implement the correct methods, but does not depend (through inheritance) on any AutoTuneTMP class. Similarly, because of the parameter set types as intermediates between parameters and search strategies, new search strategies can be added by implementing them against the parameter interfaces of the parameter sets. Thereby, all parameter types compatible with the parameter set type are immediately compatible with the new search strategy.

Listing 4.12: The parameter interface used by the `enumerated_set` parameter set type.

```
class enumerated_parameter {
public:
  virtual std::string get_name() const = 0;
  virtual std::string get_value() const = 0;
  virtual bool next() = 0;
  virtual bool prev() = 0;
  virtual void set_min() = 0;
  virtual size_t count_values() const = 0;
};
```

4.6. Evaluating the Contribution of Auto-Tuning

To highlight the benefit of auto-tuning and to demonstrate the usefulness of AutoTuneTMP, we need a baseline for the performance results obtained through auto-tuning. This baseline should reflect the performance of the kernel with auto-tuning disabled or not even implemented. In general, certain parameter value combinations might perform much worse than even a naive implementation. By using such parameter value combinations as the baseline, speedups could be artificially inflated. This eliminates the obvious choice of comparing the best-performing parameterization to the worst-performing parameterization.

Alternatively, one could approximate the expected performance by random sampling of the search space, i.e., a Monte Carlo approach. Then, we could compare the performance result obtained through auto-tuning to the expected performance. The speedup computed from this approach reflects the composition of the search space. Generally, we cannot assume that the number of fast and slow parameter combinations average out to some plausible performance baseline. Instead, a search space might be heavily biased towards either fast or slow combinations. Therefore, the measured performance does not necessarily correspond to a variant of the kernel without auto-tuning.

Instead of these approaches, we use an application-dependent baseline with all parameters set to a value that approximates the same compute kernel without the optimizations associated with the parameters. As AutoTuneTMP is not a parallelization or vectorization framework, parallelization and vectorization are both enabled in the baseline parameter combination. The baseline therefore is parallel, vectorized and non-optimized. We refer to such a parameterization as a PVN parameterization. Because parallelization and vectorization are enabled, this is baseline can be considered challenging.

For the selection of parameter values that put optimizations in a neutral state, there is

no application-independent strategy. A neutral value depends on the type of parameter and the compute kernel. Therefore, neutral parameter values need to be defined for each compute kernel individually. To demonstrate that a well-chosen PVN combination was used in the experiments discussed in this work, we will state the PVN parameter values in the evaluation sections of the auto-tuned applications.

5. A High-Level Auto-Tuned Matrix Multiplication

In this chapter, we more deeply investigate the CPPJIT approach for auto-tuning and writing compute kernels. We have already presented the basic approach in the last chapter. However, we so far have only covered a simple matrix-vector multiplication example intended to showcase the optimization loop and API of AutoTuneTMP. In the following sections, we demonstrate that the auto-tuning approach provided by AutoTuneTMP can achieve near-optimal performance. To that end, we focus on an auto-tuned matrix multiplication kernel. We use double-precision arithmetic throughout this chapter.

The multiplication of dense matrices is one of the most basic operators for solving linear algebra problems. In the following, we consider the matrix multiplication of two real-valued matrices $C = A \cdot B$. We assume the matrices C, A and B with dimensionalities $N_x \times N_y$, $N_x \times N_k$ and $N_k \times N_y$, respectively. Though algorithms with better complexity exist [34, 61], commonly matrix products of dense matrices are computed using the standard $\mathcal{O}(n^3)$ operations algorithm that computes each component of C directly by calculating $c_{i,j} = \sum_{k=1}^{N_k} a_{i,k} b_{k,j}$. We focus on this algorithm because of its relevancy and because it is a common optimization test case in HPC, e.g., as part of LINPACK [41].

Dense matrix multiplication is a popular object of study. Developing a implementations of the standard $\mathcal{O}(n^3)$ algorithm with near-optimal performance proves to be challenging, as the literature for high-performance implementation approaches shows [66, 63, 96, 76]. One could (naively) assume that modern compilers can deliver a substantial fraction of the peak performance of a modern processor. We can test this assumption by considering the naive implementation displayed in List. 5.1. As compilers do not perform parallelization, OpenMP is used so that at least in principle all resources of the processor can be used. We executed this algorithm on a modern processor with four Intel Skylake cores clocked at 4 GHz with all relevant optimization flags[1]. However, the

[1] GCC 7 was used with the flags `-O3 -march=native -mtune=native -ffast-math -fopenmp`

Listing 5.1: A naive implementation of the standard $\mathcal{O}(n^3)$ matrix multiplication algorithm that uses OpenMP for parallelization.

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      C[i * N + j] += A[i * N + k] * B[k * N + j];
```

resulting performance disappoints. For square $N \times N$ matrices with $N = 4096$, out of 256 GFLOPS only 0.67 GFLOPS are achieved. This equals approximately 0.3% of the peak performance of the processor.

On most modern hardware platforms, including Intel's Skylake cores, an FMA-only kernel is assumed for the calculation of the peak performance. As most modern hardware platforms achieve their optimal performance performing FMA instructions and because the arithmetic intensity of this algorithm is high enough, peak performance is an upper bound for the achievable performance on most platforms. However, even the best implementations achieve slightly lower performance. Necessary steps for high-performance implementations, especially cache blocking, introduce cycles spent performing instructions other than FMAs.

For a high-performance variant of the basic algorithm, a range of optimization issues need to be addressed. Though matrix multiplication offers ample opportunity for reusing data in the cache, a careful ordering of the arithmetic operations is required for the available memory bandwidth to be sufficient. This is commonly addressed by a cache blocking approach. Of course, as the targets are multi- and many-core platforms, the implementation needs to be parallelized. Furthermore, as all relevant hardware architectures in HPC implement wide vector units, the algorithm needs to be vectorized. And finally, on multi-socket machines the limited inter-socket bandwidth needs to be considered.

In the remainder of this chapter we present and evaluate an auto-tuned matrix multiplication algorithm written using only high-level constructs. We use our optimization template collection to implement the cache-level and register-level blocking, OpenMP for parallelization and the C++ library Vc for vectorization. The resulting compute kernel exposes 11 parameters that are tuned by AutoTuneTMP. We will demonstrate that the obtained implementation is competitive with the best implementations available, even those that were (likely) hand-written in assembly. Thereby, we show that Au-

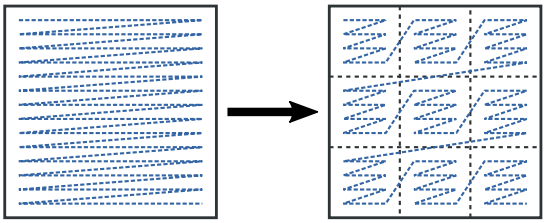


Figure 5.1.: Rewriting of a row-major matrix into tiles. The tiled matrix improves spatial data locality for accesses within a tile.

toTuneTMP is applicable for the optimization of challenging compute kernels without compromising on performance. And, as optimization templates are an essential tool for the approach to be high-level, we demonstrate their usefulness.

In Sec. 5.1, we present the approach for addressing issues related to the memory hierarchy . Parallelization is covered in the subsequent section, i.e., Sec. 5.2. In Sec. 5.3, we address core-level issues such as vectorization and register blocking. We discuss our approach for the remaining minor performance issues and give an overview of the auto-tunable parameters exposed by the compute kernel in Sec. 5.4. Having presented our auto-tuning-enabled algorithm, we proceed with an evaluation on four hardware platforms in Sec. 5.5. As part of the evaluation, we compare six of the search strategies provided by AutoTuneTMP.

5.1. Cache Blocking

Our cache-blocked algorithm uses a matrix tiling approach to improve data locality. Tiling changes the order of the elements of an array so that memory addresses within potentially higher-dimensional subarrays access a contiguous range of memory addresses. Figure 5.1 shows how a row-major matrix is reordered into square two-dimensional tiles. Of course, the shape and dimensionality of the tiles need to be adjusted for a given problem. Unfortunately, the nomenclature in the literature is not consistent. We refer to cache blocking as all schemes that improve data locality by reordering operations. In our nomenclature, tiling always refers to approaches that reorder the data itself.

Tiling can offer multiple benefits over pure cache blocking approaches, mainly because a smaller array stride is used for accessing data within a tile compared to the overall array. Firstly, as a tile is contiguous in memory, the number of entries required in the

translation-lookaside buffer (TLB) is minimized, as the number of pages is minimized. Without tiling each accessed row might require a TLB entry, even if a tile would fit into a single page. For arrays with large strides, this case unfortunately is realistic. Secondly, assuming that the stride of the tile is smaller than the size of the cache-tags, tiling helps avoiding cache conflict misses for large strides that are (multiples of) powers of 2. Lastly, the prefetchers only need to prefetch a single chunk of memory for higher memory bandwidth and more easily detectable prefetching patterns.

AutoTuneTMP provides a set of templates for tiling. The `make_tiled` template creates a tiled copy of a matrix and is designed for C++'s `std::vector`. Its sole template parameter is the dimensionality of the tiles. At runtime, it receives the array to tile as its input, as well as the dimensionalities of both the tiles and the array. Listing 5.2 shows how the `make_tiled` template is used to obtain a matrix of 2×4 tiles from an overall $N \times N$ matrix. The `undo_tiling` template implements the inverse operation of the `make_tiled` template and is shown in the last line of List. 5.2. The function template `iterate_tiles` performs a row-wise iteration of the tiles and facilitates performing per-tile operations. Analogously to the `make_tiled` template, its one template parameter is the dimensionality of the iteration space. As runtime arguments, it requires the tiled array, the tiling configuration and, lastly, a function that is called for each tile iterated.

The lambda that is used as the operation to execute by the `iterate_tiles` template shows yet another helpful template. Its single argument is an instance of the `tile_view` class. This class provides a view on a tile as if the tile were a standalone array, thereby simplifying the tile indexing. In the example, the tile view is set up automatically by the `iterate_tiles` template.

An illustration of our cache-blocking scheme is displayed in Fig. 5.2. We chose two layers of cache blocking. The outer blocking approach is controlled by the three variables $x_{\text{out}}, y_{\text{out}}$ and k_{out}. x_{out} and y_{out} split the result matrix into rectangular blocks. From the perspective of each block, the input matrices now consist of two relevant bands. Through the k_{out} parameter, the bands in the input matrices are again split into equally-sized blocks. As a result of this classical scheme, the blocks in the input matrices as well as the block in the result matrix have a size that no longer depends on the overall size of the matrices. Consequently, if x_{out}, y_{out} and k_{out} are chosen properly, the blocks can fit into the cache. The operations on the block-level are multiplications of block-sized matrices of dimensionalities $x_{\text{out}} \times y_{\text{out}}$, $x_{\text{out}} \times k_{\text{out}}$ and $k_{\text{out}} \times y_{\text{out}}$. Therefore, each block performs $2x_{\text{out}}y_{\text{out}}k_{\text{out}}$ arithmetic operations for $x_{\text{out}}y_{\text{out}} + x_{\text{out}}k_{\text{out}} + k_{\text{out}}y_{\text{out}}$ data elements. We observe that the $x_{\text{out}} \times k_{\text{out}}$ submatrix of A is reused y_{out}-times and the

Listing 5.2: A two-dimensional array is converted into a tiled representation of 2×4 tiles. The tiles are then iterated and a per-tile operation is performed. Finally, the original matrix layout is restored.

```
vector<double> m = ...;
opttmp::tiling_configuration conf = {{2, N}, {4, N}};
auto tiled = opttmp::make_tiled<2>(m, conf);
opttmp::iterate_tiles<2>(tiled, conf,
    [](opttmp::tile_view<2> &view) {
  for (size_t i = 0; i < 2; i++) {
    for (size_t j = 0; j < 4; j++) {
      view[i * 4 + j] = ...;
    }
  }
});
m = opttmp::undo_tiling<2>(tiled, conf);
```

$k_{\text{out}} \times y_{\text{out}}$ submatrix of B is reused x_{out}-times. Note that entries of the submatrix of C are reused less, as during the computation of a component the result value can be kept in the registers. Therefore, updating C requires only one read and one write per component for each of the N_k/k_{out} band pieces.

The second inner blocking layer replicates the idea of the outer blocking layer with $x_{\text{in}}, y_{\text{in}}$ and k_{in} for the blocking dimensions. By employing two layers of cache blocking, the algorithm can adapt to a multi-level cache hierarchy. Fitting into the L1 and L2 caches requires relatively small matrices for less reuse. The outer blocking layer allows for reuse in the L3 cache (where available) with larger submatrices. As each blocking layer has three parameters, there are overall six parameters related to cache blocking.

The resulting blocking scheme, written in C++, is shown in List. 5.3. Because of the two layers of blocking, the loop nest employed for cache blocking is six loops deep. The register-blocked inner part of the compute kernel is presented in the next section, as it implements a set of further optimizations. While the conversion of the input matrices into a tiled representation happens before the operation is called, List. 5.3 shows the use of `tile_view` instances to move to different tiles of the input matrices. The dimensions of the tiles match the dimensions of the blocks implied by the inner blocking layer. That is, the input matrix A is tiled into $x_{\text{in}} \times k_{\text{in}}$ tiles and the input matrix B is converted to a representation with $k_{\text{in}} \times y_{\text{in}}$-sized tiles. Note that C is initially set to zero and therefore does not need to be adjusted. However, as the algorithm assumes a tiled representation of C, C gets converted to row-major representation after the loop nest.

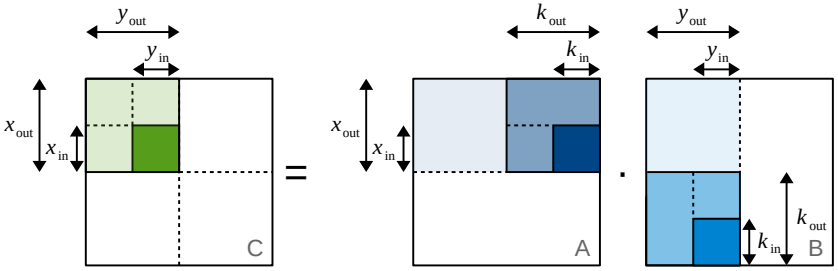


Figure 5.2.: The two cache-blocking layers of the matrix multiplication. Though square in the illustration, rectangular shapes are allowed. In the actual implementation, the left input matrix A is transposed.

In the implementation, A is used transposed in an attempt to improve prefetching. As the resulting performance improvement is minimal, we do not discuss this in further detail.

For the approach shown in List. 5.3 to be correct, certain constraints need to be fulfilled. The matrices need to be padded according to $x_{\text{out}}, y_{\text{out}}$ and k_{out}. Furthermore, each inner blocking variables needs to divide the corresponding variable of the next outer layer. For example, x_{in} needs to divide x_{out}. We describe further constraints, introduced by register blocking, in Sec. 5.3.

5.2. Parallelization

Listing 5.3 not only shows the cache blocking approach, but indicates the parallelization approach as well. By using OpenMP's collapse clause, the outer two loops are run thread-parallel, i.e., the outer blocks of C are the unit of parallelization.

Two parameters change the behavior of OpenMP. In order for the algorithm to enable or disable SMT, the number of threads is configurable. A further parameter was added to adjust OpenMP's scheduling behavior, motivated by an observed performance difference between static and dynamic scheduling. Static scheduling distributes work packages evenly across the threads as soon as the parallel region is entered. Dynamic scheduling uses a load-balancing scheme with threads fetching additional work after having processed their currently assigned work [112].

Listing 5.3: The loop nest that implements the two-layer cache blocking. The two outermost loops are collapsed into a single index range by OpenMP and used to parallelize the algorithm. Instances of the `tile_view` class facilitate the multiplication of submatrices. The vectorized register-level kernel is implemented by the `register_blocked_kernel` function, its implementation is displayed in List. 5.5.

```
1 // three outer loops, parallelize in x,y-dimensions
2 #pragma omp parallel for collapse(2), num_threads(OMP_THREADS), \
3   schedule(SCHEDULE)
4 for (size_t out_x = 0; out_x < X_size; out_x += x_out) {
5   for (size_t out_y = 0; out_y < Y_size; out_y += y_out) {
6     auto A_view = opttmp::make_view<2>(A_trans, tile_spec_A);
7     auto B_view = opttmp::make_view<2>(B, tile_spec_B);
8     auto C_view = opttmp::make_view<2>(C, tile_spec_C);
9     for (size_t out_k = 0; out_k < K_size; out_k += k_out) {
10      // three inner loops
11      for (size_t in_x = out_x; in_x < out_x + x_out; in_x += x_in) {
12        for (size_t in_y = out_y; in_y < out_y + y_out; in_y += y_in) {
13          C_view.move_to_tile({in_x, in_y});
14          for (size_t in_k = out_k; in_k < out_k + k_out;
15               in_k += k_in) {
16            A_view.move_to_tile({in_k, in_x}); // A was transposed
17            B_view.move_to_tile({in_k, in_y});
18            register_blocked_kernel(...);
19          }
20        }
21      }
22    }
23  }
24 }
```

5.3. Vectorization and Register Blocking

Fully utilizing all available resources of each core is critical for achieving close to peak performance. On most modern hardware architectures this entails that one or more vectorized FMA instructions are executed every cycle. Floating-point pipelines have a latency greater than one, e.g., the FMA latency of AMD's Zen architecture is five cycles [53]. Therefore, further independent instructions need to be scheduled after the first instruction has started to execute. On the Intel Knights Landing platform there are two pipelines that have a latency of 6 cycles for FMA instructions [53]. Therefore, 12 independent vector instructions need to be in-flight for optimal performance.

A second issue on the core level is the limited bandwidth to the caches, primarily the L1 cache. The x86 FMA3 instructions we use have three operands and compute $\mathbf{c} \mathrel{+}= \mathbf{a} \odot \mathbf{b}$, with the componentwise multiplication $\odot$, for 4-wide SIMD vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$. On many modern hardware platforms, at most two vector loads and one vector store can be performed. Computing an FMA3 instruction entirely in (cached) memory would require three loads and one store. This issue can be addressed by keeping data as much in the registers as possible.

For matrix multiplication, keeping values of C in the registers is a straightforward approach for reducing the required load and store bandwidths. By iterating the k direction, new values of A and B are required, but the same component of C is updated. As loads are needed for moving the component of C from the cache into a register and stores to write it back into the memory, the loop in the k direction should be long-running to minimize these additional loads and stores. For hardware platforms with a single FMA pipeline, keeping vectors of C in the registers might be sufficient. However, most hardware platforms implement two vector pipelines. Therefore, the L1 load and store bandwidth is still not sufficient as four L1 loads would be needed if components of A and B are streamed from memory each cycle.

To reduce the required cache bandwidth even further, we first observe that for a given k the components of one row of C share the same input value from A. Similarly, for a given k components of one column of C share the same input value from B. This motivates a two-dimensional register-blocking scheme analogously to the cache-blocking scheme with individual column and row slices from A and B—and a tiny submatrix of C. If the submatrix of C has enough entries, the register-blocking approach simultaneously addresses pipeline latency and limited load/store bandwidth. This is illustrated by Fig. 5.3 for a 4×2 register-blocking scheme and a SIMD-width of two. As the example

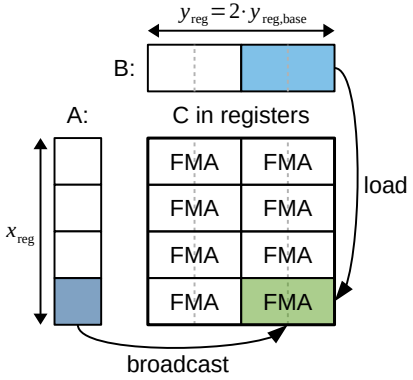


Figure 5.3.: The register-blocking scheme uses one row and column slice of each input matrix to update the whole register block. Shown is a 4×2 register block that is being updated; a SIMD width of 2 is assumed.

shows, for updating the 2×4 block 16 (scalar) FMAs are performed and only 8 values are loaded. Assuming that two FMAs with a width of two can be executed per cycle, 8 (scalar) values need to be loaded in 4 cycles. On platforms that support two vector loads per cycles, this blocking configuration would be sufficient for an algorithm that is not bound by L1 bandwidth. After four cycles, k is incremented to update the same submatrix of C with the next input values. As we require the prior FMA results for the next update, a pipeline latency of up to four can be tolerated. We denote the rows of a register block by x_{reg} and the columns by $y_{\text{reg,base}}$. $y_{\text{reg,base}}$ counts the columns in SIMD registers instead of double-precision values. From $y_{\text{reg,base}}$, we can compute y_{reg} by multiplying it with the SIMD-width of the architecture. To ensure correctness, x_{reg} needs to divide x_{in} and y_{reg} needs to divide y_{in}.

For vectorization, we use the vectorization library Vc as the basic building block [92, 91]. Vc implements a C++ approach for writing vectorized code independent of the vector-length and is portable across a wide range of hardware architectures. It mimics how scalar arithmetic is expressed in C++ and therefore offers a highly-convenient interface for vectorized codes. As scalar values and scalar constants get broadcasted to vector registers transparently, the resulting vectorized code is often identical to scalar code, except for the types[2]. For our matrix multiplication code, we use the `double_v` type of

[2]By necessity, Vc cannot treat branches as in the scalar case and instead uses a masking approach.

Listing 5.4: Declaration and use of register-blocked variables. The example shows how blocks of vectorized expressions can be written similar to scalar expressions.

```
1 using reg_array =
2   register_array<double_v, 5>;
3 reg_array a(data_ptr, Vc::vector_aligned);
4 reg_array b = 2.0;
5 reg_array r = 3.141 * (a - b);
```

Vc which represents a vector of doubles.

In order to facilitate the implementation of register blocking, AutoTuneTMP provides the `register_array` template. It mirrors Vc's scalar-like interface but works on an array of variables representing SIMD vectors. An example is shown in List. 5.4. In the example, first an alias for a `register_array` instantiation with five vector entries is declared. The base type is `double_v`. Therefore, each entry is a vector of doubles. In Line 3, a chunk of contiguous data is loaded from memory into a `register_array` variable. Assuming an aligned pointer, we can use aligned loads by specifying the flag `Vc::vector_aligned`. The scalar literal in Line 4 is broadcasted to the whole array of vector variables, highlighting the scalar-like interface of the `register_array` class (and Vc). The last line provides a second example for the scalar-like interface. The subtraction between two `register_array` instances can be written as in the scalar case. Again, a literal gets automatically broadcasted to the whole array of vector variables.

In List. 5.5, we show the inner register-blocked kernel of our matrix multiplication. Due to the high-level approach, we show the full C++ implementation. In the first line of List. 5.5, we declare an alias for the register-array type for register blocking with a row-width of $y_{\text{reg,base}}$-many vector registers. The two loops in Line 5 and 6 iterate the inner cache blocks of C in register-block steps. For a two-dimensional blocking approach, we declare an array of `reg_array` instances (Line 8). `acc` represents the tiny submatrix of C shown in Fig. 5.3 and is kept in the registers. The loop ranging from Line 10 to Line 17 implements the arithmetic operations required for the matrix multiplication. For each increment of k, $x_{\text{reg}} \times y_{\text{reg,base}}$ vector FMAs are performed. To that end, a row slice of B with the width of register block is loaded (Line 12). In Line 13, the rows of the register block are iterated and Line 15 performs the FMAs on a row of the register block. For each FMA operation, we need a single scalar from A per row. The scalar is broadcasted to a `double_v` vector. After the computational loop, `acc` gets added to the

Fortunately, matrix multiplication is branch-free.

Listing 5.5: Implementation of the register-level matrix multiplication kernel.

```
1  constexpr size_t y_reg = y_reg,base * double_v::size();
2  using reg_array = opttemp::register_array<double_v, y_reg,base>
3
4  // register blocking loops in x and y dimensions
5  for (size_t x = 0; x < x_in; x += x_reg) {
6    for (size_t y = 0; y < y_in; y += y_reg) {
7      // submatrix/accumulators, stored in x_reg · y_reg,base registers
8      std::array<reg_array, x_reg> acc;
9      // submatrix multiplication
10     for (size_t k = 0; k < k_in; k += 1) {
11       // stored in y_reg,base-many registers, usually cached in L1
12       reg_array b(B_view.pointer(k * y_in + y), Vc::vector_aligned);
13       for (size_t r = 0; r < x_reg; r += 1) {
14         // broadcast from memory, usually cached in L1 or L2
15         acc[r] += double_v(A_view[k * x_in + x + r]) * b;
16       }
17     }
18     // update register-level submatrix of C
19     double *res_ptr = C_view.pointer(x * y_in + y);
20     size_t offset = 0;
21     for (size_t r = 0; r < x_reg; r += 1) {
22       reg_array res_value(res_ptr + offset, Vc::vector_aligned);
23       res_value += acc[r];
24       res_value.memstore(res_ptr + offset, Vc::vector_aligned);
25       offset += y_in;
26     }
27   }
28 }
```

submatrix of C in the Lines 19 to 26. The overhead for adding the computed results to the submatrix of C further highlights that k_{in} should be chosen relatively large, so that the overhead for updating C is minimized.

Together with the loop nest shown in List. 5.3 this is (nearly) the entire implementation of the algorithm. Immediately showing a benefit of the high-level approach. Though low-level considerations are required for maximizing performance, the implementation is located on a higher level of abstraction, especially compared to high-performance matrix multiplications implemented using intrinsics or even assembly. Furthermore, the optimization templates and frameworks used are not specific to the matrix multiplication algorithm, which suggests applicability beyond this specific algorithm.

Lastly, we look at the code generated by the compiler, in this instance GCC 7 with

AVX512 as the target instruction set. Listing 5.6 shows the inner loop that iterates `k` in List. 5.5. As can be seen by the generated code, the compiler implements the register-blocking scheme as intended. The `VMOVAPD` instruction loads the row-slice from B, whereas the `VBROADCASTSD` instruction broadcasts scalar values of A into a vector register. As a 4×4 blocking was chosen (by the auto-tuner), 16 FMAs are performed in total. The additional scalar instructions implement the loop header and the indexing of A and B. From the generated assembly, we can expect close-to-optimal performance, as most architectures considered can perform up to (approximately) four instructions per cycles. For this reason, the additional `VMOVAPD` and `VBROADCASTSD` instructions do not affect performance on most platforms. Counting the memory references, we see that for 16 FMAs only eight memory-related instructions are required—as expected. This highlights that the critical loop does indeed run mostly within the registers.

5.4. Further Considerations and Parameter Overview

To guarantee proper alignment of all addresses we use Boost's `aligned_allocator` template to allocate the matrices. By using this allocator and because of the divisibility constraints described in Sec. 5.1 and Sec. 5.3, all loads and stores are guaranteed to be aligned. This avoids a potential performance pitfall, as loads and stores across cache line boundaries require multiple cache lines to be accessed and thereby put more pressure on the main memory and the read and write ports of the caches.

Furthermore, our implementation assumes that the matrix dimensions divide the outer loop block sizes. We therefore pad the matrices to the next larger valid matrix size before the compute kernel is executed.

Finally, to maximize the potential performance on NUMA machines, we added a parameter that, if enabled, copies the matrices A and B to the NUMA nodes before performing the matrix multiplication. Thereby, potential performance degradation due to limited inter-socket bandwidth can be alleviated. For this measure to be effective, additionally threads get pinned to a NUMA node. For measuring runtimes, we neither add the potential copying between NUMA nodes nor the padding of the input matrices to the overall runtime, as our sole goal is achieving near-optimal floating-point throughput.

The parameters of the compute kernel are summarized in Tab. 5.1. As the table shows, there are overall 11 parameters for a challenging higher-dimensional parameter space. The chosen value ranges imply 629 million distinct parameter combinations and were chosen to accommodate architectures with different cache sizes, cache hierarchy

Listing 5.6: AVX512 assembly of the register-blocked innermost loop. As 4x4 blocking was chosen by the auto-tuner, 16 FMAs are performed per iteration. Four vector variables need to be read, additionally four scalar values get broadcasted.

```
k_loop:
vbroadcastsd zmm0,QWORD PTR [rdx]
add          rax,0x100
vmovapd      zmm4,ZMMWORD PTR [rax-0x100]
vmovapd      zmm3,ZMMWORD PTR [rax-0xc0]
vmovapd      zmm2,ZMMWORD PTR [rax-0x80]
vfmadd231pd  zmm16,zmm4,zmm0
vmovapd      zmm1,ZMMWORD PTR [rax-0x40]
vfmadd231pd  zmm18,zmm3,zmm0
vfmadd231pd  zmm17,zmm2,zmm0
vfmadd231pd  zmm19,zmm1,zmm0
vbroadcastsd zmm0,QWORD PTR [rdx+0x8]
vfmadd231pd  zmm14,zmm4,zmm0
vfmadd231pd  zmm8,zmm3,zmm0
vfmadd231pd  zmm13,zmm2,zmm0
vfmadd231pd  zmm7,zmm1,zmm0
vbroadcastsd zmm0,QWORD PTR [rdx+rsi*8]
vfmadd231pd  zmm20,zmm4,zmm0
vfmadd231pd  zmm15,zmm3,zmm0
vfmadd231pd  zmm12,zmm2,zmm0
vfmadd231pd  zmm6,zmm1,zmm0
vbroadcastsd zmm0,QWORD PTR [rdx+rcx*8]
add          rdx,0x400
cmp          rax,r9
vfmadd231pd  zmm11,zmm4,zmm0
vfmadd231pd  zmm5,zmm3,zmm0
vfmadd231pd  zmm9,zmm2,zmm0
vfmadd231pd  zmm10,zmm1,zmm0
jne          k_loop
```

	$x_{\text{reg}}, y_{\text{reg,base}}$	$x_{\text{in}}, y_{\text{in}}, k_{\text{in}}$	$x_{\text{out}}, y_{\text{out}}, k_{\text{out}}$	NUMA	schedule	SMT
min	1	8	64	{on, off}	{static, dynamic}	{4-way (Phi), 2-way, off}
max	5	128	512			
step	1	8	64			

Table 5.1.: The matrix multiplication kernel exposes eleven parameters to be tuned. The parameters are grouped according to register blocking and the inner and outer loop nests. The last three parameters control whether the matrices are copied on each NUMA node before executing the kernel, the OpenMP schedule type and the number of threads, respectively.

and vector instruction sets. As 4-way SMT is only supported by a single platform used in the evaluation, an Intel Xeon Phi processor, the related value of the SMT parameter is only used on this hardware platform.

5.5. Evaluation

In this section, we evaluate the presented auto-tuning-enabled matrix multiplication algorithm. We first describe the setup of the experiments, i.e., the hardware platforms used and the input matrices. Then, we present the achieved performance for a set of different auto-tuners and the time required for auto-tuning. To evaluate whether the parameters tuned did actually contribute to the achieved performance, we look at the contribution of the parameters for the best-performing search strategies. Finally, we look closer at the tuning process on a single hardware platform.

5.5.1. Experimental Setup

To evaluate the matrix multiplication algorithm, auto-tuning experiments were conducted on four hardware platforms. As the first platform, we used an AMD Epyc 7551P with 32 cores supporting AVX2. Though it is a single socket platform, the Epyc 7551P consists of four NUMA domains with eight cores each. Each core has four 128 bit vector units that can perform one 256 bit-wide FMA per cycle. The second platform has two Intel Xeon Gold 5120 processors with 28 cores overall; it supports AVX512 and has a single FMA-enabled vector unit. Representing the slightly older Sandy Bridge processor generation, we included a dual-socket Intel Xeon E5-2670 platform that only supports AVX(1). This processor has two 256 bit vector pipelines per core, one for additions and one for multiplications. The Xeon E5-2670 is the only processor that does not support

FMA operations. Finally, we included the Intel Xeon Phi 7210 many-core processor with its 64 Silvermont-derived cores. As mentioned in Sec. 3.4, this architecture is an especially difficult optimization target. The cores are less capable compared to the other processors in most regards, yet the vector units have the highest throughput at two 512 bit FMAs per cycle. As this description shows, only two processors implement the same vector AVX512 instruction set[3]. Thanks to Vc and AutoTuneTMP, we can nevertheless target all four processors with the same compute kernel.

To evaluate the matrix multiplication algorithm described in the previous sections, we consider square matrices with $N = 8192$. The components of all matrices were drawn uniformly from the interval $[0, 1]$. This matrix size was chosen as the matrices are too large to be cached. Furthermore, the size is large enough to allow reasonably accurate measurements of the performance—even on the 64-core Xeon Phi platform. For this matrix size, we get approximately two digits of accuracy for the measured performance. Accurate measurements are highly important for the auto-tuner to choose the correct parameterization. These measurement difficulties are primarily an effect of dynamic frequency scaling and other power saving measures.

As a baseline for auto-tuning, we use a parallelized-vectorized non-optimized (PVN) approach as described in Sec. 4.6. Thus, we require reference parameter values that mostly disable the optimizations associated with the parameters. As the PVN parameterization, we chose the minimal values for all block sizes, as shown in Tab. 5.1. We further set SMT to off, NUMA-copy to off and chose static scheduling for OpenMP. As the minimal values of the outer and inner loop blocking are 8 and 64, respectively, even the initial parameterization still allows for improved cache utilization compared to a truly naive implementation. Therefore, this PVN parameterization still underestimates the speedup enabled by auto-tuning.

To ensure correctness of the executed compute kernels, we use AutoTuneTMP's parameter value adjustment functionality. Before a parameterized compute kernel is instantiated, the configured parameter values get hierarchically adjusted to the "closest" valid combination. This is required to fulfill the divisibility constraints described in Sec. 5.1 and Sec. 5.3. The register-blocking parameters are never adjusted. However, the inner blocking parameters are adjusted so that they divide the register-blocking parameters. The outer cache-blocking parameters in turn are adjusted according to the already adjusted inner cache-blocking parameters. We furthermore supplied a test

[3]The Xeon Gold and Xeon Phi actually implement slightly different instructions sets. However, both processors implement the AVX512F sub-standard, which our kernel utilizes.

functor that validates each run by using a reference result matrix computed before the auto-tuning was started.

To compare the search strategies available in AutoTuneTMP, we performed auto-tuning using six search strategies with different properties. We consider serial-compilation and parallel-compilation search strategies. Furthermore, we use group tuners to split the search space into smaller subspaces. The group tuners were configured to iterate over their subordinate tuners three times. For the group tuners, the parameters are grouped into a register-blocking parameter group, an inner cache-blocking parameter group and an outer cache-blocking parameter group (as in Tab. 5.1). The remaining parameters are grouped into an other-parameters group.

All search strategies run until they encounter a local or global runtime minimum. As Monte Carlo does not converge to a minimum, it is stopped after 50 search steps. Apart from Monte Carlo, we considered three variants of line search: (serial-compilation) line search, parallel line search and split parallel line search. The split parallel line search uses the group tuner with parallel line search tuners as its subordinate tuners. These three search strategies implement a very similar approach and are compared to show the effect of parallel compilation and of splitting up the search space into multiple smaller search spaces. Further experiments were conducted using the parallel neighborhood search and split parallel full neighborhood search. The parallel full neighborhood search can only be used as a subordinate tuner, as the curse of dimensionality prevents its application in a 11-dimensional parameter space: there are (theoretically) up to $3^{11} - 1$ "neighbors" compared to $2 \cdot 11$ "neighbors" for the non-full neighborhood search.

To put the achieved performance in context, we provide results for two reference implementations. On the Intel platforms, we used the Intel MKL 2019 library, a BLAS implementation by Intel. As this implementation is heavily optimized by the vendor, we consider it a realistic upper bound for the achievable performance. On the Epyc platform, we use a BLAS implementation called BLIS [96], which is recommended by AMD. As both BLAS implementations do not resize the matrices, we measured the performance for $N = 8208$ to avoid reduced performance likely due to aliasing issues. Furthermore, for the MKL library Intel recommends an interleaved allocation policy on NUMA machines that distributes the pages of an allocation across the NUMA nodes. Thus, our reference performance results were computed using `numactl --interleave=0,1`. Set to four nodes, the same approach improved performance of BLIS on the AMD platform as well. As our reference results, we report the best observed performance averaged across 10 repetitions on all platforms except the Xeon Phi. The Xeon Phi platform required a

different procedure.

On the Xeon Phi a fair comparison is most difficult, as close to peak performance is not even achieved by Intel's own implementation and not achievable in principle. Furthermore, the performance of the MKL was not consistent even with 20 repetitions of the multiplication task. Therefore, we state the highest and lowest measured performance. As variations in performance are likely an effect of throttling from 1.5 GHz to $\approx$ 1.4 GHz, the lower bound is more realistic for long-running tasks.

5.5.2. Performance Results

In Fig. 5.4, we show the achieved performance of the search strategies considered. The black dotted line is the device peak performance, which is the upper bound of the achievable performance on all platforms except for the Xeon Phi. The best overall result of a device is indicated by a red-colored device name. The overall best results show that our auto-tuned high-level implementation achieves a performance nearly identical to that of Intel's MKL on the dual-socket Xeon Gold platform as well as the dual-socket Xeon E5 platform. Both Intel's MKL as well as our auto-tuned implementation achieve near-peak performance on both of these platforms. On the Xeon Gold platform, we achieved 653 GFLOPS or 91% of the peak performance using line search as the search strategy. Similarly, on the Xeon E5 platform a performance of 308 GFLOPS was measured which corresponds to 80% of the peak performance. This result was achieved with split parallel line search. The best result for the AMD Epyc platform was achieved using the same search strategy. On this platform a performance of 556 GFLOPS or 87% of the peak performance was measured. This is higher than the 501 GFLOPS we measured for the BLIS library.

The observed performance on the Xeon Phi platform was slightly lower. Line search achieved the best overall result with 1215 GFLOPS or 42% of the peak performance. While this might seem rather low, even Intel's MKL achieves only 55% to 62% of the peak performance. To explain the performance on this platform, a closer look at the hardware architecture is needed.

The Xeon Phi 7210 processor is an essentially two-wide architecture, as it can decode two instructions per cycle and has no other source for decoded instructions such as a micro-op cache or a loop buffer. As it has two vector pipelines, instructions other than arithmetic instructions replace arithmetic instruction and the performance is lowered by the fraction of non-arithmetic instructions. For example, the assembly displayed in List. 5.6 implement the innermost loop using 16 FMAs, 4 loads, 4 broadcasts and four

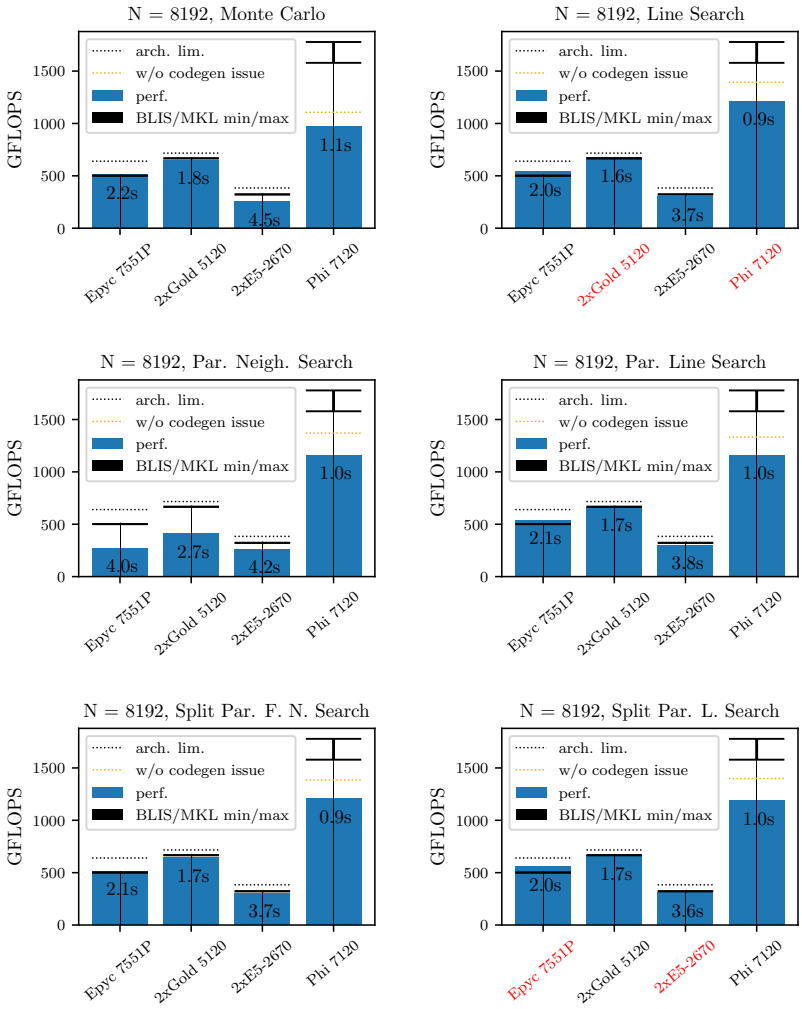


Figure 5.4.: Duration and performance of the auto-tuned kernel for multiple search strategies. The dotted lines indicate the theoretical peak performance of the hardware platforms. MKL and BLIS are used as reference implementations on the Intel and AMD platforms. Except for the Xeon Phi platform, the performance achieved by the line search variants matches that of the vendor implementation. The best overall result for each platform is emphasized by a red-colored device name.

scalar instructions (for indexing/loop header). Therefore, on the Xeon Phi platform this implementation can only achieve up to 57% of the peak performance.

The Xeon Phi implements the AVX512F instruction set which allows for embedded broadcast instructions. By embedding a broadcast instruction into an operand of an, in our case, FMA instruction, the number of instructions can be reduced. As instructions compete for decoder bandwidth, using embedded broadcasts is an important optimization on the Xeon Phi. In the assembly shown in List. 5.6 there are four broadcasts (x_{reg}-many) in the innermost loop which could be avoided. Unfortunately, neither GCC7 nor GCC9 were able to generate assembly that make use of embedded broadcasts. By eliminating the broadcasts, this loop would be limited to 67% of the peak performance. Even better, by choosing a larger 5×5 blocking scheme the theoretical performance limit further improves to 74% of the peak performance. In Fig. 5.6, the dotted orange lines extrapolate the performance of a 5×5 register-blocking scheme with embedded broadcasts from the blocking scheme detected by the auto-tuner. As the graphs show, this issue alone explains a large part of the gap between our implementation and that of Intel's MKL. We suspect that similar code generation issues explain the remaining gap. Note that all two-dimensional blocking schemes are bound by the $y_{\text{reg,base}}$-many loads to approximately 83% of the peak performance assuming no scalar instruction in the innermost loop (partially achievable through unrolling).

The other architectures (Zen, Sandy Bridge and Skylake) are not affected by this issue as all of them are at least 4-wide architectures. These architectures can execute the arithmetic vector instructions and simultaneously handle the loop overhead, loads and the broadcasts.

5.5.3. Search Strategies and Auto-Tuning

As Fig. 5.4 provides an overview of the auto-tuning results across the different tuners, we can use these results to compare the search strategies. There are two basic types of search strategies that achieved results within margin of error. Those are the line search variants and split parallel full neighborhood search. Both Monte Carlo search and parallel neighborhood search failed to get similarly close to the optimal performance. Neighborhood search seems to get stuck in a local minimum. Compared to line search, this search strategy cannot jump along an axis in the parameter space and requires a "smoother" tuning objective. On the other hand, by considering a larger neighborhood, though for fewer parameters at a time, the split parallel full neighborhood search does not seem to be affected by this issue. The strong results of split parallel full neighborhood search

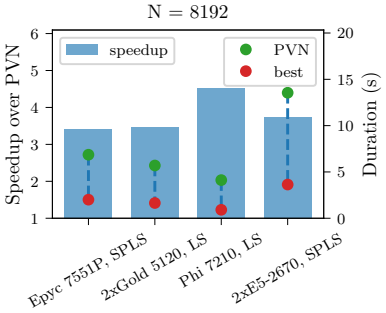


Figure 5.5.: The speedup of the best search results for each device compared to the respective performance of the PVN parameter combination.

furthermore illustrate the usefulness of the group tuner approach, as a full neighborhood search approach would be infeasible. As Monte Carlo search performed a fixed number of 50 search steps, it was at a disadvantage. For comparison, split parallel line search required 174 evaluations on the AMD Epyc platform. However, as Monte Carlo is an undirected search strategy, choosing the number of search steps appropriately can be challenging in itself.

To give an idea of the benefit of auto-tuning for this compute kernel, we compare the best detected parameterization of each device to the performance measured for the PVN parameterization. The results of this comparison are displayed in Fig. 5.5. Overall, speedups of 3.4x to 4.5x were observed. This shows that competitive performance was only achieved after auto-tuning. We emphasize that the PVN parameterization still benefited from cache blocking to a degree.

To further investigate the utility of the parameters, we performed additional experiments. Starting with the best parameterization, we set parameters to their PVN value or the closest value that results in a correct kernel. The blocking parameters were modified in groups, the remaining parameters were set to their PVN value individually. For correctness, the outer cache-blocking parameters were reset to the inner cache-blocking parameters. Similarly, the inner cache-blocking parameters were reset to the register-blocking parameters. As the resulting parameter combination is not necessarily valid, it was further adjusted using the same approach as was used during tuning. For each device, we report the speedup of the best parameterization over the partial PVN pa-

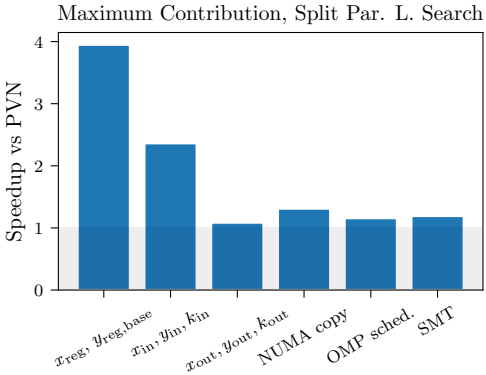


Figure 5.6.: The maximum contribution of individual parameters and parameter groups to the performance of the fastest parameterization across all devices. For this comparison, the results of split parallel line search were used.

rameterization. To assess the general usefulness of a parameter, we take the maximum speedup over all devices. As the search strategy, we chose split parallel line search, as it achieved excellent performance results across all hardware platforms.

Figure 5.6 shows the results of the partial PVN comparison. To reduce variability, 10 runs were averaged. All parameters turned out to provide at least a small benefit, with some parameters strongly improving performance. The register-blocking parameters and the inner cache-blocking parameter were the most critical parameters for high performance. NUMA-copy only significantly improved performance on the Epyc platform, the only platform with four NUMA nodes. Similarly, the OpenMP scheduling parameter improved performance only on the Xeon Phi platform, which is consistent with the fact that this platform has the highest number of cores and relatively weak per-core performance. Consequently, this platform can be expected to be most susceptible to scheduling issue. The outer cache blocking turned out to provide only a negligible performance improvement. Disabling SMT yielded a small performance increase on some platforms.

The best parameter values that were detected through auto-tuning are stated in Tab. 5.2. We can draw further conclusions from these auto-tuned parameter values. The parameters were chosen so that the inner blocking scheme can run entirely in the L1 cache on these four platforms. Similarly, the outer blocking fits into the L2 cache on

	Epyc 7551P	2xGold 5120	Phi 7210	2xE5-2670
Search Strategy	Split Par. Line S.	Line S.	Line S.	Split Par. Line S.
Duration (s)	2.0	1.6	0.9	3.6
GFLOPS (DP)	557	674	1215	308
x_{reg}	4	5	5	3
$x_{\text{reg,base}}$	3	5	5	3
x_{in}	16	110	55	120
y_{in}	12	40	40	12
k_{in}	128	64	72	72
x_{out}	256	110	55	360
y_{out}	252	200	80	516
k_{out}	128	64	72	72
NUMA copy	1	1	0	1
OMP sched.	0	1	1	1
SMT	32	56	256	16
Data (in, kB)	14	31	29	17
Data (out, kB)	347	128	53	497
F/B (out)	46 (> 3.8)	21 (> 4.3)	12 (> 7.5)	53(> 3.8)

Table 5.2.: Runtime, performance, search strategy and the parameter values of the best overall parameter combination of each device. The inner blocking parameters were chosen so that the data fits into the L1 cache of the architectures. Finally, the arithmetic intensity of the outer blocking scheme is high enough to overcome the memory bandwidth limitations.

all platforms, except for the Xeon E5-2670 where it fits into the L3 cache.

For the algorithm to be compute bound, the outer blocking scheme needs to perform enough computations on a loaded block. Of course, from the performance results we could already conclude that this is the case. The arithmetic intensity computed from the parameters (in F/B) is consistent with this observation. For this calculation, it was assumed (pessimistically) that all three submatrices get reloaded when the next multiplication of submatrices is performed.

As the next step, we consider the time required for auto-tuning. To that end, we look at the overall runtime of the tuner and the fraction required for the JIT compilation. To investigate whether the optimal parameter combination is detected early or late in the tuning process, we further provide the time required until the optimal parameter combination was encountered.

Fig. 5.7 gives an overview of the time required for tuning using the different search strategies on each device. The Monte Carlo tuner is mainly listed for completeness, as

its fixed number of search steps makes it not really comparable to the other results. Of the search strategies used, only Monte Carlo and line search did not use parallel compilation. Consequently, these tuners require a larger fraction of their runtime for compilation. Comparing line search with parallel line search, parallel compilation reduced the time required for compilation significantly on all platforms. Generally, parallel compilation was most important for the Xeon Phi platforms, due to the weak single-thread performance of the Atom-derived cores that only run at 1.4 GHz. On the Xeon Phi platform, parallel compilation reduced the runtime for line search from 3312 s to 666 s. Note that parallel compilation still could not use all cores most of the time, as only occasionally more than 10 compute kernels were to be considered in the next search step.

Considering the performance of the best parameter combination and time required for auto-tuning, split parallel full neighborhood search could be considered the best overall search strategy. The achieved performance was very close to the overall best performance across all devices and tuning was significantly faster than the other search strategies on three out of four devices.

5.5.4. A Closer Look at the Auto-Tuners

As the final step of this evaluation, we look more closely at the behavior of the auto-tuners during the tuning process. First, we consider the tuners that achieved the highest performance on each device. For these tuners, the runtimes of kernel variants considered during tuning are shown in Fig. 5.8. In all four cases, performance was strongly improved in the first steps. As both tuners are variants of line search, this corresponds to adjusting the register-blocking parameters and the inner cache-blocking parameters. On the Xeon Gold and the Xeon Phi platform, only slightly improved parameter combinations were found after 80 search steps. However, on the other two platforms significant improvements were still observed after more than a 100 search steps.

Figure 5.9 shows the Monte Carlo tuner on the Xeon Gold platform. The tuning results show a major disadvantage of this tuner compared to the other tuners. As the parameter values are randomly selected, the runtimes of kernel variants do not decrease throughout search. We compare this behavior to the split parallel full neighborhood search strategy on the same platform shown in Fig. 5.10. After expensive initial evaluations, further evaluations are comparably cheap. Faster overall tuning could likely be achieved by a few Monte Carlo steps for providing an initial guess to be used by the directed search strategies.

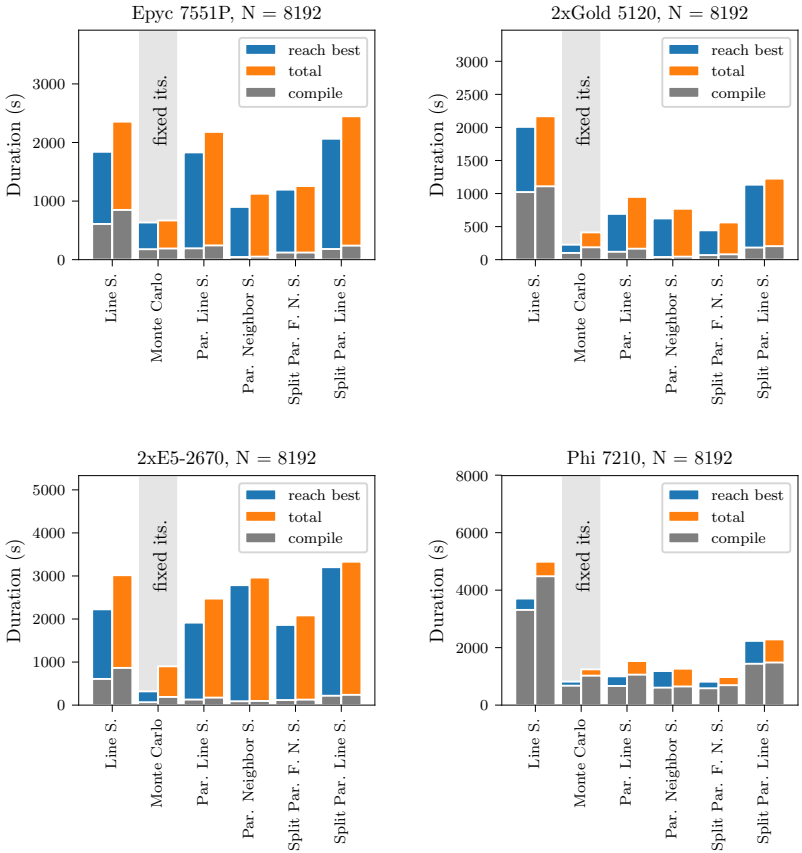


Figure 5.7.: The time required for auto-tuning the matrix multiplication kernel. We provide the total time required and the fraction used for compilation. Additionally, we state the duration until the best parameter combination was encountered. The results for the Xeon Phi platform are very different from the other platforms, as the lower single-core performance of this platform leads to more time required for (parallel) compilation. Parallel compilation is advantageous on all platforms.

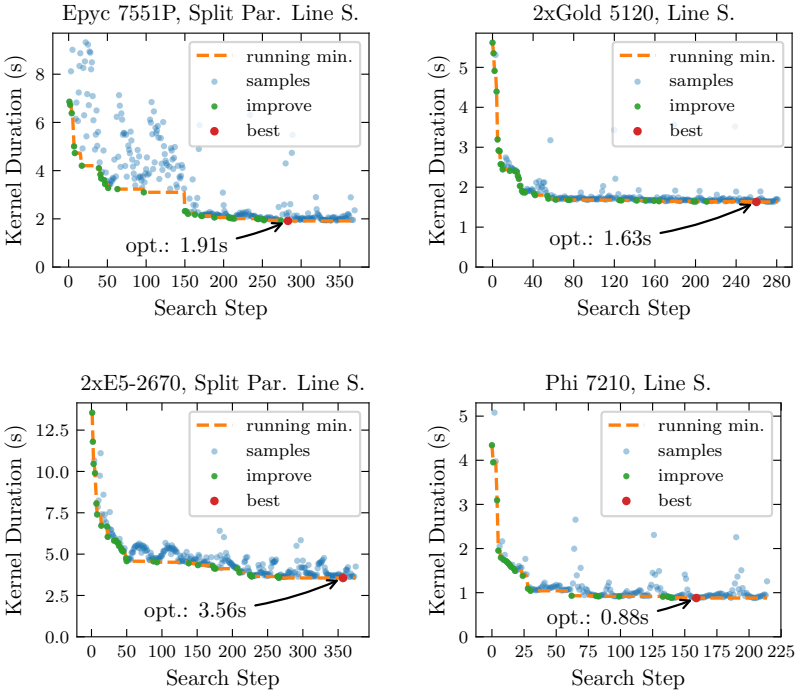


Figure 5.8.: Performance throughout the auto-tuning process for the auto-tuners that achieved the best overall performance on all devices.

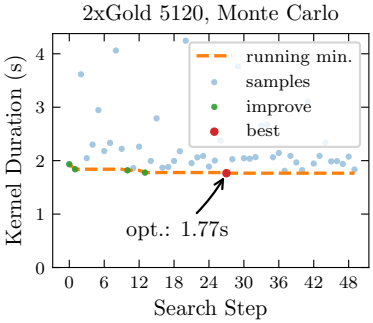


Figure 5.9.: Monte Carlo search on the dual-socket Xeon Gold 5120 platform. Throughout search relatively expensive evaluations need to be performed.

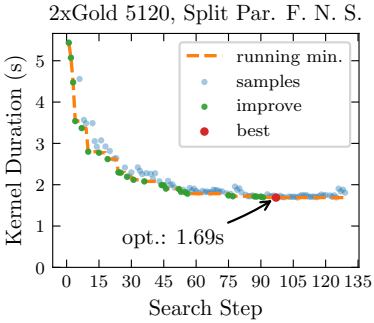


Figure 5.10.: Split parallel full neighborhood search performs few expensive evaluations after expensive evaluations in the first search steps.

Part III.

Auto-Tuned and Distributed Data Mining on Sparse Grids

In the following chapters, we investigate auto-tuning and performance portability of several sparse grid algorithms. The sparse grids method is a grid-based spatial discretization approach that is well-suited for higher-dimensional problems. While the sparse grids method has many applications, we focus on sparse grid data mining for regression in Chapter 7 and clustering problems in Chapter 8. For both types of data mining problems, this work introduces new algorithms that achieve excellent performance on a wide range of hardware platforms. These algorithms were designed for spatially-adaptive sparse grids, as spatial adaptivity allows for a reduced error or reduced runtimes—sometimes even both simultaneously.

The two regression algorithms we introduce both support auto-tuning. We show that the auto-tuning using AutoTuneTMP significantly improves performance in both cases. In the clustering chapter, the focus is on performance portability. We present node-level results as well as results for two supercomputers. All algorithm presented and evaluated in the following chapters have been implemented in the sparse grids library SG^{++}.

Contributions to Sparse Grids

In this work, we introduce a new auto-tuning-enabled variant of the streaming algorithm for sparse grid regression by Heinecke and Pflüger [74]. Our algorithm extends the basic streaming algorithm with auto-tuning capabilities and auto-tuned optimizations to improve performance and multi-GPU scalability. As a complexity-optimal approach, we further present the subspace algorithm that targets processor platforms and is auto-tuning-enabled as well. For processor platforms, the subspace algorithm strongly improves performance over the state-of-the-art streaming algorithm.

Furthermore, we introduce a high-performance sparse grid clustering algorithm that extends prior work by Peherstorfer et al [114]. Our extended algorithm has been designed for high performance, performance portability and supports a wide range of GPUs and processors efficiently. Furthermore, the results for two supercomputers presented in Sec. 8.7.3 are the first demonstration of sparse grid clustering at scale.

As an additional contribution to sparse grid data mining, we make use of a new datadriven solverless refinement approach for spatially-adaptive sparse grids called support refinement. This refinement approach significantly improves runtimes over prior surplus-based refinement strategies for a given fixed error. The new algorithms as well as the new refinement approach significantly enhance the utility of sparse grids in big data settings.

Contributions to Auto-Tuning and Performance Portability

This work introduces two new auto-tuned algorithms that are analyzed in detail. As both regression algorithms use AutoTuneTMP for auto-tuning, these applications provide further evidence for the applicability of AutoTuneTMP. Furthermore, the results show that auto-tuning significantly improves performance and therefore provide evidence for the usefulness of auto-tuning in general and AutoTuneTMP in particular. In addition to auto-tuning, performance portability is analyzed for the regression algorithms as well as the clustering algorithm. Performance portability is achieved for all three algorithms, in case of sparse grid clustering even across two supercomputers with different node-level hardware architecture.

Published Work

The following chapters extend three published papers written by the author of this work:

- "A New Subspace-Based Algorithm for Efficient Spatially Adaptive Sparse Grid Regression, Classification and Multi-evaluation" [120],
- "Heterogeneous Distributed Big Data Clustering on Sparse Grids" [119], and
- "AutoTuneTMP: Auto-Tuning in C++ With Runtime Template Metaprogramming" [118].

6. An Introduction to Sparse Grids

Grid-based discretization approaches generally suffer from the so-called curse of dimensionality, as the number of grid points required grows exponentially with increasing dimensionality of the domain [16]. The sparse grid method is a method for spatial discretization that uses a grid, but mitigates the curse of dimensionality to an extent. Sparse grids were originally introduced by Christoph Zenger for the solution of partial differential equations [164]. A thorough description of the basic theory has been given by Bungartz and Griebel [23].

The purpose of the next sections is to summarize the sparse grid theory needed for sparse grid data mining. To that end, we introduce the basic theory in Sec. 6.1, i.e., the sparse grid function space and the two types of basis functions we use. Sparse grids support spatial adaptivity for adjusting the resolution of the grid in subvolumes of the domain. We use spatial adaptivity in several data mining scenarios to achieve a given error with reduced effort. In Sec. 6.2, we first describe the concept of spatial adaptivity and then present three strategies for enabling it. Two of the presented refinement strategies are newly introduced in this work.

6.1. Sparse Grids

In order to define sparse grids, some related concepts need to be introduced beforehand. First, we define d-dimensional anisotropic grids with equidistantly-spaced grid points. Then, we introduce a hierarchical representation of these grids and describe a function space spanned by basis functions at the locations of the grid points. Sparse grids are obtained through a modification of the hierarchical representation—essentially by excluding (many) grid points.

As a first step, we define a one-dimensional grid with the interval $[0, 1]$ as domain. Given a discretization level $n \in \mathbb{N}$, we define the mesh width $h_n := 2^{-n}$. By using h_n,

we can specify the locations of the grid points

$$\Omega_n^{\text{full}} := \{ih_n : i \in \{1, \ldots, 2^n - 1\}\}. \tag{6.1}$$

Note that the grid points are equidistantly-spaced. Furthermore, this grid has $2^n - 1$ grid points and no grid points on the domain's boundary. We only use grids without boundary grid points in this work and therefore focus on this grid type. Whenever a non-zero boundary is needed, we make use of function spaces that linearly extrapolate towards the boundary. Nevertheless, we remark that the basic theory supports boundary grid points.

Next, we extend the one-dimensional grid approach to higher dimensions. As the domain, we choose the d-dimensional hypercube $\Omega := [0, 1]^d$. We require the grid to be anisotropic and therefore need a different discretization level per dimension. Thus, we use a tuple of discretization levels $\mathbf{n} := (n_1, \ldots, n_d) \in \mathbb{N}^d$. Given $\mathbf{n}$, we define a tuple of mesh widths $h_\mathbf{n} := (h_{n_1} = 2^{-n_1}, \ldots, h_{n_d} = 2^{-n_d})$ and obtain the desired anisotropic grid with grid points

$$\Omega_\mathbf{n}^{\text{full}} := \{(i_1 h_{n_1}, \ldots, i_d h_{n_d}) : \mathbf{i} \in \{1, \ldots, 2^{n_1} - 1\} \times \cdots \times \{1, \ldots, 2^{n_d} - 1\}\}. \tag{6.2}$$

This type of grid has $\prod_{j=1}^{d} (2^{n_j} - 1)$ grid points. It is fully affected by the curse of dimensionality if there is at least one $n_j > 1$ for $j \in \{1, \ldots, d\}$, as the number of grid points depends on the dimensionality exponentially in this case. We refer to this approach as the full grid approach.

The hierarchical representation of a full grid breaks up the full grid into a set of anisotropic subgrids with different discretization levels. We start by defining an index set $I_\mathbf{l}$ that enumerates the grid points on a d-dimensional anisotropic subgrid of discretization level $\mathbf{l}$:

$$I_\mathbf{l} := \{(i_1, \ldots, i_d) : 0 < i_k < 2^{l_k}, i_k \text{ odd}, k \in \{1, \ldots, d\}\}. \tag{6.3}$$

With this index set, we define the subgrids

$$\Omega_\mathbf{l} := \{\mathbf{x}_{\mathbf{l},\mathbf{i}} := (i_1 h_{l_1}, \ldots, i_d h_{l_d}) : \mathbf{i} \in I_\mathbf{l}\}, \tag{6.4}$$

with the coordinates of the grid points $\mathbf{x}_{\mathbf{l},\mathbf{i}}$. Essentially, a subgrid is an anisotropic grid as in the full grid approach, but with the even grid indices skipped.

Each grid point $\mathbf{x}_{\mathbf{l},\mathbf{i}}$ has an associated basis function $\phi_{\mathbf{l},\mathbf{i}}$. In data mining on sparse

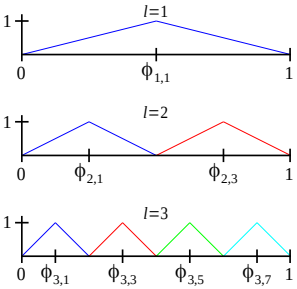


Figure 6.1.: The linear basis functions of different discretization levels for one-dimensional grids.

grids, linear, polynomial, B-spline and other types of basis functions have been used [122]. We use linear and modified-linear basis functions in this work; the latter are defined at the end of this section. Both types of basis functions have been successfully applied to data mining tasks in the past [60, 72, 73, 120, 119, 122].

The scaled and translated one-dimensional piecewise-linear basis functions, also known as hat functions, are defined as

$$\phi_{l,i}^{\text{hat}}(x) := \max(0, 1 - |2^l x - i|). \tag{6.5}$$

Although they are piecewise linear, we refer to this type of basis function as a linear basis function in the remainder of this work. The linear basis functions for one-dimensional grids are shown in Fig. 6.1 for different discretization levels.

Using a tensor-product approach, we obtain the higher-dimensional linear basis functions

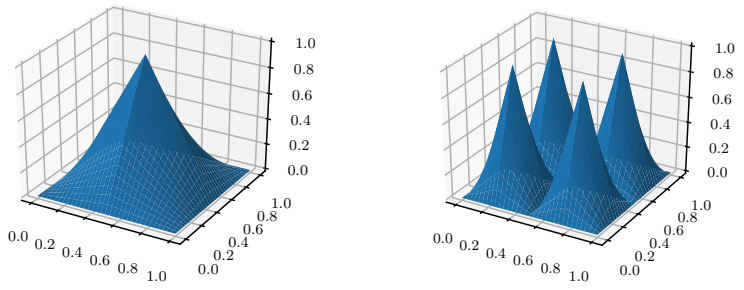
$$\phi_{\mathbf{l},\mathbf{i}}^{\text{hat}}(\mathbf{x}) := \prod_{j=1}^{d} \phi_{l_j,i_j}^{\text{hat}}(x_j). \tag{6.6}$$

Of course, because of the tensor-product approach, these basis functions are only piecewise linear if all but one dimension are held constant.

For each subgrid, we define a subspace that is spanned by basis functions $\phi_{\mathbf{l},\mathbf{i}}$ as

$$W_{\mathbf{l}} := \text{span}\{\phi_{\mathbf{l},\mathbf{i}} : \mathbf{i} \in I_{\mathbf{l}}\}. \tag{6.7}$$

To illustrate the subspaces, Fig. 6.2 shows the linear basis functions $\phi_{\mathbf{l},\mathbf{i}}^{\text{hat}}$ of the subspaces



(a) The subspace $\mathbf{l} = (1, 1)$ (b) The subspace $\mathbf{l} = (2, 2)$

Figure 6.2.: The linear basis functions of the subspaces $\mathbf{l} = (1, 1)$ and $\mathbf{l} = (2, 2)$.

(a) A 1d grid in nodal basis (b) A 1d grid in the hierarchical basis

Figure 6.3.: The hat basis functions of a 1d grid of level $l = 3$ using both the nodal and the hierarchical basis. Both representations span the same piecewise-linear function space.

$\mathbf{l} = (1, 1)$ and $\mathbf{l} = (2, 2)$.

Given the subspaces $W_\mathbf{l}$, we can define the full grid function space using the hierarchical approach as

$$V_n := \bigoplus_{|\mathbf{l}|_\infty \leq n} W_\mathbf{l}, \tag{6.8}$$

with $\bigoplus$ representing the direct sum. An example for a one-dimensional full grid with linear basis functions in both the nodal basis and the hierarchical basis is shown in Fig. 6.3. The nodal and the hierarchical representation of this $n = 3$ grid use the same grid points and span the same piecewise-linear function space. For the hierarchical representation, the three discretization levels from Fig. 6.1 are combined.

To obtain a sparse grid, we select a different set of subspaces:

$$V_n^{(1)} := \bigoplus_{|\mathbf{l}|_1 \leq n+d-1} W_\mathbf{l}. \tag{6.9}$$

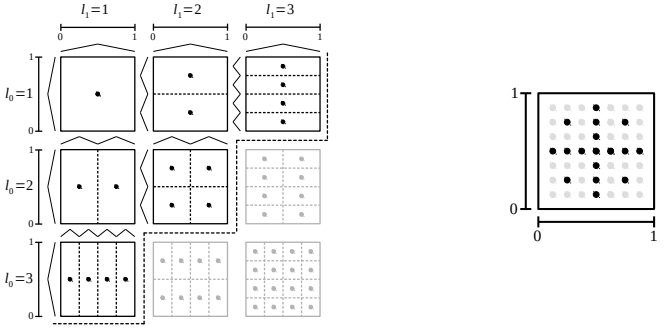


Figure 6.4.: Two-dimensional subgrids of an $l = 3$ sparse grid (left) and the $l = 3$ sparse grid itself (right). The dotted lines in the subgrid scheme outline the support of linear basis functions centered at the grid points. Grayed-out grid points and subgrids indicate the corresponding full grid.

Figure 6.4 shows the subgrids of a two-dimensional sparse grid on the left and the sparse grid obtained by superimposing the subgrids on the right-hand side. To illustrate the difference between a sparse grid (black) and a full grid, the corresponding full grid is indicated as well (gray). As Fig. 6.4 shows, the choice of subspaces of a sparse grid corresponds to a diagonal cut in the subgrid tableau. Given the definition of a sparse grid, it follows that in the one-dimensional case, sparse grids and full grids are identical. However, sparse grids employ fewer grid points for $d > 1$ and $n > 1$, i.e., for grids that are affected by the curse of dimensionality.

It has been shown that for a sufficiently smooth function u, a sparse grid interpolant $f^{(1)} \in V_n^{(1)}$ and linear basis functions, the interpolation error $||f^{(1)} - u||_{L_2}$ is slightly increased from $\mathcal{O}(h_n^2)$ to $\mathcal{O}(h_n^2(\log h_n^{-1})^{d-1})$. However, the number of grid points is significantly reduced from $\mathcal{O}(h_n^{-d})$ to $\mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1})$ [23]. Therefore, compared to a full grid, sparse grids achieve a similar error using fewer grid points.

Sparse grid functions are linear combinations of basis functions. Therefore, an individual sparse grid function $f^{(1)} \in V_n^{(1)}$ is given as

$$f^{(1)}(\mathbf{x}) = \sum_{|\mathbf{l}|_1 \leq n+d-1} \sum_{\mathbf{i} \in I_\mathbf{l}} \alpha_{\mathbf{l},\mathbf{i}} \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}). \tag{6.10}$$

Commonly, the coefficients $\boldsymbol{\alpha}$ are referred to as surpluses. Some sparse grid data mining algorithms do not exploit the hierarchical structure of the sparse grid. To simplify

the description of these algorithms, we introduce an additional notation that linearly enumerates the grid points. For a sparse grid with N grid points, we therefore define the short-hand notation

$$f^{(1)}(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}) \tag{6.11}$$

for Eq. 6.10.

For some algorithms, it is convenient to represent the grid points as a set of level-index tuples. We therefore define

$$\Omega'_n := \bigcup_{|\mathbf{l}|_1 \leq n+d-1} \{(\mathbf{l}, \mathbf{i}) : \mathbf{i} \in I_\mathbf{l}\}, \tag{6.12}$$

as the level-index representation of a sparse grid function space.

Modified-Linear Basis Functions

Modified-linear basis functions were introduced by Bungartz, Pflüger and Zimmer [24, 122]. They are standard linear basis functions inside the domain. However, the basis functions next to the domain boundary extrapolate, again linearly, towards the boundary. The one-dimensional modified-linear basis functions are defined as

$$\phi_{l,i}^{\text{mod}}(x) := \begin{cases} 1 & l = 1 \wedge i = 1, \\ \begin{cases} 2 - 2^l x & x \in [0, 2h_l] \\ 0 & \text{else} \end{cases} & l > 1 \wedge i = 1, \\ \begin{cases} 2^l x + 1 - i & x \in [1 - 2h_l, 1] \\ 0 & \text{else} \end{cases} & l > 1 \wedge i = 2^l - 1, \\ \max\left(0, 1 - \left|2^l x - i\right|\right) & \text{else.} \end{cases} \tag{6.13}$$

For the one-dimensional case, the modified-linear basis functions are depicted in Fig. 6.5 for three discretization levels.

We extend the modified-linear basis functions to higher dimensions once again using a tensor-product approach. Because modified-linear basis functions are non-zero on the boundary, they can be used as an alternative to boundary grid points. For datadriven problems, this is often a significant advantage if data points are located close to the domain boundary. Figure 6.6 shows the modified-linear basis functions of the subspaces $l = (1, 3)$ and $l = (3, 3)$.

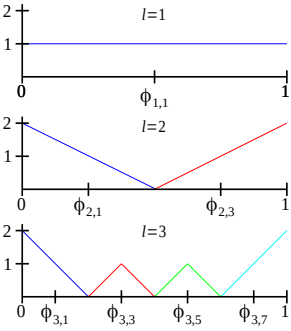
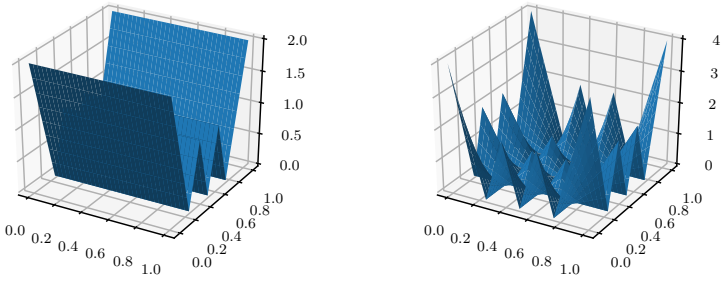


Figure 6.5.: The one-dimensional modified-linear basis functions of the subspaces $l = 1$, $l = 2$ and $l = 3$.



(a) The subspace $\mathbf{l} = (1, 3)$ (b) The subspace $\mathbf{l} = (3, 3)$

Figure 6.6.: The modified-linear basis functions of the subspaces $\mathbf{l} = (1, 3)$ and $\mathbf{l} = (3, 3)$.

These basis functions were originally introduced in the form given in Eq. 6.13. However, motivated by an algorithmic perspective, we present a second equivalent representation:

$$\phi_{l,i}^{\text{mod}}(x) = \begin{cases} 1 & l = 1 \wedge i = 1, \\ 2 \cdot \phi_{l-1,0}^{\text{hat}}(x) & l > 1 \wedge i = 1, \\ 2 \cdot \phi_{l-1,2^{l-1}}^{\text{hat}}(x) & l > 1 \wedge i = 2^l - 1, \\ \phi_{l,i}^{\text{hat}}(x) & \text{else.} \end{cases} \tag{6.14}$$

This second notation directly leads to a branch-free implementation, as we can encode the branches in the level and index parameters of the hat functions. We expand on this idea in Sec. 7.1.1.

6.2. Spatially-Adaptive Sparse Grids

We generally cannot assume that datasets are homogeneously distributed throughout the domain. As a consequence, an adjusted resolution of regions of the domain is often useful. We address this issue by using adaptive sparse grids. To differentiate adaptive sparse grid from sparse grids without adaptivity, the latter are commonly called regular sparse grids. The description of spatially-adaptive sparse grids in this section builds on work of Dirk Pflüger [122].

We define a spatially-adaptive sparse grid through the level-index representation. A spatially-adaptive sparse has the grid points $\Omega' \subseteq \Omega'_n$, for an adaptive sparse grid with an arbitrary but fixed set of grid points and a large-enough n. However, for convenience we mostly extend a given, possibly already adaptive, sparse grid by adding grid points. Algorithms that determine which grid points to add are called refinement criteria. Conversely, coarsening criteria enable the removal of grid points from the grid.

Dimensional adaptivity is a second common approach for adaptive sparse grids [60, 122]. This type of adaptivity allows for the addition (or removal) of entire subgrids. Thus, it constitutes a more coarse-grained refinement approach compared to spatial adaptivity. Through dimensional adaptivity, problems with varying resolution requirements in different dimensions can be solved more efficiently compared to regular sparse grids. However, dimensional adaptivity does not allow for an adjusted resolution in a specific region of the domain. As we target data mining applications with inhomogeneously distributed data points in this work, we employ spatial adaptivity.

We further limit ourselves to spatially-adaptive sparse grids that are consistent [60,

122, 124]. For defining this property, a sparse grid needs to be viewed as a directed graph. A parent grid point $\mathbf{x}_{\mathbf{l},\mathbf{i}}$ has two hierarchical successors (or children) in dimension j: a "left" successor with $x^{(j)}_{l_j+1,2\cdot i_j-1}$ and a "right" successor with $x^{(j)}_{l_j+1,2\cdot i_j+1}$. The other components are the same as in the parent grid point. Note that a child grid point generally has multiple parent grid points. A sparse grid is consistent if for every grid point all hierarchical parents of that grid point are themselves part of the grid. This is trivially true for the (root) grid point with $l = (1, \ldots, 1)$ and $i = (1, \ldots, 1)$, as it has no parent grid points. A grid point is a leaf node of the graph if none of its hierarchical successors are part of the grid. We employ consistent grids for two reasons. Firstly, consistency simplifies the handling of the refinement process, e.g., with respect to handling potential duplicate grid points. Secondly, the structure of a consistent grid can be exploited to enable faster algorithms, as is the case for the regression algorithm we introduce in Sec. 7.2.

So far, we have not described the refinement and coarsening criteria that we use. In the following, we present three heuristics for choosing the grid points of a spatially-adaptive sparse grid: surplus refinement, weight-support coarsening and support refinement. Both, weight-support coarsening and support refinement are newly introduced in this work.

Surplus Refinement

As a high absolute surplus value indicates a greater contribution to the overall function than smaller surplus values, this refinement approach first refines the grid points with the highest absolute surplus values [122]. That is, we infer the potential contribution of the child grid points from the contribution of the parent grid point. As for all refinement heuristics we present, examples can be constructed where this approach fails.

Surplus refinement first sorts the grid points according to the absolute value of the surpluses. Then, the $r_p \in \mathbb{N}$ grid points with the highest absolute surplus values are refined by adding all $2d$ hierarchical children of the grid point to the grid. Apart from the parameter r_p, surplus refinement has a second parameter $r_s \in \mathbb{N}$ which determines how often the surplus refinement process is repeated. Note that before each surplus refinement step, the surpluses of grid points that were added in the last refinement step need to be computed. Some applications even require all surpluses to be recomputed. As obtaining the surpluses generally requires a solution of the underlying problem (interpolation, data mining, solving a partial differential equation, ...), the refinement steps can be expensive. Surplus refinement has been applied successfully to a wide range of

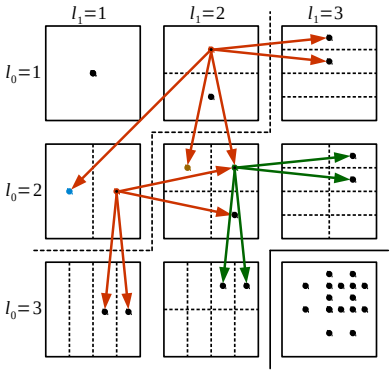


Figure 6.7.: The subgrids of a d-dimensional spatially-adaptive sparse grid that was created from an already adaptive sparse grid. Only grid points that are part of the spatially-adaptive sparse grid are shown. Three grid points were refined in two refinement steps. The resulting sparse grid is displayed in the lower right corner.

regression and classification problems [122].

Figure 6.7 illustrates the surplus refinement approach. Starting from a regular sparse grid with $l = 2$, the grid is refined two times. In the first refinement step, two grid points are refined (red). This leads to eight grid points being added to the sparse grid. The brown grid point in subgrid $(2, 2)$ additionally requires the blue grid point in subgrid $(2, 1)$ to be added to ensure consistency. During the second iteration, one grid point gets refined (green) and, consequently, four further grid points are added to the sparse grid. The spatially-adaptive sparse grid that results from the two refinement iterations is shown in the lower-right corner of the figure.

Weight-Support Coarsening

Generally, coarsening attempts to remove grid points from the grid that do not contribute significantly to the overall solution. To preserve consistency, we only allow grid points to be removed that are leaf nodes in the corresponding graph. A straightforward approach to coarsening is the removal of grid points with small absolute surplus values (which is available in SG^{++}). However, this surplus-based coarsening approach again requires the surplus values, and therefore the underlying problem to be solved.

In this work, we propose a new weight-support coarsening approach for datadriven problems. Given the level-index representation Ω' of an adaptive sparse grid, a weight-support coarsening threshold $t_{\text{ws}} \in \mathbb{R}$ and a dataset $T^{(c)} := \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$ with m data points. We define the grid points to be removed as

$$\text{WS}_{t_{\text{ws}},T^{(c)},\Omega'} := \left\{ (\mathbf{l},\mathbf{i}) \in \Omega' : \sum_{\mathbf{x}\in T^{(c)}} \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) < t_{\text{ws}} \wedge (\mathbf{l},\mathbf{i}) \text{ is leaf} \right\} \tag{6.15}$$

This criterion was designed for basis functions such as linear basis functions, where the position of the data points on the support allows inferring whether the grid point is relevant for the sparse grid function. A straightforward use-case are grid points without data points on the support. Depending on the application, such grid points might be much less important, or even entirely irrelevant, compared to grid points with many data points on the support.

For this coarsening criterion, the sums $\sum_{\mathbf{x}\in T^{(c)}} \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}_k)$ need to be computed. As we will show in Sec. 8.1, these sums are needed for the sparse grid density estimation. There, they appear as the right-hand side of the system of linear equations to be solved. We provide details on how this sum can be computed efficiently in the same section.

Support Refinement

The new support refinement approach considers the support of the basis functions and iteratively adds basis functions to the grid that have at least $t_{\text{supp}} \in \mathbb{N}$ data points on their support. The initial grid is a level $n = 1$ grid. We introduce $n_{\max} \in \mathbb{N}$ as a second parameter to control the refinement process, as the support constraint alone can lead to very large grids, especially if improperly chosen. $n_{\max}$ is the maximum allowed level as computed by the criterion $n_{\max} < |\mathbf{l}|_1 - d$ and limits the maximum depth of the iterative refinement process. Through this approach, a less deeply-refined grid is used in regions with few data points. Conversely, more grid points are spent where there are many data points, i.e., where there is potential for a more complex structure that needs to be represented.

For $t_{\text{supp}} > 1$ and if there are no duplicate data points, the support refinement approach will lead to a grid of finite size, as the volume of the support is halved whenever a hierarchical successor of a grid point is considered. Furthermore, if the supports of the hierarchical successors partition the support of the parent basis function, sparse grids created with this adaptivity approach are guaranteed to be consistent. This applies to

the linear and modified-linear basis functions we use.

Algorithm 1 shows the support refinement approach in more detail. As each refinement iteration needs to iterate the dataset for each candidate (in the `count_support` function), the support refinement is moderately expensive. To reduce the associated cost, the algorithm was implemented in OpenCL so that (multiple) GPUs can be used. By testing the support of a group of candidate grid points simultaneously, i.e., blocking of the loop that iterates the hierarchical successors, the algorithm becomes cache efficient. In our implementation, a candidate grid point is assigned to each OpenCL work-item. As a work-group jointly iterates the dataset, loaded data points get shared efficiently through the shared memory.

Algorithm 1: Support refinement creates a spatially-adaptive sparse grid for a given dataset. The `successors` function computes the set of all hierarchical successors of a set of grid points. The `count_support` function counts the number of data points on the support.

Input : Dataset $T^{(c)} := \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$, maximum level $n_{\max}$, threshold t_{supp}
Output: Spatially-adaptive sparse grid Ω' in level-index representation

if $|T^{(c)}| < t_{\text{supp}}$ **then**
  $\Omega' \leftarrow \{\}$
  return
$\Omega' \leftarrow \{(\mathbf{1}, \mathbf{1})\}$
$\Omega^{(c)} \leftarrow \{(\mathbf{1}, \mathbf{1})\}$ `// successors are candidates`
for $i = 2 \dots n_{\max}$ **do**
  $\Omega^{(c)}_{\text{next}} \leftarrow \emptyset$ `// successors are candidates in next iteration`
  for $(\mathbf{l}, \mathbf{i}) \in$ `successors`$(\Omega^{(c)})$ **do**
    if `count_support`$(T^{(c)}, \mathbf{l}, \mathbf{i}) > t_{\text{supp}}$ **then**
      $\Omega' \leftarrow \Omega' \cup \{(\mathbf{l}, \mathbf{i})\}$
      $\Omega^{(c)}_{\text{next}} \leftarrow \Omega^{(c)}_{\text{next}} \cup \{(\mathbf{l}, \mathbf{i})\}$
  $\Omega^{(c)} \leftarrow \Omega^{(c)}_{\text{next}}$

In Fig. 6.8, we show a spatially-adaptive sparse grid created through support refinement. As the figure shows, the data points are inhomogeneously distributed in the square $[0.1, 0.9] \times [0.1, 0.9]$. Through support refinement, the grid points are primarily spent in regions with a high number of data points.

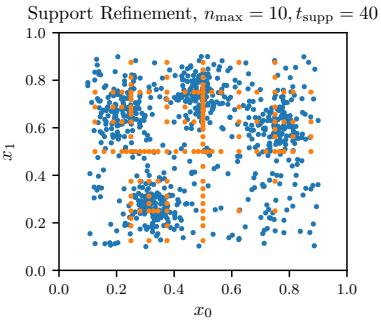


Figure 6.8.: The spatially-adaptive sparse grid obtained through support refinement for an inhomogeneously distributed $2d$ dataset.

7. Least-Squares Regression on Sparse Grids

Given a potentially noisy dataset

$$T^{(\mathrm{r})} := \{(\mathbf{x}_i, y_i) : \mathbf{x}_i \in [0,1]^d, y_i \in \mathbb{R}\}_{i=1}^m \tag{7.1}$$

that was normalized to the hypercube $[0,1]^d$ with m data points.

The tuples $(\mathbf{x}_i, y_i)$ can be interpreted as function evaluations of an unknown function $f : [0,1]^d \to \mathbb{R}$ that is to be reconstructed by a least-squares approach.

With an ansatz function space V, the least-squares regression problem is given by

$$\hat{f} := \operatorname*{arg\,min}_{u \in V} \left(\frac{1}{m} \sum_{i=1}^{m} (y_i - u(\mathbf{x}_i))^2 + \lambda \mathcal{C}(u) \right). \tag{7.2}$$

The first term minimizes the mean-squared error (MSE) and thereby ensures closeness to the data points. As only minimizing the MSE can lead to overfitting, the regularization term $\mathcal{C}(u)$ introduces a smoothness constraint. The regularization is controlled by the regularization parameter $\lambda \in \mathbb{R}$. Whereas least-squares regression has been well-known for a long time, the sparse grid regression method was introduced by Garcke et al. in 2001 [59, 60].

In this work, we select the sparse grid function space $V_n^{(1)}$ for the function space V. Furthermore, we use a weight-decay regularization approach [19, 122]. Therefore, we choose

$$\mathcal{C}(u) := \sum_{i=1}^{N} \alpha_i^2, \tag{7.3}$$

with the surpluses $\boldsymbol{\alpha}$ of the sparse grid function $u \in V_n^{(1)}$. These choices lead to the system of linear equations

$$\left(\frac{1}{m} B^T B + \lambda I \right) \alpha = \frac{1}{m} B^T \mathbf{y}, \tag{7.4}$$

with $B_{ij} = \phi_j(\mathbf{x}_i)$ [19].

As the result of $B^T B$ has the dimensionality $N \times N$, the system matrix is too large to be stored directly. To address this issue, we solve the system of linear equations with a conjugate gradient (CG) solver [136]. The CG solver requires the application of a vector to the system matrix. To avoid assembling the system matrix, we formulate the algorithm as two matrix-vector products that need to be computed in each CG iteration: $\mathbf{v} := B\boldsymbol{\alpha}$ and $\mathbf{v}' := B^T\mathbf{v}$. As B can be large as well, we use an implicit approach for storing B and recompute components of B when they are accessed. We remark that the computation of $v_i = \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}_i)$ is a sparse grid function evaluation. Thus, $B\boldsymbol{\alpha}$ is referred to as the multi-evaluation operator. Conversely, $B^T\mathbf{v}$ is called the transposed multi-evaluation operator.

For spatially-adaptive sparse grids, different algorithms have been proposed to compute both matrix-vector products. In this work, we explore three algorithms: the recursive algorithm, the streaming algorithm and the subspace algorithm. We use the recursive algorithm as a performance baseline and only provide a brief description of its multi-evaluation operator. The other two algorithms are introduced in detail. In Sec. 7.1, we present an auto-tuning-enabled variant of the streaming algorithm called unified streaming that targets GPUs as well as processor architectures. The auto-tuned subspace algorithm, which targets processor architectures, is introduced in Sec. 7.2.

Apart from the supported hardware platforms, the streaming algorithm and the subspace algorithm differ in their complexity. The streaming algorithm has a worse complexity, but it maps very well to modern hardware architectures. Conversely, the subspace algorithm has a better complexity, yet is challenging to implement efficiently even on processor architectures. In the evaluation of both algorithms, in Sec. 7.3, we investigate this trade-off by analyzing the performance of both algorithms. As both algorithms support auto-tuning, we investigate the contribution of auto-tuning to the achieved performance as well.

For other types of sparse grids, specifically regular grids and dimensionally-adaptive grids, further high-performance algorithms exist [108, 25]. These algorithms make use of the fact that in both cases the subspaces are full, i.e., contain all possible grid points. As these algorithms cannot be used in the context of spatially-adaptive sparse grids, they are not covered in this work.

The Recursive Algorithm for Multi-Evaluation Operator

The recursive algorithm makes use of the hierarchical structure of the grid to avoid the evaluation of basis functions outside of their support. It was originally proposed by Bungartz, Pflüger and Zimmer [24]. For simplicity, we consider a one-dimensional example. If $\phi_{l,i}(x) > 0$, then either $\phi_{l+1,2i-1}(x) > 0$ or $\phi_{l+1,2i+1}(x) > 0$. The case where the evaluation happens on the support's boundary of the successor grid points can be handled by arbitrarily choosing the left or right successor, as the basis functions of both successor grid points will evaluate to zero. The recursive algorithm follows the path of non-zero evaluations until the grid point of the highest-level subgrid has been processed. In case of spatially-adaptive sparse grids, a successor grid point might not be part of the grid. Due to the consistency requirement and because the supports of the successor grid points partition the support of the parent grid point, the recursion can be safely aborted if a relevant successor grid point is missing.

For dimensionality $d > 1$, grid points can have multiple parents. We use a variant of the algorithm that ensures that each child grid point is only considered once—based on the so-called unidirectional principle [122]. For m data points and a sparse grid of level n, a multi-evaluation operator that uses the recursive algorithm to evaluate individual data points has a time complexity of $\mathcal{O}(mdn^d)$ [122].

7.1. The Streaming Algorithm for Regression

The streaming algorithm for sparse grid multi-evaluation, regression and classification [74] was developed by Heinecke and Pflüger. The core idea of this algorithm is to disregard the structure of the grid and treat the grid as a set of independent level-index tuples that represent the grid points. As the algorithm evaluates every basis function for every data point, its complexity is $\mathcal{O}(Nmd)$. Note that the streaming algorithm is not linear in d, as $N \in \mathcal{O}(h_n^{-1}(\log h_n^{-1})^{d-1}) = \mathcal{O}(2^n n^{d-1})$. Thus, the complexity can be stated as $\mathcal{O}(Nmd) = \mathcal{O}(2^n n^{d-1} md)$, which is worse than that of the recursive algorithm.

Despite the worse complexity, it was shown that the streaming approach achieves higher performance than the recursive approach, as it can be implemented efficiently for modern hardware architectures. Furthermore, the complexity assumes a regular sparse grid. However, spatially-adaptive sparse grids with few grid points per subgrid mitigate the complexity disadvantage [120]. Heinecke et al. showed that the streaming approach outperforms the recursive approach with speedups of 1.23x to 14x on a Xeon X5650 processor. Speedups obtained on many other processor platforms were similar.

Furthermore, it is well-suited for modern HPC architectures including GPUs and processors with wide vector units [74, 72]. In this work, we further extend their algorithm with auto-tuning capabilities and add optimizations to better utilize modern hardware platforms.

Algorithm 2 shows the computation of $\mathbf{v} := B\boldsymbol{\alpha}$, i.e., the multi-evaluation operation. The two outer loops together iterate the components of the matrix B. To compute the components of the matrix, another iteration in the dimensionality is required (the product in Alg. 2). The evaluation of the basis functions is computationally expensive, with the exact cost depending on the type of basis function employed.

Algorithm 2: The streaming algorithm for the multi-evaluation operation $\mathbf{v} := B\boldsymbol{\alpha}$.

Input : $T^{(c)} := \{\mathbf{x}_j : \mathbf{x}_j \in [0,1]^d\}_{j=1}^m$, spatially-adaptive sparse grid in level-index representation Ω' stored in array, surpluses $\boldsymbol{\alpha}$ stored in array with same indexing as sparse grid

Output: $\mathbf{v}$ with $|\mathbf{v}| = m$ stored in array with same indexing as the dataset

```
1 for j = 1 ... m do
2     v_j ← 0;
3     for i' = 1 ... N do
4         l, i ← Ω'[i']
5         v_j += α[i'] ∏_{d=1}^{dim} φ_{l_d,i_d}(x_j^(d))
```

The streaming algorithm for the transposed multi-evaluation operation is displayed in Alg. 3. Compared to the multi-evaluation algorithm the two loops are swapped. Now, the outer loop iterates the grid points and the inner loop iterates the data points. The operations in the product that iterates the dimension are the same.

Despite the simple algorithmic structure of both operators, we will see that high-performance variants of the streaming algorithm are significantly more complex. In the following sections, we first describe our new high-performance variant, the unified streaming algorithm, and afterwards introduce a prior approach which we call the masked streaming algorithm. In the evaluation, the latter variant is used as a performance baseline for the unified streaming algorithm.

7.1.1. Unified Streaming for the Linear and Modified-Linear Basis

To efficiently evaluate linear and modified-linear basis functions, we use a new unified approach. As we will describe, with some pre- and postprocessing, the modified-linear

Algorithm 3: The streaming algorithm for the transposed multi-evaluation operation $\mathbf{v}' := B^T\mathbf{v}$

Input : $T^{(c)} := \{\mathbf{x}_j : \mathbf{x}_j \in [0,1]^d\}_{j=1}^m$, spatially-adaptive sparse grid in level-index representation Ω'_n stored in array, $\mathbf{v}$ with $|\mathbf{v}| = m$ stored in array with same indexing as the dataset

Output: $\mathbf{v}'$ with $|\mathbf{v}'| = N$ stored in array with same indexing as the sparse grid

```
for i' = 1 ... N do
    l, i ← Ω'[i']
    v'[i'] ← 0;
    for j = 1 ... m do
        v'[i'] += v[j] ∏_{d=1}^{dim} φ_{l_d,i_d}(x_j^(d))
```

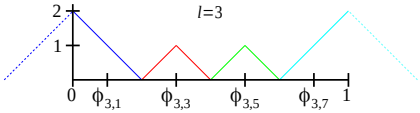


Figure 7.1.: For the modified-linear basis, the basis functions on the boundary can be treated as hat functions, as the domain is limited to $[0,1]^d$.

basis functions can be computed with the same approach as the linear basis functions. To that end, we note that the basis functions on the boundary can be treated as hat functions, as all evaluations happen inside the domain, as illustrated by Figure 7.1. Therefore, we compute most one-dimensional basis functions as hat functions. However, two issues remain: the constant basis function for $l = 1$ requires special treatment and the hat functions centered on the boundary need to be scaled.

In the definition of the hat function in Eq. 6.5, the level l is only used in the expression $h_l^{-1} = 2^l$. As the level only appears in the one-dimensional basis function evaluations, we replace it by the derived value 2^l. To handle the different cases of a basis function evaluation by a single arithmetic expression, we map l and i to l' and i' and modify the definition of the hat function to

$$\phi'_{l',i'} = \max(0, 1 - |l'x - i'|). \tag{7.5}$$

Table 7.1 shows the mapping of a level-index pairs (l, i). Note that the $l = 1$ case is handled properly by the modified hat function as $\phi'_{0,0}(x) = 1$. The $d > 1$ case is obtained by applying the same approach in each dimension.

To scale the basis d-dimensional basis functions $\phi'_{\mathbf{l}',\mathbf{i}'}$, we notice that only the boundary

Table 7.1.: Mapping of l and i of a modified-linear basis function to l' and i' of the unified evaluation approach. The values can be precomputed solely from the sparse grid.

Condition	l'	i'
$l = 1 \wedge i = 1$	0	0
$l > 1 \wedge i = 1$	2^{l-1}	0
$l > 1 \wedge i = 2^l - 1$	2^{l-1}	2^{l-1}
else	2^l	i

points need to be scaled. Therefore, we introduce the scaling factors

$$s_{l,i} := \begin{cases} 2 & l > 1 \wedge (i = 1 \vee i = 2^l - 1), \\ 1 & \text{else.} \end{cases} \tag{7.6}$$

Given the mapping in Tab. 7.1, the evaluation of a d-dimensional modified-linear basis function can be written as

$$\phi^{\text{mod}}_{\mathbf{l},\mathbf{i}} = \prod_{j=1}^{\mathrm{d}} s_{l_j,i_j} \phi'_{l'_j,i'_j} = \underbrace{\prod_{j=1}^{\mathrm{d}} s_{l_j,i_j}}_{=:s_{\mathbf{l},\mathbf{i}}} \prod_{j=1}^{\mathrm{d}} \phi'_{l'_j,i'_j} = s_{\mathbf{l},\mathbf{i}} \prod_{j=1}^{\mathrm{d}} \phi'_{l'_j,i'_j}. \tag{7.7}$$

As Eq. 7.7 shows, $s_{\mathbf{l},\mathbf{i}}$ is a single value for each d-dimensional basis function. For the whole grid, we therefore obtain a vector of scaling factors $\mathbf{s}$ with one factor $s_{\mathbf{l},\mathbf{i}}$ per grid point. Each scaling factor $s_{\mathbf{l},\mathbf{i}}$ is itself a product of one-dimensional scaling factors calculated as shown above.

As a result of the use of scaling factors, the multi-evaluation operator $\mathbf{v} := B\boldsymbol{\alpha}$ with linearly-enumerated basis functions instead of level-index notation can be stated as

$$\mathbf{v} = B'(\mathbf{s} \odot \boldsymbol{\alpha}), \tag{7.8}$$

with $B'_{ij} := \phi'_i(\mathbf{x}_j)$, the scaling factors $\mathbf{s}$ for the grid points and the componentwise multiplication $\odot$. Thus, applying the scaling factors becomes a vector-vector operation that can be computed before the more expensive matrix-vector product. Analogously, the transposed multi-evaluation operator $\mathbf{v}' := B^T\mathbf{v}$ becomes

$$\mathbf{v}' = \mathbf{s} \odot (B^T\mathbf{v}). \tag{7.9}$$

Here, the application of the scaling vector can be treated as a post-processing step.

The vector $\mathbf{s}$ itself only needs to be computed once for a sparse grid. Similarly, all required adjusted level and index values are computed once for a sparse grid. Overall, with the presented approach the use of modified-linear basis functions incurs a negligible overhead compared to linear basis functions.

Adjusting the same algorithm so that it can process linear basis functions is straightforward. We use the original level and index values, i.e., $l' = 2^l$ and $i' = i$ for all one-dimensional basis functions. The componentwise vector-vector multiplications with the scaling vector are simply omitted.

The cost of the multi-evaluation operator can be further reduced. During the computation of the basis functions, all evaluations of one-dimensional basis functions with $l_j = 1$ can be skipped due to $\phi'_{(0,0)}(x) = 1$. In the multi-evaluation case, this does not lead to issues with respect to the vectorization and significantly reduces the runtimes. This approach cannot be applied to the transposed operator, as we vectorize the algorithm over the grid points and therefore cannot easily skip one-dimensional basis function evaluations.

7.1.2. Implementing the Unified Streaming Algorithm

The unified streaming approach was implemented in OpenCL. Algorithm 4 outlines the implementation of the high-performance OpenCL multi-evaluation operator. Some technical details are not shown, for example the padding required if m is not evenly divisible in the outermost loop, or the buffer management that OpenCL requires. In the remainder of this section, we first describe the OpenCL implementation of both operators in more detail, then the multi-GPU approach and finally summarize the parameters of the compute kernels.

Work-items are generated by splitting the matrix in both dimensions. As we use OpenCL, this simultaneously addresses thread-level parallelism, vector-level parallelism and instruction-level parallelism. Each component of $\mathbf{v}$ can be computed independently. Therefore, we assign ranges of components of the result vector $\mathbf{v}$ to work-groups, i.e., we split the matrix along the rows. The size of these ranges is controlled by the parameter *local-size*, which specifies the size of the work-groups.

Because we can assign more than one result vector component to a work-item, the *data-block* parameter further modifies the size of the range processed by a work-group. By interleaving the computation of the components, this leads to consecutive instructions that are independent. Thus, instruction-level parallelism is increased. In our implemen-

Algorithm 4: The multi-evaluation operation $\mathbf{v} := B\boldsymbol{\alpha}$ of the streaming algorithm with some optimizations and the mapping to OpenCL concepts indicated.

Input : $T^{(c)} := \{(\mathbf{x}_j) : \mathbf{x}_j \in [0,1]^d\}_{j=1}^m$, spatially-adaptive sparse grid in level-index representation Ω' stored in array, surpluses $\boldsymbol{\alpha}$ stored in array with same indexing as sparse grid

Output: $\mathbf{v}$ with $|\mathbf{v}| = m$ stored in array with same indexing as the dataset

$\mathbf{v} \leftarrow \mathbf{0}$
`// Specification of OpenCL grid`
`// Split rows, parallelization`
for wg-index $= 1$; wg-index $\leq m$; wg-index += local-size $\cdot$ data-block **do**
  `// Split columns, parallelization`
  gs-range $\leftarrow \frac{N}{\text{grid-split}}$
  for gs-index $= 1$; gs-index $\leq N/$gs-range; gs-index += 1 **do**
    `// OpenCL-kernel level, implicitly vectorized`
    `// Blocking for ILP`
    for block-index = wg-index; block-index $<$ wg-index + local-size $\cdot$ data-block; block-index += data-block **do**
      for $j = 1; j \leq$ data-block; j += 1 **do**
        $v_j^{(\text{acc})} \leftarrow 0$ `// Thread-private accumulator`
      `// Iterate assigned grid point range`
      for $i =$ gs-index $\cdot$ gs-range; $i \leq$ (gs-index + 1) $\cdot$ gs-range; i += 1 **do**
        $\mathbf{l}', \mathbf{i}' \leftarrow \Omega'[i]$ `// array access, local-memory cached`
        for $j = 1; j \leq$ data-block; j += 1 **do**
          $\text{temp}_j \leftarrow \boldsymbol{\alpha}[i]$ `// array access, local-memory cached`
        `// Unrolled depending on parameter`
        for $d = 1; d \leq$ dim; d += 1 **do**
          for $j = 1; j \leq$ data-block; j += 1 **do**
            `//` $x^{(d)}_{\text{block-index}+j}$ `is stored in registers`
            $\text{temp}_j \leftarrow \text{temp}_j \cdot \phi'_{l'_d, i'_d}(x^{(d)}_{\text{block-index}+j})$
        for $j = 1; j \leq$ data-block; j += 1 **do**
          $v_j^{(\text{acc})}$ += temp_j
      for $j = 1; j \leq$ data-block; j += 1 **do**
        `// Required due to column split`
        `atomic_add`$(v_{\text{block-index}+j}, v_j^{(acc)})$

tation, we keep the level and index values in the private memory. The private memory gets mapped primarily to the L1 cache on processor architectures and to the register file on GPUs. If *data-block* is increased, the memory required per work-item increases. On GPUs, the register memory is statically partitioned according to the memory requirements of the work-groups that are currently executed. Therefore, a too high value of *data-block* can lead to fewer work-groups executed simultaneously.

To better support larger machines with many GPUs (and for smaller data sets), the matrix B is further split in the column direction. This second split of the matrix increases the number of work-groups and therefore helps to better utilize available parallel resources. The parameter *grid-split* controls into how many ranges the matrix is split in the column direction. Because the column-split approach leads to multiple work-groups updating the same result vector components, additional synchronization is required when $\mathbf{v}$ is written to. As is indicated in Alg. 4, this synchronization was realized using atomic operations. Each work-item of a work-group performs one atomic-add operation. Due to the relative infrequency of the atomic-add operations, there are only $m \cdot$ *grid-split* of them, the synchronization has no measurable impact on the overall performance.

Apart from exposing parallelism, a critical optimization for GPU-like architectures is the use of OpenCL's shared memory abstraction. As the sparse grid does not change during the execution of the multi-evaluation kernel, the $(\mathbf{l}, \mathbf{i})$-tuples can be shared across a work-group. To that end, additional work-group-level synchronization in the loop iterating the grid points is required. Our compute kernels support two parameters that have an effect on how the shared memory is used. The first parameter controls whether to use local memory at all. The second parameter, *prefetch-size*, specifies how many grid points are loaded from the global memory into the shared memory. By grouping the global memory accesses, they get combined into fewer instructions that access more memory (coalescing). As memory accesses have the granularity of a cache line, this enables achieving a higher memory bandwidth. To keep Alg. 4 more readable, the loading of the grid points into the shared memory is not shown.

In Alg. 5, we show the implementation of the transposed operator. Because of the transposition of the matrix B, splitting the matrix in the row direction is now a parallelization over the grid points instead of the data points. Analogously, the splitting in column direction creates ranges of data points. Two parameters are consequently renamed: *data-block* becomes *grid-block* and *grid-split* becomes *data-split*. Generally, column splitting is more important for the transposed operator than it is for the multi-evaluation operator. Under the (realistic) assumption that $N \ll m$, row splitting alone

would not expose enough parallel work. However, because of the large datasets targeted, adding the column splitting approach yields enough parallel work to saturate even wide hardware architectures.

Another consequence of the matrix transposition is that different data is kept in the registers and shared through local memory. For the transposed operator, each thread has a grid point assigned. Therefore, the $(\mathbf{l}, \mathbf{i})$-tuple is kept in the registers. Conversely, the data points are iterated. As a work-group jointly iterates the data set, analogously to the multi-evaluation operator, the data points can be shared efficiently through the shared memory. However, more memory is required to store the $(\mathbf{l}, \mathbf{i})$-tuple compared to a data point. Thus, the transposed operator requires more registers per thread on GPUs. This can affect performance if the dimensionality increases, as more registers translate to reduced occupancy.

For supporting regression using multiple OpenCL devices, our implementation splits the matrix a second time in row direction. As the ranges of the result vector can be computed independently, this enables a communication-free multi-device approach. The parameter *transfer-whole-dataset* of the multi-evaluation operator has an effect on how memory transfers in multi-device scenarios are managed. If the parameter is set to `true`, the whole dataset is transferred to all devices independent of the ranges that the devices process. Conversely, if the parameter is disabled, only the range processed next by a device is transferred to the device. Transferring the whole dataset might entail the transfer of data that is never accessed. However, because the data can be kept on the devices throughout runtime of the CG solver, many transfers of small ranges can be skipped. For the transposed operator, the *transfer-whole-grid* parameter allows the analogous choice.

As we implemented the unified streaming algorithm before AutoTuneTMP, it provides an example for the integration of AutoTuneTMP into an application. The "untuned" implementation was written as a class that wraps all OpenCL calls and takes care of kernel details such as padding. To integrate auto-tuning, we use the generalized compute kernel type. As described in Sec. 4.4, the use of the generalized kernel type requires two functors to be set up: the `kernel` functor and the `apply_parameters` functor. The kernel functor instantiates the "untuned" implementation and then forwards the arguments of the kernel functor to the method that calls the OpenCL kernel. During instantiation, the parameter values are read from a JSON file which was set up by the `apply_parameters` functor. Overall, the required wrapper functions and the specification of the parameters as AutoTuneTMP parameter types required less than 100 lines of C++ code per operator.

Algorithm 5: The transposed multi-evaluation operation $\mathbf{v}' := B^T\mathbf{v}$ of the streaming algorithm with some optimizations and the mapping to OpenCL concepts indicated.

Input : $T^{(c)} := \{\mathbf{x}_j : \mathbf{x}_j \in [0,1]^d\}_{j=1}^m$, spatially-adaptive sparse grid in level-index representation Ω' stored in array, $\mathbf{v}$ with $|\mathbf{v}| = m$ stored in array with same indexing as the dataset

Output: $\mathbf{v}'$ with $|\mathbf{v}'| = N$ stored in array with same indexing as the sparse grid

```
1  v ← 0
   // Specification of OpenCL grid
   // Split rows, parallelization
2  for wg-index = 1; wg-index ≤ N; wg-index += local-size · grid-block do
       // Split columns, parallelization
3      ds-range ← N / data-split
4      for ds-index = 1; ds-index ≤ m/ds-range; ds-index += 1 do
           // OpenCL-kernel level
           // Blocking for ILP
5          for block-index = wg-index; block-index <
            wg-index + local-size · grid-block; block-index += grid-block do
6              for i = 1; i ≤ grid-block; i += 1 do
7                  v_i^(acc) ← 0 // Thread-private accumulator
8                  l', i' ← Ω'[i] // Stored in registers
               // Iterate assigned data point range
9              for j = ds-index · ds-range; j ≤ (ds-index + 1) · ds-range; j += 1 do
10                 x ← T^(r)[j] // Array access, local-memory cached
11                 for i = 1; i ≤ grid-block; i += 1 do
12                     temp_i ← v[block-index + i]
                   // Unrolled depending on parameter
13                 for d = 1; d ≤ dim; d += 1 do
14                     for i = 1; i ≤ grid-block; i += 1 do
15                         temp_i ← temp_i · φ_{l'_d, i'_d}(x^(d))
16                 for i = 1; i ≤ grid-block; i += 1 do
17                     v_i^(acc) += temp_i
18             for i = 1; i ≤ grid-block; i += 1 do
                   // Required due to column split
19                 atomic_add(v'_{block-index+i}, v_i^(acc))
```

Name (/Transposed)	Description	Value Range
use-local-memory	enable local memory	{true, false}
local-size	work-group size	{64, 128, 256}
data-block/grid-block	improve ILP	{1, 2, 4, 8}
grid-split/data-split	more work-groups	{1, 2, 4, 8}
max-dim-unroll	pipeline efficiency	{1, 2, 4, 10}
prefetch-size	local memory required	{16, 32, 64, 128}
transfer-whole-dataset/transfer-whole-grid	for multi-GPU	{true, false}

Table 7.2.: Parameters of both operators of the unified streaming algorithm. All parameters are replicated for both operators. However, the data-block, grid-split and transfer-whole-dataset parameter were renamed for the transposed operator, reflecting the transposition of the matrix.

The auto-tunable parameters exposed by both operators of the unified streaming algorithm are listed in Tab. 7.2. As both operators use an independent set of parameters, most parameters are duplicated. Three parameters change their names because of the matrix transposition. The value ranges of the parameters were chosen to be plausible relative to the characteristics of the hardware platforms used in the evaluation. Based on the value ranges stated, the parameter spaces span 3072 parameter combinations each. The number of valid combinations is slightly smaller, as `local-size` needs to be greater than or equal to *prefetch-size*. That this is the case is tested through a *validate-parameters* functor.

7.1.3. Other Approaches for the Modified-Linear Basis

Modified-linear basis functions can be evaluated by simply implementing the four-way conditional as stated in Eq. 6.13. However, this approach is inefficient on modern hardware, as the conditionals lead to non-vectorized code on processors. As GPUs can dynamically mask threads at runtime that do not take a specific branch, vectorization still happens on these platforms. Still, this automatic masking leads to the evaluation of all branches that are taken by at least one SIMD element. For example, the SIMD width on Nvidia GPUs is 32, on AMD GPUs it is 16. Therefore, taking multiple branches is likely and, consequently, this approach is inefficient on GPUs as well.

To enable vectorization on processors and improve the efficiency of basis function evaluations, Heinecke et al. developed a masking approach that superimposes the branches and realizes individual branches by additional data associated with the grid points [75, 72]. Instead of only level and index, the masking approach uses four values to represent

Table 7.3.: Mapping of l and i of a modified-linear basis function to the four values needed for the evaluation using the masked evaluation approach. The values only depend on the sparse grid and can therefore be precomputed.

Condition	l^{mask}	i^{mask}	a^{mask}	o^{mask}
$l = 1 \wedge i = 1$	0	0	0x0	1
$l > 1 \wedge i = 1$	-2^l	0	0x0	2
$l > 1 \wedge i = 2^l - 1$	2^l	i	0x0	1
else	2^l	i	0x80...0	1

a grid point: l^{mask}, i^{mask}, a^{mask} and o^{mask}. The values l^{mask} and i^{mask} are adjusted values of the level l and index i of a basis function. Through the bit pattern a^{mask}, it is determined whether an absolute value is computed or not. The value o^{mask} is an offset value used for scaling the basis function. The map for computing these values depending on the four cases can be found in Tab. 7.3.

Given the mapping in Tab. 7.3, a basis function can now be computed with

$$\phi_{l,i}^{\text{mod}}(x) = \max(\text{or}_{\text{bit}}(l^{\text{mask}} \cdot x - i^{\text{mask}}, a^{\text{mask}}) + o^{\text{mask}}, 0). \tag{7.10}$$

By plugging in the mapped values of the four cases, it can be verified that the masking approach is equal to Eq. 6.13. To achieve an efficient implementation, the four values needed for the masking approach are precomputed for each grid point and used instead of l and i.

Compared to the unified approach, the masked approach requires double the data to represent a grid point. While this does not strongly affect performance on processors, it leads to worse utilization on GPUs, as more data per thread leads to fewer threads being scheduled. Furthermore, as the data required to store grid points scales linearly in the dimensionality, the unified approach allows for higher-dimensional problems to be solved before performance is noticeably affected. Additionally, the masking algorithm lacks most parameters compared to the unified streaming approach. It can, however, use the shared memory similar to the unified streaming algorithm. In the results we report, the use of the shared memory was enabled in all cases. Apart from lacking autotuning capabilities and most parameterization options, the masked algorithm shares its implementation approach with the unified streaming algorithm.

7.1.4. Performance Analysis

To assess the performance of the streaming algorithms, we count the number of floating-point operations. For linear basis functions, a single CG iteration of the unified streaming algorithm requires

$$2 \cdot \underbrace{m \cdot N}_{\text{matrix size}} \cdot \underbrace{d \cdot 6\,\mathrm{F}}_{\text{basis eval.}} \tag{7.11}$$

floating-point operations. The same number of floating-point operations is required by the masked streaming algorithm in case of both linear and modified-linear basis functions.

Similarly, for modified-linear basis functions, the transposed operator of the unified streaming algorithm requires the same number of floating-point instructions as the transposed operator for linear basis functions. However, the multi-evaluation operator skips one-dimensional basis functions with $l_j = 1$. Therefore, we define $N^{1d}_{l_j>1}$ to represent the remaining one-dimensional basis functions with $l_j > 1$ (note the implicit factor d). As a result, the multi-evaluation operator requires $m \cdot N^{1d}_{l_j>1} \cdot 6\,\mathrm{F}$ floating-point operations.

As the data related to the sparse grid and the dataset can both be efficiently cached in the shared and private memories, the arithmetic intensity is high enough for the algorithms to be entirely bound by capabilities of the floating-point units. Nevertheless, the achievable performance is below peak performance. Most modern hardware platforms support FMA instructions and peak performance is only achieved for FMA instructions. Other common floating-point instructions run at half the performance of FMA instructions, many instructions, e.g., those for transcendentals, run at even lower speed. A one-dimensional basis function evaluation consists of six arithmetic operations. As two arithmetic operations can be implemented as one FMA instruction, the compiled kernel consists of five floating-point instructions. Accounted for FMAs, the achievable performance can be approximated with $\frac{4}{5} \cdot \frac{1}{2} + \frac{1}{5} \cdot 1 = 60\%$ of the peak performance on most hardware platforms.

On AMD GPUs, computing the absolute value can be embedded in other instructions [1]. Therefore, a one-dimensional basis function evaluation only requires four instructions. Additionally, GPUs of this vendor can compute double-precision additions at the same rate as single-precision additions [4]. Most other double-precision instructions on this platform run at half the single-precision speed. Accounting for these issues, on GPUs of this vendor a single-precision bound of 62.5% and a double-precision bound of 75% is obtained.

7.2. The Subspace Algorithm for Regression

For a given data point, the streaming algorithm always evaluates all basis functions, even if the data point is outside of the support of a basis function. From a time-complexity perspective, the recursive algorithm can resolve this issue. However, Heinecke et al. showed that the recursive algorithm does not map to modern hardware platforms efficiently and is therefore overall significantly slower [72]. The subspace algorithm for regression has the same complexity as the recursive algorithm, as it evaluates at most one grid point per subspace. Additionally, it was designed so that it can be parallelized and vectorized efficiently. It, therefore, combines the advantages of both other algorithms. Due to more complex synchronization, the subspace algorithm so far could only be implemented competitively for processors. We remark on the state of an experimental GPU implementation in Sec. 7.2.7.

In its basic form, the multi-evaluation operator of the subspace algorithm is shown in Alg. 6. For each data point $\mathbf{x}_j$, the algorithm iterates the subspaces. On each subspace, the index $\mathbf{i}$ of the grid point to evaluate can be computed from the data point $\mathbf{x}_j$ and the current level $\mathbf{l}$. The subspace algorithm stores the surpluses in a d-dimensional array per subspace. For accessing the surplus of the current grid point, a linear index is calculated from the multi-index $\mathbf{i}$ (`flat_index`). As the sparse grid can be spatially-adaptive, a computed grid point might not be part of the sparse grid. To detect missing grid points, their surpluses are set to NaN values in the surplus array. If the surplus is not a NaN value and therefore the corresponding grid point is part of the grid, the basis function $\phi_{\mathbf{l},\mathbf{i}}$ gets evaluated and the result vector $\mathbf{v}$ is updated. The algorithm is compatible with a wide range of basis functions, we use it with linear and modified-linear basis functions.

An important aspect of the algorithm is an efficient computation of the index value i_k in dimension k. A grid point has the location $x^{(k)}_{l_k,i_k} = h_{l_k} i_k$ and a data point always has support on the basis function of the closest grid point or no support at all, at least for linear and modified-linear basis functions. Therefore, by solving for i_k, plugging in the data point $x^{(k)}$ and rounding, the index i_k of the nearest grid point can be computed. If $x^{(k)} \in [h_{l_k} i_k, h_{l_k}(i_k+1)[$, i.e., to the right of the searched-for grid point, computing $\lfloor 2^{l_k} x^{(k)} \rfloor$ immediately gives the index i_k in dimension k. In case of $x^{(k)} \in [h_{l_k}(i_k-1), h_{l_k} i_k[$, the value i_k-1 gets computed. With a conditional addition the correct index is calculated in both cases, as is shown in Alg. 6. Overall, the d-dimensional index $\mathbf{i}$ can be computed in $5d$ arithmetic instructions. Note that the rounding approach does not work correctly for $x^{(k)} = 1$. Therefore, the domain was restricted to $[0,1[^d$ for this

Algorithm 6: The basic subspace algorithm for the multi-evaluation operation $\mathbf{v} := B\boldsymbol{\alpha}$.

Input : $T^{(c)} := \{(\mathbf{x}_j) : \mathbf{x}_j \in [0,1]^d\}_{j=1}^m$, a spatially-adaptive sparse grid with subgrids $L \subset \mathbb{N}^d$, surpluses $\boldsymbol{\alpha}_\mathbf{l}$ for each subgrid $\mathbf{l} \in L$, subgrids stored as arrays with NaN-values to signal missing grid points

Output: $\mathbf{v}$

```
1  def calculate_index(l, x):
2      for k ∈ {1, ..., d} do
           // cond. add: bitXOR returns one if ⌊2^{l_k} x^k⌋ is even
3          i_k ← ⌊2^{l_k} x^k⌋ + bitXOR(bitAND(1, ⌊2^{l_k} x^k⌋), 1)
4      return i
5  def flat_index(l, i):
       // divisions: even indices are used in the surplus array
6      i_flat ← i_1 / 2
7      for k ∈ {2, ..., d} do
8          i_flat ← i_flat 2^{l_k − 1} + i_k / 2
9      return i_flat
10 for j ∈ {1, ..., m} do
11     v_j ← 0
12     for l ∈ L do
13         i ← calculate_index(l, x_j)
14         i_flat ← flat_index(l, i)
15         α_{l,i} ← fetch_surplus(α_l, i_flat) // array access
16         if ¬isNaN(α_{l,i}) then
17             v_j += α_{l,i} φ_{l,i}(x_j)
```

algorithm. Still, our implementation accepts dataset with the larger domain, but sets data points with $x^{(k)} = 1$ to the largest floating-point value < 1. Given this approach, the restricted domain should not be of practical relevance.

Another moderately-expensive step is the calculation of the flat index for $\mathbf{i}$. The flat index is used to access the d-dimensional array that contains the surpluses of the subspace. Algorithm 6 shows the Horner scheme we employ to compute the flat index. In our highly-optimized implementation, the index calculation, the index flattening and the one-dimensional basis function evaluation are combined. The divisions and exponentiations are implemented as integer bitshifts for improved efficiency. Flooring is implemented as a double-to-integer conversion.

The basic subspace algorithm has a complexity of $\mathcal{O}(mdn^d)$, as it evaluates one d-dimensional basis function per subspace and data point—just like the recursive algorithm. In Sec. 7.2.3, we present an approach for partially reclaiming the factor d.

For the subspace algorithm to be competitive with the streaming approach, further measures were necessary. Firstly, we present subspace skipping to not consider subspaces which cannot contribute to the solution. Secondly, we explain our parallelization and vectorization approach to better exploit hardware capabilities. Thirdly, an approach for avoiding computations is presented that makes use of the order in which subspaces are processed. Finally, we describe the transposed multi-evaluation operator, which uses the same approach as the multi-evaluation operator—with some adjustments.

7.2.1. Subspace Skipping and Blocking

For a consistent sparse grid, we know from the definition of a consistent grid that the parent grid points are part of the sparse grids. That implies that if a parent grid point does not exist, its (recursive) child grid points cannot exist either. From an algorithmic standpoint, we would therefore like to track which grid points did not exist and mark all subspaces where child grid points would be evaluated so that they can be disregarded. However, the time required to keep track of the subspaces to skip should not exceed the amount of work saved by avoiding evaluations.

An approach for skipping subspaces at a low-enough cost is obtained by ordering the subspaces lexicographically by level. By building an algorithm that iterates the subspaces always in that order, subspaces can be skipped along the order of iteration. Analogously to the hierarchical relationship between grid points, we define a parent-child

relation between subspaces with levels $\mathbf{l}'$ and $\mathbf{l}$:

$$\mathbf{l}' \text{ is a child subspace of the parent } \mathbf{l} \iff \forall k \in \{1 \dots d\} : l'_k \geq l_k. \tag{7.12}$$

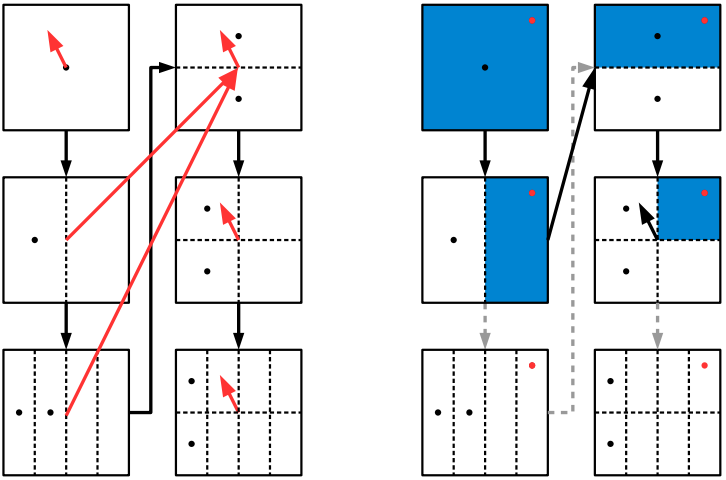
We notice that due to the consistency criterion, it holds that if a required grid point did not exist on the parent subspace, all of the child subspaces can be skipped. As the skipping criterion, we jump to the next subspace in the order of iteration that is not a child subspace of the current subspace. Given that criterion, the skip targets can be annotated to the subspace, as they are now a property of the grid. Therefore, the skip targets can be calculated in a precomputation step, which is algorithmically cheap as it only requires an iteration through the subspaces.

In Fig. 7.2, we show the skip targets of a two-dimensional spatially-adaptive sparse grid and the subspaces evaluated for a specific evaluation point. The black arrows in Fig. 7.2a indicate the order in which the subspaces are iterated whereas the red arrows point to the skip target. Red arrows that do not point to a subspace indicate that the evaluation is finished if a grid point is missing on such a subspace. The grid used in the example has three opportunities to skip subspaces that save at least one basis function evaluation. Figure 7.2b shows an evaluation of a sparse grid function at the location indicated by the red dot. The black arrows connect the subspaces that are actually iterated. As the figure shows, two subspaces can be skipped.

The subspace algorithm with subspace skipping is displayed in Alg. 7. Compared to Alg. 6, this algorithm iterates the dataset in chunks of data points. Each chunk iterates the subspaces. To enable subspace skipping, the indices of data points that do not evaluate a basis function on the current subspace are filtered. The variable `Valid` contains the remaining indices of data points that evaluate on the current subspace. To implement this filtering approach, each data point is associated with a variable next_j that holds the linear index (in the order of iteration) of the next subspace to evaluate. After the basis function evaluation, next_j is either incremented or updated with the skip target of the current subspace.

Whether to enable subspace skipping is controlled by another auto-tuned parameter of the algorithm called *enable-subspace-skipping*. Note that subspace skipping is only advantageous for spatially-adaptive sparse grids and yields no benefits in the regular case. Furthermore, due to the overhead that subspace skipping introduces, it generally is more beneficial for deeply-refined sparse grids.

Iterating the dataset in chunks improves performance. As it improves performance



(a) Target subspaces for subspace skipping (red arrows) for each subspace.

(b) Example evaluation of a data point (red dot) with two subspaces skipped.

Figure 7.2.: A two-dimensional spatially-adaptive sparse grid with the black and gray arrows highlighting the order in which the subspaces are processed by the subspace algorithm. This sparse grid offers three opportunities to avoid unnecessary work. Figure 7.2a shows to which subspace a skip leads. In Fig. 7.2b, the sparse grid is evaluated at a data point (red dot), leading to two skips. Because of subspace skipping, only four instead of six basis functions get evaluated.

Algorithm 7: The multi-evaluation operator $\mathbf{v} := B\boldsymbol{\alpha}$ of the subspace algorithm with subspace skipping, parallelization and vectorization outlined. All input and output data structures are assumed to be padded.

Input : $T^{(c)} := \{\mathbf{x}_j : \mathbf{x}_j \in [0, 1[^d\}_{j=1}^m$, a spatially-adaptive sparse grid with subgrids $L \subset \mathbb{N}^d$, surpluses $\boldsymbol{\alpha}_\mathbf{l}$ for each subgrid $\mathbf{l} \in L$, subgrids as arrays with NaN-values to signal missing grid points

Output: $\mathbf{v}$

```
1  v ← 0
2  next ← 1 // track next subspace to evaluate, per data point
   // parallelized with OpenMP
3  for j := 1; j < m; j += chunk-size do
       // iterate the spatially-adaptive sparse grid
4      for k ∈ {1, ..., |L|} do
           // filter data point indices (for skipping)
5          Valid ← {j' | j' ∈ {j, ..., j + chunk-size − 1} ∧ next_j' = k}
6          l ← L_k
           // loop gets vectorized
7          for o ∈ Valid do
8              i ←calculate_index(l, x_o) // O(d) operations, vectorized
9              e ←evaluate_basis(l, x_o) // O(d) operations, vectorized
10             i_flat ←flat_index(l, i) // O(d) operations, vectorized
               // remainder does not get vectorized; scalar operations
11             α_l,i ←fetch_surplus(α_l, i_flat)
12             if ¬isNaN(α_l,i) then
13                 v_o += e · α_l,i
14                 next_o += 1
15             else
16                 next_o ←skip_target(l) // precomputed
```

by improving data locality, it constitutes a cache-blocking approach. By evaluating many data points per subspace, the associated surplus array is accessed many times. This enables reusing data already loaded into the cache. For very small subspaces, the whole subspace might get loaded into the cache—with repeated accesses to the surpluses. Large subspaces can be processed faster as well, as they tend to be only sparsely populated. Due to the filtering step, remaining surplus accesses can coincide on the same grid points. The *chunk-size* parameter that controls the size of the chunks needs to be chosen carefully. Larger values improve data locality. However, if too large a value was chosen, the associated data might no longer fit into the L1 cache.

7.2.2. Parallelization and Vectorization

In order to fully utilize modern hardware platforms, the subspace algorithm needs to be parallelized and vectorized. Parallelization is straightforward, as we can use the same approach used for the streaming algorithm, namely parallelization over the data points. Due to the blocking approach described in the previous section, we parallelized the algorithm by assigning chunks to threads. Algorithm 7 indicates this outermost loop parallelization approach.

Vectorization is slightly more complicated, as it interacts with the subspace-skipping approach. In principle, we would like to vectorize all work associated to a chunk. However, due to the filtering step, vectorization cannot be implemented on the chunks directly. Instead, the filtered indices are processed with vector instructions. In the inner loop, which iterates `Valid` in Algorithm 7 (Line 7), the most expensive steps are well-suited for vectorization. Therefore, the calculation of the d-dimensional index, the evaluation of the associated basis functions and the computation of the linear indices for accessing the surplus array were vectorized.

Two scalar steps remain. After the filter step, the remaining data points are not in contiguous memory. Therefore, to enable efficient vectorization the remaining data points get copied into contiguous memory. Due to the fixed, small size of the chunks, the gather step takes place within the cache which reduces its cost. Secondly, the `isNaN` testing as well as the updating of v_0 and next_o are performed with scalar instructions due to the unpredictability of the branch. While vectorization cannot result in a perfect speedup due to the remaining scalar fraction, the vectorized functions each perform $\mathcal{O}(d)$ operations. Therefore, most instructions in the inner loop get vectorized.

Vectorization benefits from high values of the *chunk-size* parameter. If too few data points evaluate on a subspace to fully populate a vector, a padding approach is used.

Therefore, higher values of *chunk-size* increase the probability that enough work is available for the vectorization to be efficient. An additional parameter, *unroll-vectorization*, controls unrolling of the vectorized loop. If set to `true`, two interleaved vectorized iterations of the vectorized loop are performed simultaneously. This can improve pipeline utilization, as more independent instructions are available for scheduling. Finally, the parameter *vector-padding* is needed to ensure that the data structures are padded correctly. It pads to four components if unrolling is disabled and to eight components if unrolling is enabled.

A final aspect of the vectorization approach is the handling of the different types of basis functions. For linear basis function, we can vectorize directly by replacing the instructions with vector instructions. Modified-linear basis functions have to be handled less efficiently. As the index i is computed and not loaded, the precomputation scheme of the unified streaming algorithm cannot be used. However, as the algorithm is less limited by the performance of the arithmetic units, we use a scalar implementation with four branches that directly implements the mathematical definition from Eq. 6.13. We show in the evaluation that this approach is only slightly more costly than evaluating linear basis functions.

7.2.3. Reusing Intermediate Results

The subspace iteration scheme described in Sec. 7.2.1 enables another type of optimization. When moving from one subspace to the next, some computed values can be reused. For example, we consider an evaluation first on the $\mathbf{l} = (1, 1, 1, 1)$ subspace and then proceed with the $\mathbf{l}' = (1, 1, 1, 2)$ subspace. We further assume the indices of the basis functions to be $\mathbf{i} = (1, 1, 1, 1)$ and $\mathbf{i}' = (1, 1, 1, 3)$, respectively. For a given data point $\mathbf{x}$, the basis function evaluations become

$$\phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) = \phi_{1,1}(x^{(1)})\phi_{1,1}(x^{(2)})\phi_{1,1}(x^{(3)})\phi_{1,1}(x^{(4)}) \tag{7.13}$$

and

$$\phi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) = \phi_{1,1}(x^{(1)})\phi_{1,1}(x^{(2)})\phi_{1,1}(x^{(3)})\phi_{1,3}(x^{(4)}). \tag{7.14}$$

Therefore, in this example three terms are recomputed redundantly.

This observation is generalized by annotating the subspaces with a number that describes which dimension changed from the last subspace in the order of iteration, i.e.,

lexicographically sorted by level. For the example above, we would annotate the value four. Reusing these values is implemented by storing all products of one-dimensional basis function evaluations up to a certain dimension, for dimension d' this approach therefore computes the expression

$$\prod_{k=1}^{d'} \phi_{\mathbf{l},\mathbf{i}}(x^{(k)}). \tag{7.15}$$

As these intermediate results get computed anyway, this approach does not introduce additional arithmetic work. Of course, additional storage space is required. In case of the basis function evaluations, d entries per element in a chunk are needed.

The same ideas as for the basis functions can be applied analogously to the calculation of the index $\mathbf{i}$ and for the computation of the linear index i_{flat}. Consequently, the algorithm avoids many redundant one-dimensional computations when iterating the subspaces. Whether partial results are reused is controlled by the Boolean parameter *reuse-intermediates*.

7.2.4. Transposed Multi-Evaluation

For the transposed multi-evaluation operator $\mathbf{v}' := B^T\mathbf{v}$, we choose a similar approach as for the multi-evaluation operator. To obtain an efficient algorithm, we again want to exploit the grid structure. However, due to the transposition of the matrix, the grid structure is now located in the column direction instead of the row direction. Therefore, our algorithm for the transposed operator iterates the rows.

Figure 7.3 illustrates the approach we propose for a regular two-dimensional $l = 2$ sparse grid and a dataset with five data points. In the figure, the rows are sorted according to the subspaces. Because of this, we can observe an evaluation-like structure in the individual columns. As the grid points partition the support, in the index range of a subspace one evaluation per column needs to be performed (blue). We furthermore see that if we traverse a column, the same component from the vector $\mathbf{v}$ is multiplied to the matrix component. For each subspace, we can compute the index of the grid point to evaluate. However, we then need to update the corresponding result component in the vector $\mathbf{v}'$. As there is a result component for each grid point, we again use the d-dimensional subspace arrays. Before, the subspace arrays were used to efficiently locate the surplus values $\alpha_{\mathbf{l},\mathbf{i}}$. We now use the same data structure to update the correct component of $\mathbf{v}'$.

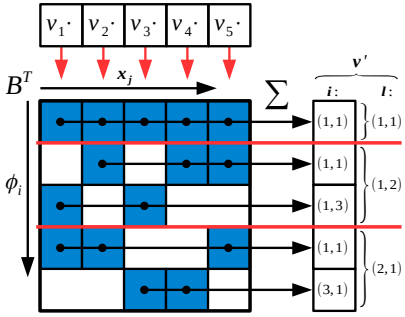


Figure 7.3.: The transposed multi-evaluation operator for a 2d regular sparse grid of level 2 and a dataset with 5 data points. By sorting the grid points according to their subspace, ranges of rows of the matrix correspond to evaluations on the same subspace (delimited by the red lines). Per column and subspace range, there is one non-zero evaluation (blue). Each column has one value v_j that is multiplied to the basis function evaluation at the component. As the algorithm operates column-wise, $\mathbf{v}'$ is stored in per-subspace arrays.

The transposed operator in its high-performance form is outlined in Alg. 8. To highlight the hierarchical storage, the result values are denoted by $v'_{\mathbf{l},\mathbf{i}}$ and the arrays by $\mathbf{v}'_{\mathbf{l}}$. Due to the parallelization over the data points and because we employ the same optimizations as for the multi-evaluation operator, the resulting algorithm is very similar to the multi-evaluation operator at first glance. There are, however, two notable differences. As we use d-dimensional arrays to store the results during the computation, the $v'_{\mathbf{l},\mathbf{i}}$ that correspond to a grid point get initialized to zero in the beginning. Potential other array entries get once again set to NaN. After the computation, the final results are extracted from the arrays and $\mathbf{v}'$ gets stored in a flat array. Furthermore, because of the column-parallelization approach, synchronization is needed to avoid race conditions whenever $v'_{\mathbf{l},\mathbf{i}}$ values are updated. We use a mutex to lock the subspace currently processed by a thread. This is realized in Alg. 8 by the calls to `lockSubspace` and `unlockSubspace`. Still, the first few subspaces in iteration order cannot be handled at optimal performance. Synchronization is especially problematic for the very first subspace with level $\mathbf{1}$, as all threads start with this subspace and want to update the same result value $v'_{\mathbf{1},\mathbf{1}}$ (see Fig. 7.3). After a few subspaces where some threads needed to wait until they could acquire a lock, the threads are spread-out enough for synchronization to become cheap. However, compared to the multi-evaluation operator a slightly lower

performance is expected.

Algorithm 8: Transposed operator $\mathbf{v}' := B^T\mathbf{v}$ of the subspace algorithm. All input and output data structures are assumed to be padded.

Input : $T^{(c)} := \{\mathbf{x}_j : \mathbf{x}_j \in [0,1[^d\}_{j=1}^m$, a spatially-adaptive sparse grid with subgrids $L \subset \mathbb{N}^d$ and in level-index notation Ω', multi-evaluation result $\mathbf{v}$

Output: $\mathbf{v}'$

```
1  zero_arrays(Ω', {v'_l : l ∈ L}) // zero existing grid points
2  next ← 1 // track next subspace to evaluate, per data point
   // parallelized with OpenMP
3  for j := 1; j < m; j += chunk-size do
       // iterate the spatially-adaptive sparse grid
4      for k ∈ {1, ..., |L|} do
           // filter indices, others skip current subspace
5          Valid ← {j' | j' ∈ {j, ..., j + chunk-size − 1} ∧ next_j' = k}
6          l ← L_k
7          lock_subspace()
           // loop gets vectorized
8          for o ∈ Valid do
9              i ←calculate_index(l, x_o) // O(d) operations, vectorized
10             e ←evaluate_basis(l, x_o) // O(d) operations, vectorized
11             i_flat ←flat_index(l, i) // O(d) operations, vectorized
               // remainder does not get vectorized; scalar operations
12             v'_{l,i} ←fetch_surplus(v'_l, i_flat)
13             if ¬isNaN(v'_{l,i}) then
14                 α_{l,i} += e · v_o
15                 next_o += 1
16             else
17                 next_o ←skip_target(l) // precomputed
18         unlock_subspace()
   // extract values for existing grid point into vector
19 v' ←extract_from_arrays(Ω', {v'_l : l ∈ L})
```

7.2.5. Memory Requirements

A potential disadvantage of the subspace algorithm compared to both the recursive and the streaming algorithm is the amount of memory required. The subspace algorithm stores each subspace as a d-dimensional array with $2^{|\mathbf{l}|_1-d}$ elements. This is not a dis-

advantage for regular sparse grids. For this type of grid, the array structure is actually more efficient in storing surpluses than a level-index storage, as a grid point is stored using a single floating-point variable and the level only needs to be stored on a per-subspace basis. However, in case of spatial adaptivity very deeply refined grids can result in high memory requirements. For example, starting with an $n = 1$ regular grid and performing 20 surplus refinement steps, it is possible to reach a subspace $\mathbf{l}$ with $|\mathbf{l}|_1 = 21$ and therefore up to 2^{21-d} grid points. For even more deeply-refined sparse grids, memory requirements can become problematic.

To mitigate this issue, our implementation avoids storing memory-intensive subspaces as arrays and instead stores the grid points as a vector of indices. The vector of indices of a subspace is unpacked into an array when the subspace is processed by a chunk of data points. Therefore, a single array is required that is large enough to represent the subspace $\max_{\mathbf{l}\in L} 2^{|\mathbf{l}|_1-d}$, i.e., the largest subspace. Because of the parallelization, our implementation uses one such array per thread. Whether to store a subspace as an index list is controlled by how well the subspace is utilized. To assess the utilization, we compute the ratio of the grid points on the subspace and the number of grid points the full subspace would contain. The parameter *list-ratio* controls how subspaces are represented. If the utilization is below the value of the *list-ratio* parameter, the subspace is stored as an index list.

The subspace algorithm only is less memory efficient compared to other algorithms if large subspaces exist that are sparsely populated. However, these subspaces can be efficiently stored in lists of indices as described above. Assuming a most deeply-refined subspace $\mathbf{l}_{\max}$ with $|\mathbf{l}_{\max}|_1 - d \leq 30$, memory consumption per thread for the one large array is limited to feasible 8 GB. We have not observed high memory utilization in any of our experiments.

7.2.6. Implementation and Parameter Overview

In the implementation, the code was parallelized with OpenMP. For vectorization, AVX2 vector intrinsics were used. Both the multi-evaluation and the transposed multi-evaluation kernel were auto-tuned with AutoTuneTMP and its CPPJIT pipeline. Note that our implementation predates AutoTuneTMP's optimization template collection. Thus, parameterization of the two compute kernels was implemented using conditional compilation controlled by C preprocessor variables.

The subspace algorithm is not limited by either the floating-point throughput or memory bandwidth. In measurements on an Intel i7 6700K with four Skylake cores, the

Name	Description	Value Range
chunk-size	data point blocking	$\{16, 32, 64, 128, 256, 512\}$
enable-subspace-skipping	skip subspace	{true, false}
reuse-intermediates	avoid redundant 1d operations	{true, false}
unroll-vectorization	unroll vectorized loop	{true, false}
vector-padding	8 required if unrolling	$\{4, 8\}$
list-ratio	limit memory requirements	$\{0.1, 0.2, 0.3\}$

Table 7.4.: Parameters and their values for the subspace algorithm. The same parameters are replicated for both the multi-evaluation and the transposed multi-evaluation operator.

subspace algorithm achieved 2.4 instructions per cycle and core. This includes vector as well as scalar instructions. Overall, this shows that the algorithm efficiently uses the available execution resources. On the instruction level, the most expensive instructions access the surplus arrays. However, memory bandwidth requirements of the algorithm are generally low due to the effective blocking approach. Memory latency is a minor bottleneck, as is expected due to the quasi-random memory access patterns of the algorithm. Mispredicted branches incur a significant performance penalty. Unfortunately, the most problematic branches test whether a retrieved surplus is NaN and are, as mentioned, unpredictable. Because of this analysis, we provide results for this algorithm from a runtime perspective and not for other performance metrics.

In Tab. 7.4, an overview of the parameters and their value ranges is given. We chose fixed sets of values for all parameters. Of course, the Boolean parameters have two valid values. The vector padding parameter only has two valid choices as well, as it needs to be set depending on the value of the loop unrolling parameter. The *chunk-size* parameter has a fairly wide value range and only needs to divide the effective SIMD width, i.e., four or eight. If the subspace utilization falls below the value of the *list-ratio* parameter, the grid points of this subspace are stored in index-vector format. As we tune for performance and not for minimizing memory use in our auto-tuning experiments, it should not have an influence on the auto-tuning results. Given these parameter ranges and constraints, there are 144 possible parameter combinations. The dependency between the *unroll-vectorization* parameter and the *vector-padding* parameter was implemented through an adjustment functor. Thereby, only valid parameter value combinations are considered.

7.2.7. The Subspace Algorithm on GPU?

A proof-of-concept variant of the subspace algorithm for Nvidia GPUs has been created in joined work with Maximilian Luz [100]. It can perform subspace skipping and has a thread-level blocking approach, but lacks the caching of intermediate values. The intermediate values need to be stored in the registers on the GPU, so that they can be accessed at the required level of performance. This leads to high per-thread register requirements. In contrast, on modern processor platforms the large caches help to avoid this issue. Despite this important optimization still missing, first experiments indicate that the performance is competitive with that of the streaming algorithm on the same platform [100]. However, as we do not consider the work on the subspace algorithm for GPUs to be mature yet, we do not include it in the evaluation.

7.3. Evaluation

In this section, we evaluate both the streaming and the subspace algorithm. We first describe the datasets that we used and the configurations of the sparse grids. Then, we present results that cover performance, performance portability and the effect of auto-tuning for the unified streaming algorithm. Afterwards, we show the same types of results for the subspace algorithm. The evaluation of the subspace algorithm additionally includes a comparison of the different algorithms with the recursive algorithm as the baseline.

7.3.1. Datasets and Experimental Setup

To evaluate both auto-tuned regression algorithms, we use two datasets: the DR5 dataset and the Friedman1 dataset. The DR5 dataset is a real-world dataset that is based on the fifth data release of the Sloan Digital Sky Survey [3]. We use a postprocessed version of the original dataset as described by Dirk Pflüger [122]. The DR5 dataset contains measurements from five optical filters, the target value is the spectral redshift. Through the redshift the distance to the measured galaxies can be calculated. To give an idea of the spatial structure of the postprocessed dataset, we provide projections onto the coordinate planes for the five measured dimensions in Fig. 7.4.

In our experiments, we learn the DR5 dataset at a fixed mean-squared error and compare the runtimes and derived performance metrics. To learn the DR5 dataset, we use spatially-adaptive sparse grids based on two refinement strategies: surplus refinement

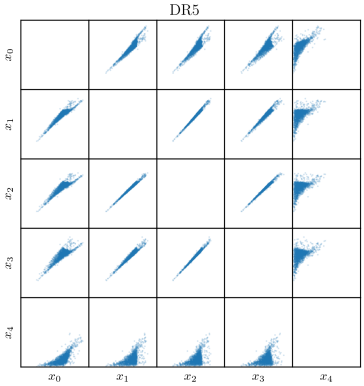


Figure 7.4.: Projections onto the coordinate planes of the DR5 dataset. For this illustration 2000 data points were uniformly sampled.

and support refinement. The target error values and the configuration of the surplus-refined sparse grids are based on experiments by Dirk Pflüger [122]. We chose the configuration that minimized the error in these experiment. By selecting parameters for support refinement that achieve the same error as the surplus experiments, a fair comparison of the refinement strategies is possible.

The Friedman1 dataset is a 10-dimensional synthetic dataset that was generated by uniformly sampling the function

$$\text{fried1}(\mathbf{x}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon, \tag{7.16}$$

with ϵ representing normally distributed noise given by the normal distribution $\epsilon \sim \mathcal{N}(0, 1)$. Due to the construction of the dataset, five dimensions are purely noise. This dataset has been originally presented by Jerome Friedman [56]. It has been shown that sparse grids can learn the Friedman1 function to a low error using only 400 grid points and modified-linear basis functions [72]. However, due to its synthetic nature, the size can be arbitrarily chosen. We can therefore use this dataset for weak-scaling experiments and for experiments that demonstrate the maximum performance potential of the regression algorithms.

As the Friedman1 dataset is only used for performance testing, we chose a sparse grid large enough so that the hardware platforms under test are fully utilized. Furthermore,

Name	DR5-Sur-S	DR5-Supp-S/DR5-Supp-D	Fried1-S/Fried1-D
Basis	mod.-lin.	mod.-lin.	mod.-lin.
Precision	single	single/double	single/double
λ	10^{-5}	10^{-5}	10^{-5}
$n/n_{\max}$	6	10	7
CG iterations	500	500/300	100/100
Ref. type	surplus	support	regular
Config.	$r_s = 7, r_p = 200$	$t_{\text{supp}} = 500$	-
Final grid size	34568	51293	397825
Data points	371908	371908	$2 \cdot 10^5$ per device
MSE	$4.9 \cdot 10^{-4}$	$4.9 \cdot 10^{-4}$	-

Table 7.5.: The regression configurations for both datasets. For surplus refinement, the settings were chosen so that they are comparable to prior results by Dirk Pflüger and Alexander Heinecke [122, 72]. Support refinement was configured to reproduce the same error as surplus refinement. As the Friedman1 derived Fried1-S dataset is used purely for performance testing, a regular sparse grid was chosen.

a regular sparse grid was chosen. As we learn the DR5 dataset with spatially-adaptive sparse grids, this allows for a comparison of the effect of the different grid types on the performance.

Table 7.5 shows the parameters of the experiments. The DR5 experiments are split into two categories for the two refinement criteria. Additionally, support refinement experiments were conducted using both single and double-precision arithmetic. To achieve the same error, the single-precision experiments required more CG iterations. The mean-squared errors for the DR5 dataset as shown in Tab. 7.5 were computed by predicting the values of a separate $6 \cdot 10^5$ data points test dataset that was not used for training.

Both the unified streaming algorithm and the subspace algorithm support linear as well as modified-linear basis functions. As the modified-linear approach allows for a lower error of $4.9 \cdot 10^{-4}$ without any grid points on the boundary, our experiments focus on modified-linear basis functions. However, if linear basis functions are used with otherwise identical parameters, for the DR5 dataset an MSE of $5.8 \cdot 10^{-4}$ was computed. This result immediately highlights the advantage of using modified-linear basis functions.

In all auto-tuning experiments, we averaged the runtime measurements across five kernel executions per search step. Thereby, we tried to mitigate the effect of power-saving features of the hardware platforms.

7.3.2. Performance and Portability of the Unified Streaming Algorithm

In several experiments, we investigate performance and node-level scalability of the unified streaming algorithm. We further demonstrate that support refinement allows for a significantly lower time-to-solution compared to surplus refinement. Afterwards, we evaluate the performance portability of the unified streaming algorithm. In the third and final part of this section, we look at the differences between the multi-evaluation and the transposed multi-evaluation operator. For evaluating the unified streaming algorithm, we only make use of modified-linear basis functions, as this type of basis function enabled a lower error and is algorithmically slightly more challenging.

As a first step of this evaluation, we consider a node with eight GeForce GTX 1080 Ti graphics processors. On this node, we conducted the single-precision experiments shown in Tab. 7.5 using up to eight GPUs. The restriction to single-precision arithmetic reflects the limitation of GeForce GTX 1080 Ti graphics processor which does not support double precision efficiently (see Tab. 3.1). As the size of the DR5 dataset is fixed, it was used for strong-scaling experiments. We used the configurable-size Friedman1 dataset for weak-scaling with $2 \cdot 10^5$ data points per GPU to expose enough parallelism. The DR5 experiments allow for an analysis of the performance in a realistic setting, whereas the Friedman1 experiments are intended to showcase maximal performance. As we varied the refinement strategies, we further use the DR5 experiments to compare support refinement and surplus refinement.

In all experiments in this section, we use an auto-tuned set of parameters that was obtained using line search. The auto-tuning itself is detailed in Sec. 7.3.3. In tuning for the DR5 experiments, we took the number of devices into account by performing auto-tuning for eight devices simultaneously. Thereby, we can account for the lower available per-device work at high device counts. As the Friedman1 experiments guarantee enough work per device due to the larger sparse grid and larger generated dataset, we used the parameters tuned for a single device in the multi-device context.

Strong-Scaling and Refinement

We consider two algorithms, the unified streaming algorithm and the masked streaming algorithm, and combine them with two refinement criteria, support refinement and surplus refinement. Figure 7.5 shows the duration and achieved performance of the experiments with the DR5 dataset. On a single device, the slowest combination was

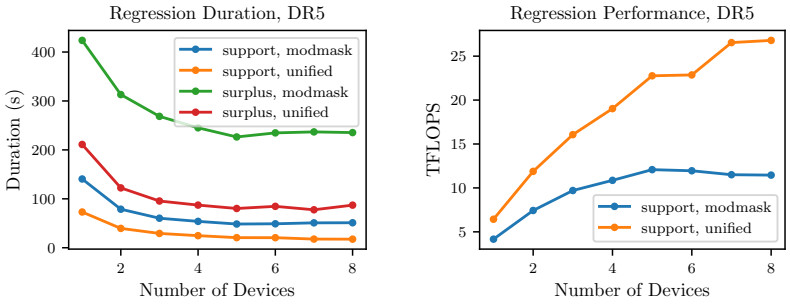


Figure 7.5.: Strong scaling using one to eight Nvidia GTX 1080 Ti GPUs. In these experiments, the DR5 dataset was learned. Both support refinement and the auto-tuned unified streaming algorithm individually contribute to improved performance. The best result is obtained by combining support refinement with the unified streaming algorithm. As the performance of the surplus-based refinement approach varies throughout the refinement steps, it is not shown.

the combination of the masked streaming algorithm and surplus refinement. It required 424 s to complete the experiment. The fastest combination was the unified streaming algorithm together with support refinement. This combination required only 73 s. Therefore, the two new components, the unified streaming algorithm and support refinement, achieve a speedup of 5.8x on a single device compared to the prior approach.

Increasing the number of devices, the unified streaming algorithm together with support refinement consistently outperform the other combinations. Using all eight devices, this combination only required 18 s, whereas the combination of the masked streaming algorithm and surplus refinement took 235 s. Therefore, the new components achieve a speedup of 13.1x compared to the prior approach. Note that support refinement only required 0.3 s using eight devices and its runtime is therefore insignificant relative to that of the solver.

The better performance of a single device is explained by the improved vectorization approach and auto-tuning. However, the unified streaming algorithm displays better scalability as well. While the implementations are not identical and the improved scalability is partially due to technicalities such as avoiding allocations, one parameter has a strong effect. Due to the small size of the grid, the row-splitting parallelization approach

controlled by the *data-split* parameter was chosen with a value of 8 by the auto-tuner. Thereby, there was eight times the parallel work available for the transposed operator compared to the masked algorithm.

As we targeted the same error, we can compare the achieved performance to older results by Alexander Heinecke [72]. Alexander Heinecke used the DR5 dataset with the same surplus refinement configuration that we used in a strong-scaling setting. He reported approximately 13 TFLOPS for the masked streaming algorithm on 128 nodes with two Xeon E5-2680 processors each. In his experiments, he employed double precision whereas we use single precision. However, due to the increased number of iterations required to achieve the same error, this is inconsequential. Overall, due to a new refinement criterion, the auto-tuned unified streaming algorithm and modern GPUs, we achieve a 2x speedup over this configuration using only a single node.

Weak-Scaling

The results of the weak-scaling experiments using the Friedman1 experiments are shown in Fig. 7.6. Similar to the results for the DR5 dataset, the auto-tuned unified streaming algorithm is significantly faster at all device counts. Using a single device, a speedup of 4.8x is achieved. If eight devices are used, the speedup slightly increases to 5.8x. These results show that while the unified streaming algorithm is significantly faster than the masked streaming algorithm, if enough work is available the masked algorithm scales only slightly worse.

While the masking algorithm achieved 13 TFLOPS using eight GPUs, the unified streaming algorithm achieved 50 TFLOPS or 48% of the GPU peak performance of the node assuming boost frequencies of 1.8 GHz. Given a limit of 60% of the peak performance due to the instruction mix, this corresponds to 80% of the achievable peak performance.

Performance Portability

To investigate performance portability, we performed the DR5 and Friedman1 experiments on four GPUs and one processor platform. The GPUs we used were a FirePro W8100, a Vega VII, a GeForce GTX 1080 Ti and a Tesla P100. Our processor platform had two Xeon Gold 5120 processors. The results of these experiments are shown in Fig. 7.7. We provide both the duration of the experiments and the measured performance in TFLOPS. The dotted lines in the performance charts indicate the theoretically achievable performance. With regard to the runtimes, it is important to remember that

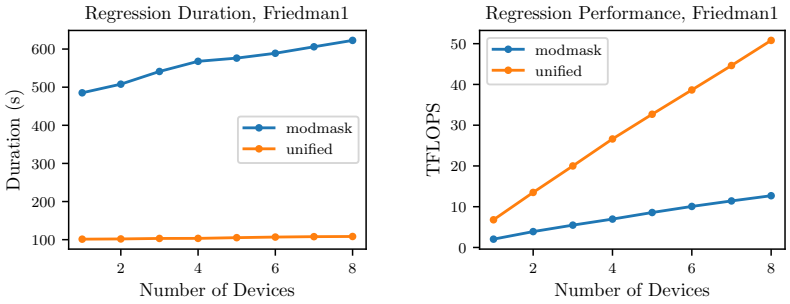


Figure 7.6.: Weak scaling using one to eight Nvidia GTX 1080 Ti. We used Friedman1 datasets with $2 \cdot 10^5$ data points per device. A regular sparse grid with level 7 (0.5M grid points) was used, i.e., the size of the grid to process is fixed. The unified streaming algorithm displays near-optimal scalability at a high absolute performance, achieving 48% (of a maximum of 60%) of the peak performance of eight devices.

the DR5 experiments compute more CG iterations in the single-precision case compared to the double-precision case (500 vs 300 iteration). Thus, devices supporting double precision at half the single-precision rate achieve similar runtimes in both cases. On the other hand, the Friedman1 experiments use the same number of iterations. Therefore, most devices require double the time to finish the Fried1-D experiment. The Vega VII GPU is the exception, as it has a double-precision rate that is a quarter of its single-precision rate.

The runtimes displayed fit the theoretical performance of the different devices. As the dual-socket Xeon Gold 5120 and the older FirePro W8100 have the lowest peak performance, the Tesla P100, the GeForce GTX 1080 Ti and the Vega VII achieve much higher performance. For the Vega VII, as the most recent device, the lowest single-precision runtimes were measured. As the Tesla P100 is the highest-performance double-precision device in our survey, the best double-precision runtimes were measured for this device.

Furthermore, the performance results show that we achieved a major fraction of the achievable peak performance on all devices in the comparison. The worst result was obtained for the 2xXeon Gold 5120 platform at 38% of the peak performance. Adjusted for the performance model described in Sec. 7.1.4, this corresponds to 64% of the achiev-

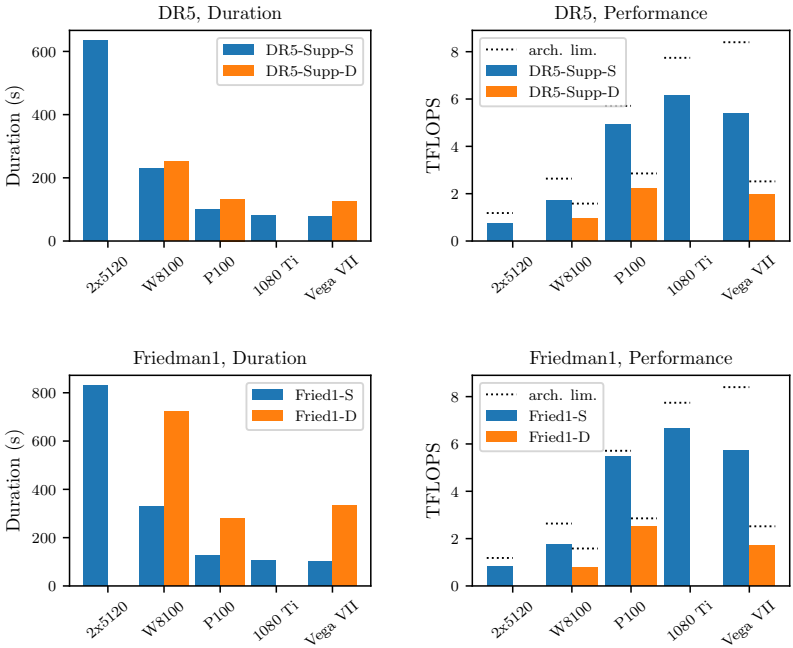


Figure 7.7.: Duration and performance of the unified streaming algorithm for the DR5 and Friedman1 datasets across different hardware platforms. Missing items are due to device or OpenCL platform limitations. The dotted lines in the two performance charts indicate the achievable performance limited by the architecture.

able peak performance. The different GPU platforms achieved not only a much higher performance than the processor platform, but also generally a better utilization of the computational resources. At the low end, when processing the Fried1-D experiment the FirePro W8100 achieved 38% peak performance or 67% of the achievable peak performance. At the high end, the Vega VII achieved 58% of the raw peak performance or 78% of the achievable peak performance in the Fried1-D experiment. The best device utilization was achieved for the Tesla P100 which reached 96% achievable peak in the Fried1-S experiment.

Excluding the FirePro W8100, between 64% and 96% of the achievable performance was measured across all experiments. Therefore, the experiments show that our auto-tuned implementation approach facilitates achieving a large fraction of the peak performance on these modern GPU devices. The utilization reported likely underestimates the actual utilization, as the performance was calculated based on the highest clock frequency for each device. Especially the FirePro W8100 tends to reduce its frequency under heavy load, which we did not account for in these experiments. However, this is investigated for sparse grid clustering in Sec. 8.7.2.

While the durations vary between the two datasets, similar performance was measured for both datasets. This highlights that the DR5 experiments offer enough parallel work to fully utilize a single device.

In Fig. 7.7, some results are missing. The OpenCL platform used on the Xeon Gold 5120 platform does not implement support for double-precision atomics even though the hardware itself supports it. As this feature is needed for the matrix column splitting approach, the algorithm would need adjustments to be supported on that platform. As stated earlier, the GeForce GTX 1080 Ti only supports double precision at a rate too low to be usable for our data mining purposes.

Multi-Evaluation and the Transposed Operator

So far, we evaluated the multi-evaluation and the transposed multi-evaluation operator combined. However, as there are performance differences between the two operators, we investigate those in the following.

For the performance of the individual operator, we conducted experiments where we performed a single call to each operator. These experiments were repeated five times and the measurements averaged. To show that the behavior is similar on all hardware platforms, we considered the same hardware platforms as for the performance portability experiments. As we could not measure a qualitatively different behavior for double-

Name (/Transposed)	Description	PVN value
data-block/grid-block	improve ILP	1
grid-split/data-split	more thread blocks	1
max-dim-unroll	pipeline efficiency	1
prefetch-size	local memory required	16
transfer-whole-dataset/transfer-whole-grid	for multi-GPU	true
use-local-memory	enable local memory	false
local-size	work-group size	64

Table 7.6.: The PVN parameter combination for both operators of the unified streaming algorithm used as the parallelized and vectorized baseline.

precision arithmetic, we only show results for single-precision arithmetic.

The results of these experiments for both datasets are shown in Fig. 7.8. A first glance at the runtimes shows that the runtimes of the two operators differ remarkably. The lower runtimes of the multi-evaluation operator are an effect of the unified streaming algorithm skipping all one-dimensional evaluation with $l_j = 1$. In separate experiments with linear basis functions for which the results are not shown, we measured nearly identical performance of both operators. As one-dimensional evaluations cannot be skipped for linear basis function, the performance matched that of the transposed operator in the depicted modified-linear case.

7.3.3. Auto-Tuning the Unified Streaming Algorithm

In this section, the benefits of auto-tuning the unified streaming algorithm are investigated. First, we show that performance significantly improves through auto-tuning for both regression operators on all hardware platforms. Then, we examine the contribution of the individual parameters across all hardware platforms. Lastly, we compare different search strategies and discuss the total time required for auto-tuning on the Tesla P100 platform.

Benefit of Auto-Tuning

To assess the benefit of auto-tuning, we use a PVN parameterization approach. That is, the baseline is parallelized and vectorized and parameter values are chosen to disable the corresponding optimizations to the extent possible. The values of the PVN combination used in the experiments in this section are shown in Tab. 7.6.

As a first step, we compare the performance of the PVN parameter combination on

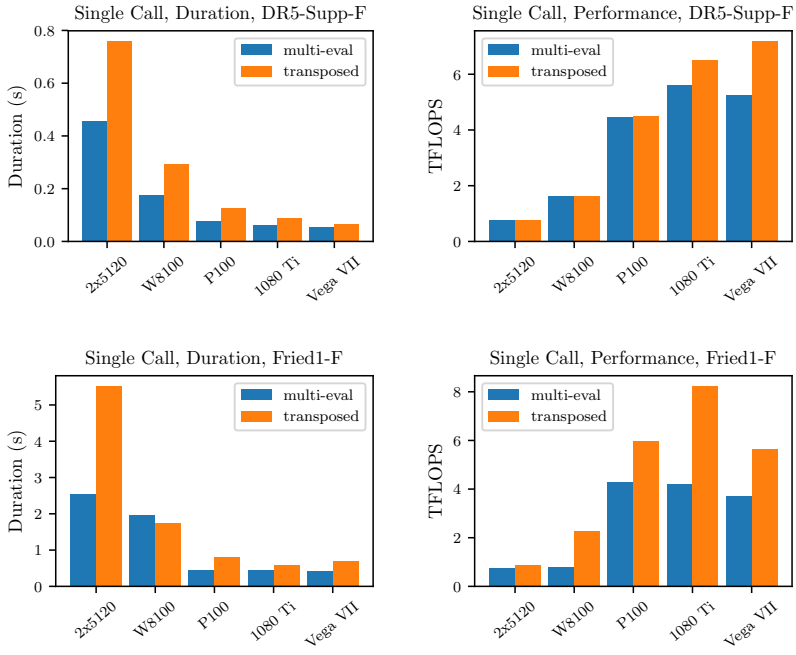


Figure 7.8.: The performance of the two operators for modified-linear basis functions. As $l_j = 1$ basis function evaluations can be skipped in the modified-linear case, the multi-evaluation operator is significantly faster. The performance differences between the regular Friedman1 experiments and the spatially-adaptive DR5 experiments show that the performance of the multi-evaluation operator in part depends on the structure of the sparse grid.

each hardware platform to the performance of the auto-tuned parameter combinations. We used two experimental setups, the DR5-Supp-S and Fried1-S experiments, and two search strategies: line search and neighborhood search. These experiments were conducted on the same hardware platforms as the performance experiments. We measured the runtimes of individual kernel calls and report the mean runtime over five kernel calls to reduce the effect mainly of frequency scaling. In this section, we only discuss the single-precision results, as using double-precision arithmetic yielded no deeper insights. For completeness, double-precision results can be found in Sec. B.1.

Figure 7.9 shows the results of the experiments described above. The figure shows both the speedups achieved over the PVN parameter combination (bars) and the duration ofs the kernel calls (red and green dots). Overall, both line search and neighborhood search can strongly improve performance and achieved similar kernel runtimes after auto-tuning.

The observed speedup varied strongly depending on the hardware platform. On the dual-socket Xeon Gold 5120 platform, the least benefit from auto-tuning was measured with speedups ranging from 1.04x to 1.86x. In contrast, the strongest improvements were measured on the older FirePro W8100 platform with speedups ranging from 2.40x to 6.29x. With respect to the different search strategies, with one exception no significant differences could be measured in the achieved performance. In auto-tuning for the DR5-Supp-S experiment on the Tesla P100 platform, the line search strategy achieved a significantly higher performance. The kernel runtimes suggest that the search strategies detected the same or an equivalent optimum. In manual experiments, the achieved runtimes could not be further improved. This suggests that the achieved performance is the global maximum or very close to it.

Generally, both operators show strong benefits from auto-tuning. However, on most platforms the multi-evaluation operator could be improved further than the transposed operator. The underlying reason is the difference in the amount of registers used by the operators. As storing a level-index tuple in the register file of a GPU requires twice the registers compared to a data point, register pressure on GPUs was higher for the transposed operator.

Parameter Contribution

We have shown that auto-tuning strongly improves performance. As the next step, we investigate the contribution of the individual parameters using the same approach as in Sec. 5.5. That is, we reset individual parameters of the parameter combination

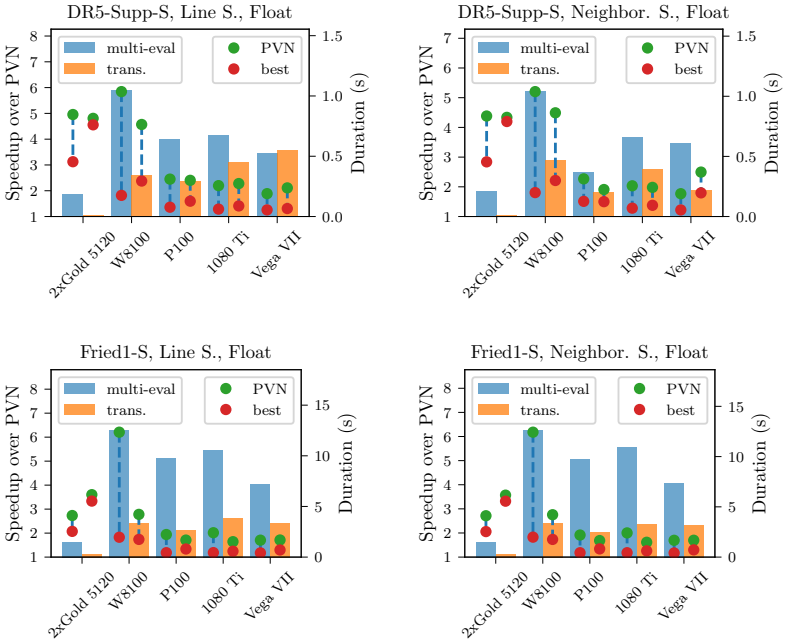


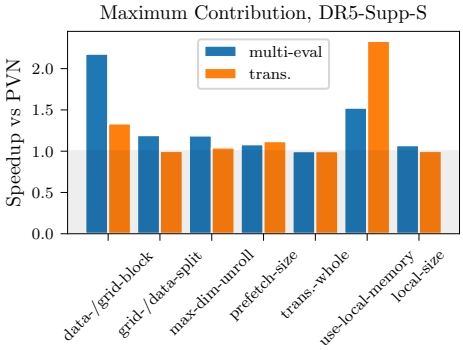
Figure 7.9.: The performance improvements of auto-tuning the unified streaming algorithm. Results are shown for both datasets and two search strategies: line search (left) and neighborhood search (right). Speedups between 1.04x and 6.29x were obtained over PVN initial parameter values (bars). The durations shown are for a single operation execution (dots).

obtained through auto-tuning to their PVN combination value. Then, we measure the modified parameter combination and compare its performance to that of the unmodified auto-tuned parameter combination. For these experiments, we used the auto-tuned parameters obtained through line search. As parameters are useful if they improve performance on at least one hardware platform, we report the maximum contribution of a parameter across all five hardware platforms.

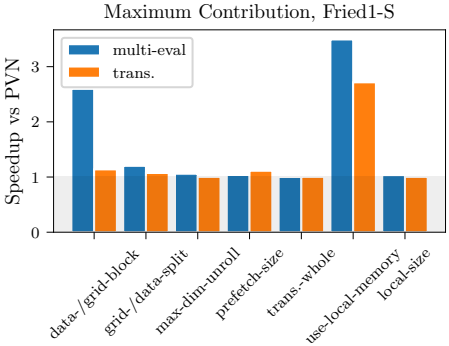
The results of these experiments are shown in Fig. 7.10 for both datasets. The blocking parameters and the local-memory-related parameters provide the largest benefits. This shows that improving instruction-level parallelism and utilizing the local memory are critical, the latter of course on GPUs. On the Xeon Gold platform, where a local memory is not implemented physically, the use-local-memory parameter had a negligible effect on performance. The parameters with the highest contribution in the DR5-Supp-S scenario proved even more beneficial in the Fried1-S scenario. Conversely, parameters that contributed less to the DR5-Supp-S performance had an even lower contribution in the Fried1-S scenario. This is explained by the different grid sizes of the scenarios. The Fried1-S scenario uses a larger sparse grids and therefore parameters that increase the amount of available parallelism are less beneficial as enough "raw" parallelism is available.

For the two operators, the parameters behave differently. The blocking and split parameters are all more beneficial in the multi-evaluation case. These parameters improve performance at the cost of a higher register pressure. However, as the transposed operator needs to store double the data in the register files of GPUs, register pressure is already higher for the transposed operator. Therefore, there was less of an opportunity for improved performance.

The *max-dim-unroll* parameter only proved to be useful in one case: multi-evaluation of the DR5 dataset. Still, this parameter improved performance by up to 19%. The *prefetch-size* parameters mainly act as supporting parameters for local memory use. From these results, we can conclude that the additional pending memory requests only yield small benefits, in this case of up to 8%. The *local-size* parameters, which control the size of the work-groups, have no effect on performance. A larger work-group would allow a more efficient use of the memory bandwidth, as data can be shared between more threads. However, memory bandwidth seems to be sufficient so that more and smaller work-group have no negative effect. Finally, the *transfer-whole-dataset* and *transfer-whole-grid* parameters can improve performance if work is split across multiple GPUs. As these experiments only used a single GPU, improved performance was not expected.



(a) Parameter contributions for the DR5-Supp-S experiment



(b) Parameter contributions for the Fried1-S experiment

Figure 7.10.: Contribution of individual parameters to the performance of the best parameter combination obtained through line search. The charts show the maximum benefit of a parameter for any of the hardware platforms.

Auto-Tuning on a Tesla P100 GPU

As the last step in the evaluation the unified streaming algorithm, we investigate auto-tuning on the Tesla P100 platform in more detail. Figure 7.11 shows the runtimes of the operators during the auto-tuning process, i.e., for different parameterizations. In addition to line search and neighborhood search, we provide results for the Monte Carlo search strategy, as it gives an idea of the difficulty of the search problem. For this search strategy, we do not use the PVN combination as it draws random parameter values.

As Fig. 7.11 shows, line search and Monte Carlo search achieve the same level of performance. The final performance of the neighborhood search is competitive for the transposed operator, but worse for the multi-evaluation operator. The Monte Carlo search approach achieved competitive performance in both cases, but required more search steps until a near-optimal parameter combination was found. Additionally, it needed to evaluate the most low-performance parameter combinations. Only the line search approach reached a near-optimal minimum within few search steps.

Finally, we consider the total time required for the auto-tuning process. The durations shown in Fig. 7.12 include the five kernel executions per search step. The compilation of a kernel only took 1×10^{-4} s on average on the P100 platform. This illustrates a major advantage of using the OpenCL C kernel language for auto-tuning: compilation is not an impediment for fast auto-tuning.

As Fig. 7.12 shows, using a directed search strategy is generally beneficial. We have seen above that the Monte Carlo tuner was able to find a near-optimal parameter combination in all four cases. However, in two of the four cases line search found its best overall combination faster, in one case they performed similar and only in one case was the Monte Carlo approach faster. Additionally, overall duration for tuning was highest for the Monte Carlo approach, though this is an effect of the fixed iteration count. Neighborhood search took significantly longer than line search for the multi-evaluation operator. For the transposed operator the durations were similar. Overall, line search was the most consistent search strategy of the three strategies evaluated. In three out of four cases, it took the least time for tuning and in all cases found a near-optimal parameter combination. In the appendix (Sec. B.2), we display the best parameter values of all search strategies for this device.

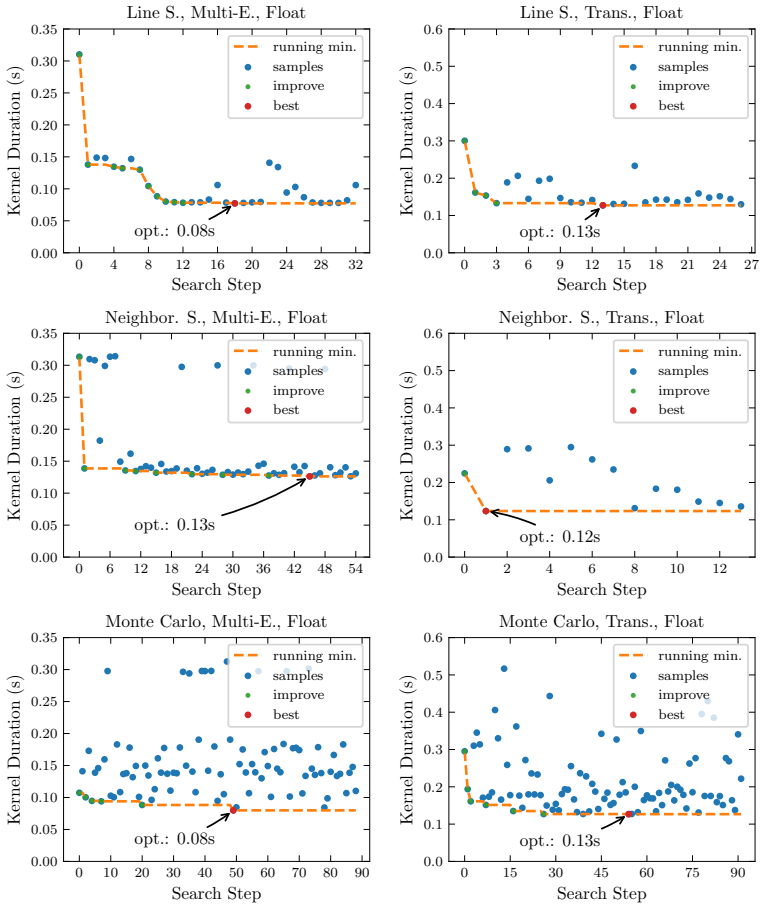


Figure 7.11.: Auto-tuning of the unified streaming algorithm on the Tesla P100 platform for the DR5-Supp-S setup. Shown is the performance at each search step for the multi-evaluation (left) and the transposed operator (right). The directed search strategies use a PVN parameterization as initial parameter values.

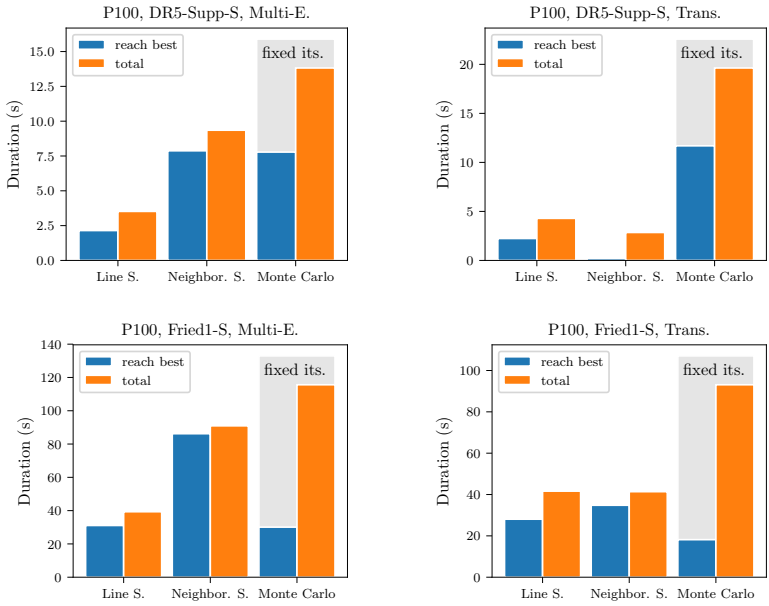


Figure 7.12.: The total time required for auto-tuning the unified streaming kernels for the DR5-Supp-S and Fried1-S setups and the time to reach the best overall parameter combination. Runtimes were measured on the Tesla P100 platform. The durations include five executions of each kernel variant.

7.3.4. Performance and Portability of the Subspace Algorithm

To assess the performance of the subspace algorithm, we use the DR5-Supp-D and Fried1-D setups as described in Tab. 7.5. To demonstrate performance portability, we compare the subspace algorithm to the streaming algorithm and the recursive algorithm on four processor platforms. We chose two platforms by AMD, one with an A10-7850K and one with an Epyc 7551P processor, and two dual-socket Intel platforms. The Intel platforms use Xeon Gold 5120 processors and Xeon E5-2670 processors, respectively. The AMD A10-7850K is a two-module processor that is equivalent to a dual-core CPU regarding floating-point performance. In contrast, the Epyc 7551P is a high-performance 32-core processor based on the Zen microarchitecture. The Intel platforms cover two different high-performance microarchitectures as well. The Xeon Gold 5120 is based on the Skylake architecture, whereas the Xeon E5-2670 is a Sandy Bridge processor. We (again) used AutoTuneTMP with line search as the search strategy to obtain auto-tuned parameters that were used for the experiments in this section.

In addition to results for modified-linear basis functions, we further provide results for linear basis functions. The purpose of these experiments is to show that for the subspace algorithm the type of basis function only has a minor effect on performance. In these experiments, we used the same settings as in the experiments with modified-linear basis functions, only the basis functions themselves were changed.

Unfortunately, of the hardware platforms considered, a high-performance OpenCL implementation was only available for the Intel devices. On other platforms, no vendor implementation was available and more generic implementations such as POCL were not able to vectorize our OpenCL kernels. For a comparison across multiple hardware platforms, we therefore could not use the unified streaming algorithm. Instead, we used two implementations of the streaming algorithm built with AVX intrinsics for vectorization and OpenMP for parallelization, one for linear and one for modified-linear basis functions. These implementations were optimized manually and achieve nearly the same performance on the dual-socket Xeon Gold 5120 system as the OpenCL streaming algorithms.

Figure 7.13 shows the results for the DR5 experiments. The baseline for the speedups in the performance charts are the results of the recursive algorithm on each platform. On all four hardware platforms used, a similar pattern can be observed. The recursive approach is slowest in all cases, the streaming approach is faster and the subspace algorithm is again faster than the two other algorithms. For modified-linear basis functions on the Xeon Gold platform, the recursive algorithm took 2782 s, whereas the stream-

ing variant required only 701 s. The subspace algorithm reduces the duration to only 230 s. Therefore, on this platform the streaming algorithm is 4.0x faster than the recursive approach and the subspace algorithm is 3.0x faster than the streaming algorithm. Using linear basis functions on the same platform, the streaming algorithm achieves a speedup of 3.9x over the recursive algorithm. The subspace algorithm achieves a 3.5x speedup over the streaming algorithm. Generally, performance is slightly higher if linear basis functions are used. This is an effect of the vectorized one-dimensional basis function evaluations. Overall, the subspace algorithm is a substantial improvement over the streaming approach, comparable to the increased performance the streaming algorithm achieved over the recursive algorithm.

Furthermore, the subspace algorithm enables processors to be more competitive with GPU devices. For the unified streaming algorithm, the Vega VII GPU turned out to be the fasted device for both datasets. It required 73 s for the DR5-Supp-S setup. Achieving the same error on the Xeon Gold platform using the subspace algorithm took 231 s. Therefore, while the Xeon Gold platform took 8.5x longer using the unified streaming algorithm, the subspace algorithm narrows the gap to a 3.2x difference in runtimes.

All results for the DR5 dataset use support refinement and therefore benefit from the improved performance of this refinement approach (as was shown in Sec. 7.3.2). A comparison of the subspace algorithm with support refinement to the masked streaming algorithm with surplus refinement would yield an even higher speedup.

The results for the Fried1-D setup are displayed in Fig. 7.14. While the ranking of the algorithms is the same, there are significant quantitative differences. As the sparse grid in these experiments is regular and the recursive as well as the subspace algorithm benefit from the regular structure of the sparse grid, these two algorithms performed better compared to the DR5 experiment. Relative to the recursive algorithm, the subspace algorithm is 45x faster on the AMD Epyc platform for linear basis functions and 43x faster for modified-linear basis functions. Across all platforms and the two types of basis functions, a speedup of $\approx$ 30x is observed. Compared to the streaming approach, the subspace algorithm achieves a speedup of 12x on the Epyc platform using modified-linear basis function. For linear basis functions, the speedup increases to 14x on the same platform.

As described in Sec. 7.2.6, we profiled the subspace algorithm on a platform with Intel Skylake cores which led to the conclusion that the algorithm runs efficiently on this architecture. Furthermore, we measured that the streaming intrinsics kernel runs efficiently on all hardware platforms in our survey. Therefore, the similar relative performance of

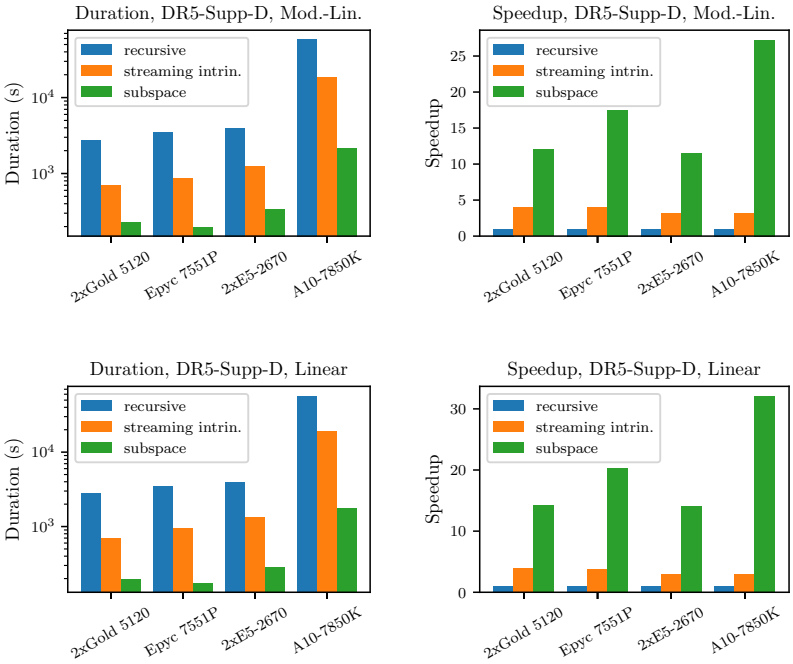


Figure 7.13.: A comparison of the subspace, streaming and recursive algorithms using the DR5-Supp-D setup. The experiments with linear basis functions used the same settings as those with modified-linear basis functions, except for the type of basis function. While the masked streaming algorithm is clearly faster than the recursive algorithm, the subspace algorithm is much faster than either of them.

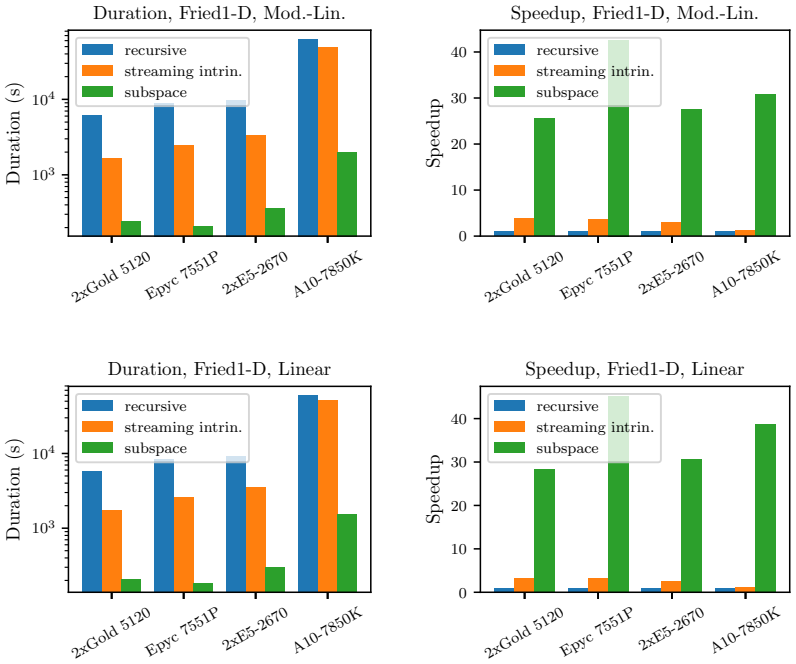


Figure 7.14.: Investigating the performance of the recursive, streaming and subspace algorithm for the Friedman1 dataset. Again, the experiments with linear basis functions used the same settings as those with modified-linear basis functions, except for the type of basis function. As the sparse grid used is regular, the streaming algorithm performs relatively worse. The subspace algorithm benefits from its lower complexity.

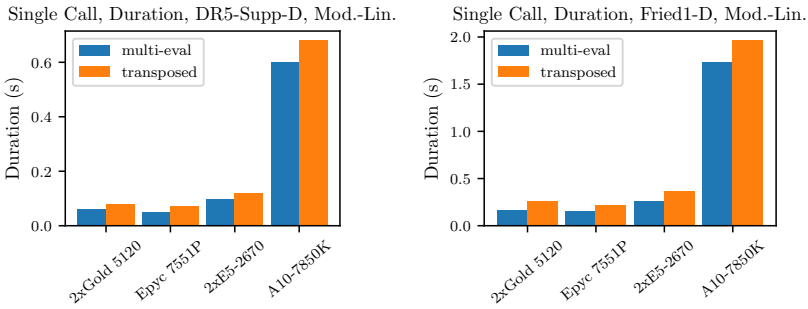


Figure 7.15.: Durations of a single call of each operator for both datasets across all processor platforms.

the algorithms on each hardware platform suggests that the subspace algorithm runs efficiently on the other hardware platforms as well. As there are substantial differences between the hardware platforms, the subspace algorithm can be considered performance portable.

Multi-Evaluation and the Transposed Operator

To analyze possible performance differences between the multi-evaluation and the transposed multi-evaluation operator, we look at a single call of each operator. Results for this type of experiment are shown in Fig. 7.15 for both datasets. The transposed operator is slightly faster on all platforms. However, we suspect that the observed difference in performance is an effect of slightly different implementations and not a fundamental difference between the algorithms. These results show that the synchronization required by the transposed operator does not affect performance.

7.3.5. Auto-Tuning the Subspace Algorithm

In this section, we investigate auto-tuning of the subspace algorithm. To that end, we apply the same approach as in the evaluation of the unified streaming algorithm. First, we show the significant performance improvements obtained by auto-tuning both regression operators on multiple platforms. Then, we examine the contribution of individual parameters across all hardware platforms. Afterwards, auto-tuning on a dual-socket Xeon Gold 5120 platform is analyzed in detail. Finally, we discuss the time required for

Name	Description	PVN value
chunk-size	data point blocking	16
enable-subspace-skipping	skip subspace	false
reuse-intermediates	avoid redundant 1d operations	false
unroll-vectorization	unroll vectorized loop	false
vector-padding	8 required if unrolling	4
list-ratio	limit memory requirements	0.1

Table 7.7.: PVN parameter combination for the multi-evaluation and the transposed multi-evaluation operator used as the parallelized and vectorized baseline.

auto-tuning on the same platform.

Benefit of Auto-Tuning

To assess the benefit of auto-tuning, we compare the parameter combination obtained through auto-tuning to the PVN parameter combination, the usual thread- and vector-parallel baseline. The parameter values for the PVN combination are shown in Tab. 7.7. For these experiments, we again used line search and neighborhood search.

In Fig. 7.16, we show the results of the PVN comparison on four hardware platforms for both the DR5-Supp-S as well as the Fried1-S setup. The advantage of the auto-tuned approach versus the PVN parameter combination is given as a speedup (bars). We further provide the absolute runtimes of the kernels (green and red dots). To account for frequency scaling and cache effects, we used five kernel executions to smooth the kernel runtimes. The runtimes shown are the average of a single kernel call. We omitted the runtimes for the AMD A10-7850K processor, as its absolute performance was much lower compared to the other devices.

Through auto-tuning the performance of the two operators is improved in all cases and speedups of 2x to 4.2x are achieved. Generally, both operators strongly benefit from auto-tuning. In most cases, the search strategies found a minimum with (nearly) the same performance. Only in one case, optimizing the transposed operator for the Fried1-D setup on the Xeon Gold platform, did the neighborhood search not achieve the same performance as the line search.

The runtimes show that the PVN performance of the transposed operator was worse compared that of the multi-evaluation operator. For the transposed operator, a lock needs to be acquired before the subspace can be processed. However, the cost of synchronization can be lowered by choosing a larger value for *chunk-size*, as the acquired lock is used more efficiently. The multi-evaluation operator does not require any syn-

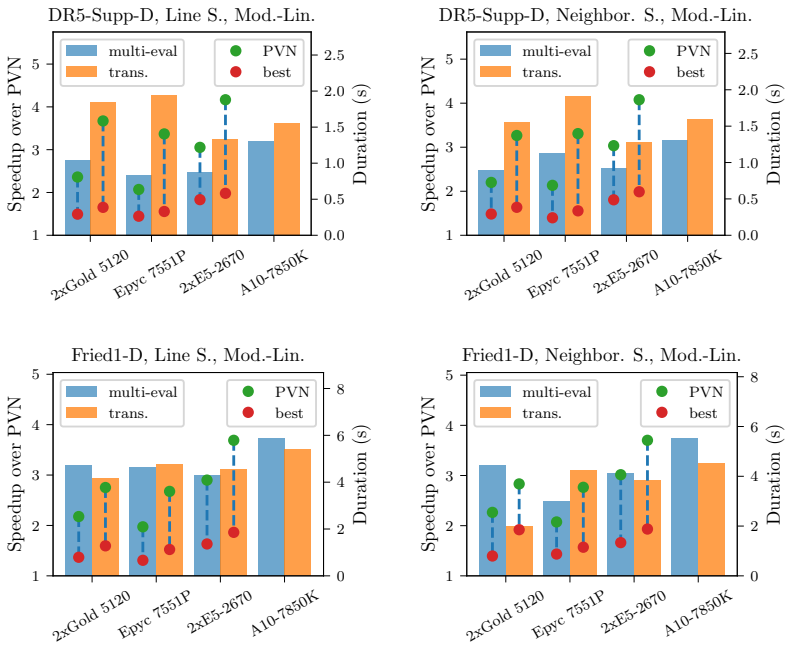


Figure 7.16.: The performance improvements achieved through auto-tuning the subspace algorithm. Results are shown for both datasets and two search strategies: line search (left) and neighborhood search (right). Speedups between 2.0x and 4.2x were obtained over PVN initial parameter values (bars). The durations shown are for a single operation execution (dots). Durations for the A10-7850K are not shown due to the low absolute performance of the device compared to the other devices.

chronization and is therefore initially faster.

Similar to the unified streaming algorithm, we attempted to manually improve on the parameter values obtained through auto-tuning. However, we were not able to further improve performance. Therefore, we consider the auto-tuned parameter values to be the global optimum or very close to it. Overall, auto-tuning strongly improved the performance of the subspace algorithm for both datasets and on all hardware platforms.

Parameter Contribution

To quantify the contribution of individual parameters additional experiments were performed. Again, we reset individual parameters to their PVN combination value. The performance of the thereby generated parameter combination was measured and compared to the auto-tuned parameter combination. As the auto-tuned parameterization, we used the parameter values returned by line search.

The results for both datasets are displayed in Fig. 7.17. It is straightforward to identify the two most important parameters. The *chunk-size* parameter and the *reuse-intermediates* parameter strongly improved performance, both enabling an up to ≈3x speedup. The speedups of these two parameters show that a more simplistic variant of the subspace algorithm, e.g., as shown in Sec. 7.2, would actually not be competitive with the streaming algorithm (for spatially-adaptive sparse grids).

The other parameters had a negligible effect on the performance for the Fried1-D setup, but a significant effect in case of the DR5-Supp-D experiment. In case of the DR5-Supp-D setup, the subspace-skipping parameter as well as the unroll-vectorization parameter were important. The subspace-skipping parameter enables an up 1.2x speedup, whereas the unroll-vectorization parameter enables an up to 1.4x speedup. Note that the unroll-vectorization parameter is tied to the vector-padding parameter to ensure correctness. The list-ratio parameter optimizes memory usage. As these experiments only targeted optimal performance and the two scenarios investigated have low memory requirements, this parameter cannot contribute to the overall performance.

The results in Fig. 7.17 highlight that the type of grid, regular or spatially-adaptive, leads to different performance characteristics. In published work, we showed that for the DR5 dataset subspace skipping becomes more important for a more-deeply refined grid. In those experiments, subspace skipping enabled a 1.8x speedup [120].

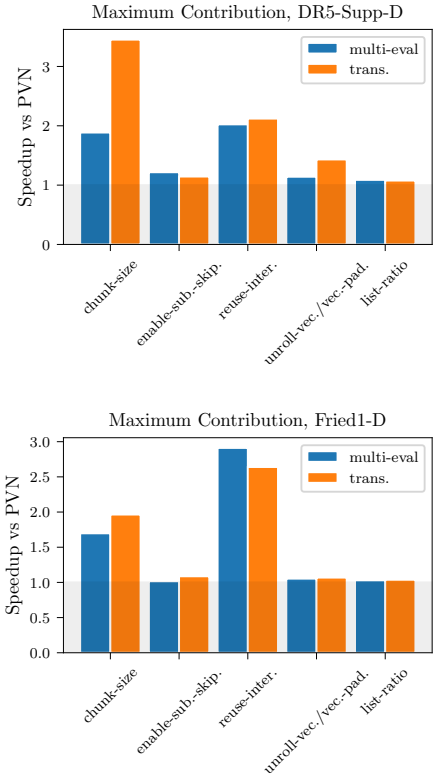


Figure 7.17.: The charts show the maximum benefit of a parameter for any of the hardware platforms used (see Fig. 7.16). Results are shown for the DR5-Supp-D setup (upper) and the Fried1-D setup (lower). The differences between the two datasets are explained by different types of sparse grid used. For this comparison, the results of line search were used.

Auto-Tuning on a Dual-Socket Xeon Gold 5120 Platform

For a deeper look at auto-tuning of the subspace algorithm, we investigate auto-tuning on the dual-socket Xeon Gold 5120 platform. Figure 7.18 shows the time required for executing the compute kernel variants with different parameterizations throughout the parameter search. The results shown are for both operators and the DR5-Supp-D experiment. In addition to line search and neighborhood search, we (again) provide results for the Monte Carlo approach as well.

The results show that all three search strategies can find a near-optimal solution quickly. Line search proved to be more consistent compared to neighborhood search requiring around 10 search steps to find a near-optimal parameter combination. Neighborhood search required more search steps. In case of the transposed operator 24 search steps were needed. Even the Monte Carlo approach is able to find a good solution within 10 search steps. This indicates that the search space is not too difficult to learn. We show the best parameter values of all search strategies for this device in the appendix (Sec. B.3).

Lastly, we consider the overall time required for auto-tuning on the Xeon Gold platform. Figure 7.19 shows the time required to find the best parameter combination as well as the time required until the tuner finished. Due to the fixed number of search steps used, the Monte Carlo tuner again had the longest runtimes. It sometimes found a near-optimal solution quickly, as for the DR5-Supp-D experiments and the transposed operator, but in other cases it took longest.

Line search and neighborhood search generally required similar amounts of time both to converge to a local minimum and to finish the search. In three out of four cases the line search had a slight edge on the neighborhood search. More than 200 s were needed only in one case.

In the line search experiments, the compilation of a single kernel took on average 1.7 s. This could be reduced by a search method with support for parallel compilation such as AutoTuneTMP's parallel line search. The number of repetitions used for averaging could be reduced as well, but of course at a trade-off with respect to tuning quality.

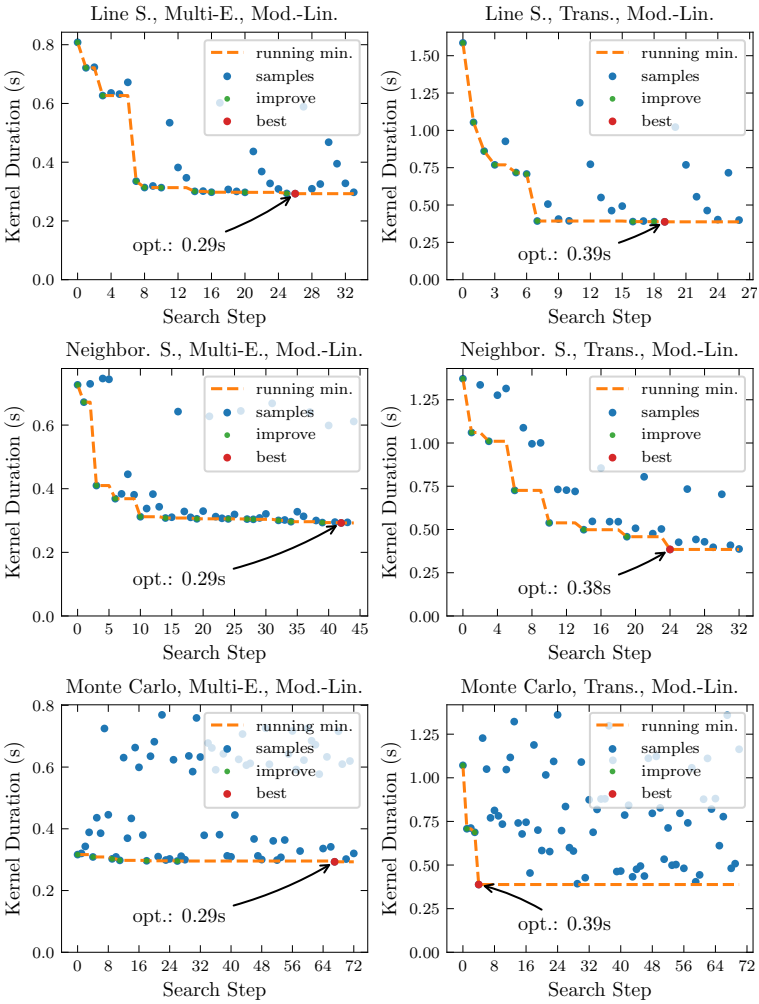


Figure 7.18.: Auto-tuning of the subspace algorithm for the 2xXeon Gold 5120 platform using the DR5-Supp-D experimental setup. The results for the multi-evaluation (left) and transposed operator (right) show that a fast parameterization is located quickly. Only the neighborhood search required more than 10 search steps.

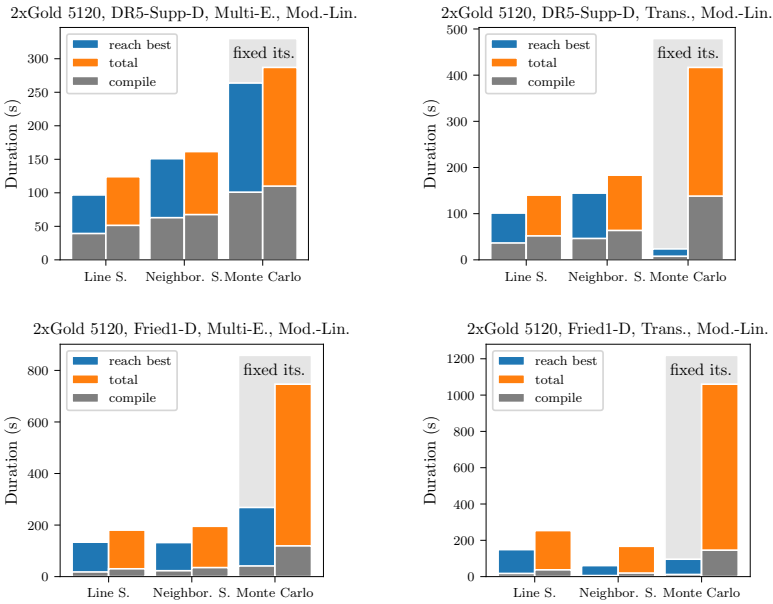


Figure 7.19.: The total time required for auto-tuning the subspace kernels for the DR5-Supp-D and Fried1-D setups and the time to reach the best overall parameterization in each case. Runtimes were measured on the 2xXeon Gold 5120 platform. As shown, the durations include all JIT compilations and five executions of each compute kernel variant.

7.4. Summary

Throughout the previous sections, we presented and evaluated two auto-tuned sparse grid regression algorithms. The unified streaming algorithm achieved close-to-optimal performance on a wide range of hardware architectures. We showed that it significantly improves on the state-of-the-art masked streaming approach. The subspace algorithm was compared to both the recursive and the masked streaming algorithm on multiple hardware platforms. On four processor platforms, it was shown to be a strong improvement compared to both competing algorithms.

To obtain these results, we used AutoTuneTMP's generalized kernel approach to integrate AutoTuneTMP with the OpenCL implementation of the unified streaming algorithm. The subspace algorithm was implemented with AutoTuneTMP's CPPJIT approach, though without using the optimization templates. Both algorithms benefited strongly from auto-tuning. In case of the unified streaming algorithm, auto-tuning enabled an up to 6.29x speedup with generally high speedups except for the single processor-based platforms. For the subspace algorithm, strong improvements could be measured on all hardware platforms with speedups between 2.0x and 4.2x.

As part of the evaluation of the unified streaming algorithm, we showed that support refinement can be a better approach to spatial adaptivity in datadriven problems than the commonly-used surplus refinement approach. In our experiments, it enabled a better than 2x speedup for the DR5 dataset. By combining the unified streaming algorithm with support refinement, a 5.8x speedup over the masked streaming approach together with surplus refinement was measured.

8. Clustering on Sparse Grids

In this chapter, we present the distributed sparse grid clustering algorithm. As first steps, we define clustering and consider some approaches to clustering. Then, we introduce the sparse grid clustering algorithm from a high-level perspective. After the high-level introduction, we describe the further structure of this chapter.

Estivill-Castro argued that the data mining task of clustering is notoriously hard to define [47]. We therefore only give an informal definition of clustering:

Definition 8.0.1 (Clustering)**.** Clustering, or cluster analysis, partitions a dataset according to a given measure of similarity. The partitions returned by the algorithms are called clusters.

Clustering is one of the most important operations in data mining. Because of this, a wide range of clustering algorithms have been proposed. A classic approach to clustering is the k-means algorithm [69]. This algorithm starts with an initial guess of k cluster centers. In an iterative process, assignment of data points to cluster centers and the location of the cluster centers is improved. Many extensions of k-means exist that improve the basic algorithm. Kanungo et al. proposed a variant that uses k-d-trees to reduce the number of comparisons [88]. Arthur et al. use an improved scheme for guessing the initial cluster centers [9].

Another class of clustering algorithms use a density-based approach. DBSCAN is a widely-used clustering method that was originally proposed by Ester et al [45]. DBSCAN spans an ϵ-sphere around the data points. Clusters are extended by joining overlapping spheres that contain a sufficient number of data points. It was originally stated that the complexity of DBSCAN is $\mathcal{O}(m \log m)$ for a dataset with m data points. However, Gan et al. showed that it is at least $\Omega(m^{4/3})$ [58]. DENCLUE is a second example of a density-based clustering algorithm [79]. It uses a kernel density estimation approach.

Notable further clustering algorithms include spectral clustering [99] and clustering using neural networks [168]. Many more clustering algorithms are described in the literature [69, 47].

As sparse grid clustering has been implemented for heterogeneous platforms and can make use of distributed resources, we briefly address the literature in GPU-accelerated and distributed clustering. In 2006, and therefore before the age of CUDA and OpenCL, Takizawa and Kobayashi presented a GPU-accelerated k-means algorithm [141]. Further GPU-enabled k-means variants have been developed since then [50, 83, 17, 51]. GPU-enabled variants of the DBSCAN algorithm have been developed as well. CUDA-DClust uses spatial partitioning to reduce the number of distance calculations [21]. G-DBSCAN operates without spatial partitioning and therefore has a quadratic complexity in the number of data points [5].

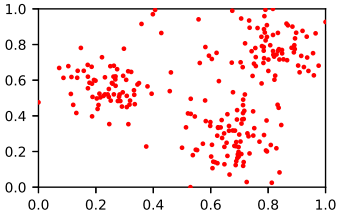
As a distributed k-means variant, k-means++ was able to utilize a Hadoop cluster with 1968 nodes clustering a dataset with 4.8 million data points [11]. Distributed map-reduce variants of the DBSCAN algorithm are implemented by MR-DBSCAN and RP-DBSCAN. MR-DBSCAN could cluster 1.9 billion data points using 128 nodes [70]. RP-DBSCAN was able to compute the clusters of a dataset with 4.4 billion data points using 12 nodes [137].

CUDA-DClust, MR-DBSCAN and RP-DBSCAN use spatial partitioning to avoid a quadratic dependency on m. However, the k-d-tree-like data structure employed by CUDA-DClust should become less effective for higher dimensionalities. Similarly, the partitioning schemes used by MR-DBSCAN is affected by the curse of dimensionality. However, RP-DBSCAN implements a spatially-adaptive partitioning approach that enables the processing of slightly higher-dimensional datasets.

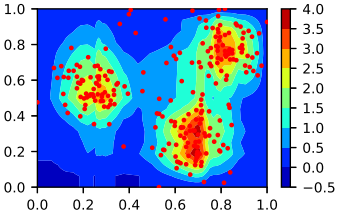
The distributed sparse grids clustering algorithm is a density-based clustering algorithm. It uses sparse grid function spaces as ansatz spaces for a density estimation of the data points. Given a dataset $T^{(c)} := \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$, sparse grid clustering is a four-step algorithm:

1. Compute a sparse grid density estimation.
2. Create a k-nearest-neighbor graph (an approximate solution can be sufficient).
3. Use the density estimation to prune low-density nodes and edges.
4. Find the weakly-connected components of the graph and return them as clusters.

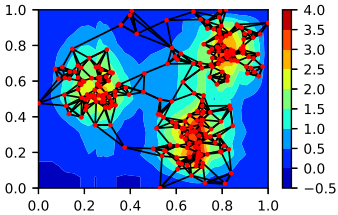
These steps are illustrated in Fig. 8.1 for a 2-dimensional dataset with slightly overlapping clusters. By pruning low-density nodes and edges, connections between clusters get removed. Therefore, the desired clusters are obtained through the connected component search. Of course, what is considered a low density depends on the application;



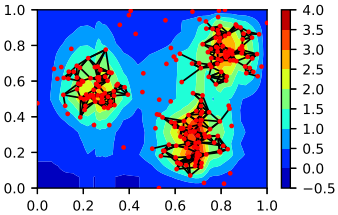
(a) The dataset for cluster analysis



(b) A sparse grid density estimation of the 2d dataset.



(c) After calculation of the kNN graph



(d) The kNN graph after pruning

Figure 8.1.: The sparse grid clustering algorithm is used to cluster a two-dimensional dataset with three slightly overlapping clusters. A kNN graph is calculated and then pruned by using a sparse grid density estimation to remove nodes and edges in low-density regions. After the pruning step, three clusters remain.

this property is shared with other density-based clustering algorithms. We provide manually-optimized OpenCL algorithms for each step of the algorithm, except for the connected component search. For distributed environments such as supercomputers, we implemented a manager-worker approach using MPI.

In the following sections, we introduce the steps of the algorithm in more detail and then evaluate the algorithm. We present the sparse grid density estimation in Sec. 8.1. Afterwards, we introduce our approach for computing the k-nearest-neighbor graph (Sec. 8.2), describe the pruning step (Sec. 8.3) and finally the connected component search (Sec. 8.4). Having introduced the algorithms for the steps of the clustering algorithm, the node-level and distributed implementation of these components is presented in Sec. 8.5. The sparse grid clustering algorithm exposes parameters that determine the obtained solution. Our approach for choosing these parameters and for measuring

clustering quality is described in Sec. 8.6. In the evaluation in Sec. 8.7, we consider the node-level as well as the distributed case. We demonstrate performance portability across a set of node-level platforms and two supercomputers. For most of the evaluation, we use regular sparse grids and several 10-dimensional datasets. As the last part of the evaluation, we consider spatial adaptivity for three datasets with 5, 10 and 20 dimensions.

8.1. Estimating Densities on Sparse Grids

To obtain a density estimation, we use a sparse grid density estimation approach. The underlying density estimation idea is based on work by Hegland et al [71]. Their approach was first applied to sparse grids by Peherstorfer et al [114].

The density estimation approach by Hegland et al. uses an initial density guess f_ϵ that is smoothed using a spline-smoothing approach:

$$\hat{f} = \arg\min_{u \in V} \int_\Omega (u(\mathbf{x}) - f_\epsilon(\mathbf{x}))^2 d\mathbf{x} + \lambda \mathcal{C}(u). \tag{8.1}$$

The term $\int_\Omega (u(\mathbf{x}) - f_\epsilon(\mathbf{x}))^2 d\mathbf{x}$ ensures that $\hat{f}$ approximates the initial density guess f_ϵ. The regularization term $\mathcal{C}(u)$ introduces a smoothness constraint and is controlled by the regularization parameter $\lambda \in \mathbb{R}$.

In our work, we use the initial density guess proposed by Hegland et al.:

$$f_\epsilon := \frac{1}{m} \sum_{i=1}^{m} \delta_{\mathbf{x}_i}. \tag{8.2}$$

This approach centers a Dirac delta function $\delta_{\mathbf{x}_i}$ at each data point $\mathbf{x}_i$.

For estimating densities, we choose the same regularization strategy as for regression. That is, we again employ a weight-decay regularization approach [19]. As the underlying function space, we choose the sparse grid function space $V_n^{(1)}$. Therefore, the problem to solve becomes

$$\hat{f} = \arg\min_{u \in V_n^{(1)}} \int_\Omega (u(\mathbf{x}) - f_\epsilon(\mathbf{x}))^2 d\mathbf{x} + \lambda \sum_{i=1}^{N} \alpha_i^2. \tag{8.3}$$

As was shown, this approach leads to a system of linear equations

$$(\hat{B} + \lambda I)\boldsymbol{\alpha} = \mathbf{b}, \tag{8.4}$$

with $\hat{B}_{ij} = (\phi_i, \phi_j)_{L_2}$, the identity matrix I and $b_i = \frac{1}{m}\sum_{j=1}^{m} \phi_i(\mathbf{x}_j)$ [114].

The system of linear equations shows a highly useful property of this density estimation approach: the data points only appear on the right-hand side. As the right-hand side linearly depends on the dataset, the whole density estimation algorithm is linear in m. Consequently, the algorithm is well-suited for big data scenarios. Of course, the method further benefits from the use of a sparse grid function space as described in Sec. 6.

$\hat{B}$ has the dimensionality $N \times N$. It is too large to be stored directly if more than $\approx 10^5$ grid points are required. We address this issue analogously to how the related issue of sparse grid regression was addressed. That is, we employ a CG solver [136]. By using a CG solver, the matrix-vector product $\mathbf{v}' = (\hat{B} + \lambda I)\mathbf{v}$ needs to be computed in every iteration. Therefore, we can again use an implicit approach to store the matrix $\hat{B}$ and recalculate components of $\hat{B}$ when they are accessed.

To compute components of $\hat{B}$ the L_2-norm of pairs of basis functions needs to be computed efficiently. We notice that the d-dimensional integral leads to a series of one-dimensional integrals that need to be solved

$$(\phi_{\mathbf{l},\mathbf{i}}, \phi_{\mathbf{l}',\mathbf{i}'})_{L_2} = \int_\Omega \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x})\phi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x})d\mathbf{x}, \tag{8.5}$$

$$= \int_0^1 \phi_{l_1,i_1}(x_1)\phi_{l_1',i_1'}(x_1)dx_1 \cdots \int_0^1 \phi_{l_d,i_d}(x_d)\phi_{l_d',i_d'}(x_d)dx_d. \tag{8.6}$$

Throughout this chapter, we use linear basis functions. For this type of basis function, the one-dimensional integrals can be computed directly:

$$\int_0^1 \phi_{l,i}(x)\phi_{l',i'}(x)dx = \begin{cases} \frac{2}{3}h_l, & x_{li} = x_{l'i'}, \\ h_{l'}\phi_{l,i}(x_{l'i'}) + h_l\phi_{l'i'}(x_{li}), & \text{else.} \end{cases} \tag{8.7}$$

To illustrate the effect of the regularization parameter λ, Fig. 8.2 shows a density estimation for a small dataset with four data points and varied λ. For smaller values, the density estimation more closely resembles the initial density guess. Because our regularization approach penalizes large surpluses, an increasingly larger λ leads to function values close to zero throughout the domain.

For a high-performance density estimation, both the calculation of the right-hand side $\mathbf{b}$ and the matrix-vector product $\mathbf{v}' = (\hat{B} + \lambda I)\mathbf{v}$ need to be calculated efficiently. Calculating the right-hand side requires $\mathcal{O}(mN)$ operations, whereas calculating $\mathbf{v}'$ requires $\mathcal{O}(N^2)$ operations. At first glance, the calculation of the right-hand side might seem more expensive, as $N \ll m$ is a reasonable assumption for learning large datasets.

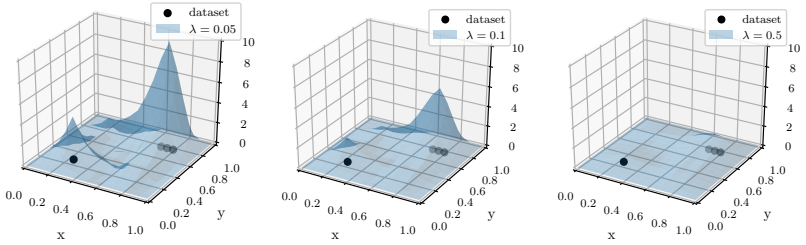


Figure 8.2.: The effect of the regularization parameter λ on a 2d dataset with four data points. The function becomes smoother for higher values of λ.

However, as we use a CG solver, the matrix-vector product gets computed repeatedly.

Algorithm 9 shows the basic approach for computing the right-hand side. As every outer loop iteration computes a single component of $\mathbf{b}$, these iterations are all independent. The outer loop can therefore be executed in parallel. Furthermore, as linear basis functions can be evaluated without branches, the algorithm is straightforward to vectorize.

Algorithm 9: Calculating the right-hand side $\mathbf{b}$.

Input : dataset $T^{(c)} = \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$, sparse grid basis functions $\phi_1 \dots \phi_N$ with $\phi_i = \prod_{d=1}^{\dim} \phi_i^{(d)}$

Output: $\mathbf{b}$

```
for i ∈ {1, ..., N} do
    b_i ← 0
    for j ∈ {1, ..., m} do
        b_i += ∏_{d=1}^{dim} φ_i^(d)(x_j^(d))
    b_i ← (1/m) b_i
```

The algorithm for computing the matrix-vector product $\mathbf{v}'$ is shown in Alg. 10. Again, the outer loop iterations are independent as they compute individual components of $\mathbf{v}'$. Therefore, this algorithm can be parallelized by executing the outer loop iterations in different threads as well. Vectorization is not quite as straightforward, as the two cases of the one-dimensional integral from Eq. 8.7 need to be differentiated. Because $\mathbf{h_l}$ gets precomputed, calculating $\frac{2}{3}h_{l_d}$ is only a single multiplication. As the $x_{li} = x_{l'i'}$ case is very cheap to compute, we always compute both cases and then select the appropriate result. To choose the correct integration case, we use OpenCL's `select` function. As

the condition is a simple comparison, on modern compute architectures the `select` statement can be compiled to a conditional move that is a single instruction.

Algorithm 10: Calculating the matrix-vector product $\mathbf{v}' = (\hat{B} + \lambda I)\mathbf{v}$

Input : $\mathbf{v}$, sparse grid basis functions $\phi_1 \dots \phi_N$ with $\phi_i = \prod_{d=1}^{\dim} \phi_i^{(d)}$, $\mathbf{h}_i = (h_i^{(1)}, \dots, h_i^{(\dim)})$ and grid points $\mathbf{x}_i = (x_i^{(1)}, \dots, x_i^{(\dim)})$

Output: $\mathbf{v}'$

```
for i ∈ {1, ..., N} do
    v'_i ← 0
    for j ∈ {1, ..., N} do
        r ← 1
        for d ∈ {1, ..., dim} do
            r_eq ← (2/3) h_i
            r_else ← h_j^(d) φ_i^(d)(x_j^(d)) + h_i^(d) φ_j^(d)(x_i^(d))
            r ·= select(r_eq, r_else, x_i^(d) = x_j^(d))
        v'_i += r · v_j
    v'_i += λ · v_i
```

Our algorithmic approach might seem wasteful, as the support of the pair of basis functions might be disjoint for an integration result of zero. However, computing a one-dimensional integral requires only few arithmetic operations and checking whether the support overlaps is, relative to the integration, quite costly. Furthermore, skipping the current integration introduces a branch to a vectorized loop. This would only effectively reduce the cost of the operation if all SIMD lanes detect a disjoint support in the same iteration. Otherwise, the same computations are performed with some SIMD lanes disabled through masking.

8.2. k-Nearest-Neighbor Graph Creation

The goal of the k-nearest-neighbor graph creation is to construct a directed graph $G = (V', E)$. Each node $v_i \in V'$ corresponds to a data point $\mathbf{x}_i \in T^{\mathrm{c}}$ so that $|V'| = m$. For the edges $(v_i, v_j) \in V' \times V'$ and the Euclidean distances $d_{i,j} := ||\mathbf{x}_i - \mathbf{x}_j||_2$, we define the sets $F_{(v_i,v_j)}$:

$$F_{(v_i,v_j)} := \{(v_i, v_{j'}) | v_{j'} \in V' \wedge (d_{i,j'} < d_{i,j} \vee (d_{i,j'} = d_{i,j} \wedge j' < j))\}. \tag{8.8}$$

$F_{(v_i,v_j)}$ contains all outgoing edges of v_i with shorter distances that (v_i, v_j). In case of multiple edges with the same distance as d_{ij}, we add all edges with lower index value j'. Now, we can define the set of edges E of the k-nearest-neighbor graph, so that it only contains the k edges with the smallest distances between the corresponding data points:

$$E := \{(v_i, v_j) | v_i, v_j \in V' \wedge |F_{(v_i,v_j)}| < k\}. \tag{8.9}$$

If $m \geq k$, exactly k edges per node are obtained, as we ordered the same-distance edges by their index.

The sparse grid clustering solution is unchanged as long as the weakly-connected components of the k-nearest-neighbor graph are unchanged. Therefore, missing edges do not affect the solution unless they further split a connected component. Additionally, as the graph pruning step removes low-density nodes and edges, the overall algorithm is tolerant with respect to edges that introduce additional connectivity between connected components. These two reasons motivate our use of an approximate k-nearest-neighbor graph algorithm.

The naive k-nearest-neighbor graph algorithm performs $\mathcal{O}(kdm^2)$ comparisons to return the k-nearest neighbors. It compares all pairs of nodes and keeps track of the k nodes with the smallest distance (and potentially lower index). We implemented a binning variant of this algorithm that is shown in Alg. 11. This algorithm implicitly divides the dataset into b partitions. For each data point, the nearest neighbor in each bin gets computed. Of these b nearest neighbors, the k neighbors with the smallest distance are selected.

Through binning, the complexity of the algorithm is slightly reduced from $\mathcal{O}(kdm^2)$ to $\mathcal{O}(dm^2)$, as only a single comparison is required in the innermost loop. Furthermore, as it is linear in the dimensionality d, this algorithm is not affected by the curse of dimensionality. This property is helpful for the higher-dimensional problems we target. A drawback, of course, is the quadratic dependency on m.

Similar to the density estimation algorithms, this algorithm maps well to modern hardware platforms. The outermost loop iterations are independent and the algorithm therefore is embarrassingly parallel. Furthermore, there is only a single branch that can be vectorized through masking.

The parameters of this algorithm are the number of bins b and number of neighbors k. k can be small, but should be large enough so that a large connected component is not split into many smaller ones. In our experiments, we did not observe any improvements

Algorithm 11: A binning $\mathcal{O}(dm^2)$ variant of the naive k-nearest-neighbor algorithm. The `extract_nearest_k` function selects the k bins with the smallest computed distance to the currently processed data point.

Input : dataset $T^{(c)} = \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$
Output: k-nearest-neighbor graph g as neighborhood list

```
for c ∈ {1, ..., b} do
    dists_c ← 0
for i ∈ {1, ..., m} do
    c ← 0 // c iterates the bins
    for j ∈ {1, ..., m} do
        dist ← distance(x_i, x_j) // iterates d
        if dist < dists_{c+1} then
            bins_{c+1} ← j
            dists_{c+1} ← dist
        c ← (c + 1) mod b
    g_i ← extract_nearest_k(dists, bins)
```

for $k > 6$.

Of course, b needs to be larger than k, so that k neighbors can be returned for each data point. Furthermore, b should be as large as possible to minimize the error of the k-nearest-neighbor graph. Finally, it should be small enough, so that the bins fit into the fastest memory of the device. That is, the bins should fit in the register on a GPU and into the L1 cache (or even the registers) on a CPU. In all of our experiments, we set b to 16, as this choice seems to neither significantly affect the detected clusters nor the measured performance.

8.3. Pruning the k-Nearest-Neighbor Graph

Given a sparse grid density estimation and an approximate k-nearest-neighbor graph, we now remove nodes that reside in low-density regions and edges that intersect low-density regions. As what is considered "low-density" depends on the application, a density threshold t is a parameter of the clustering algorithm. To decide whether a node resides in a low-density region, the density estimation is evaluated at the location of the node. Outgoing edges get pruned if the density estimation evaluated at the midpoint of the edge returns a value below t. Viewed from the perspective of the edges, overall each edge gets evaluated at three locations: the two endpoints and the midpoint.

Algorithm 12 shows our pruning approach as a streaming algorithm. The algorithm iterates the dataset. For each data point, first the density at the location of the data point is checked. If the node was not pruned, the midpoints of the line segments between a node and its k nearest neighbors are considered. They are computed by the `load_midpoints` function. The evaluations of $\hat{f}$ imply iterations of the grid points for a complexity of $\mathcal{O}(mkNd)$. Structurally, iterations of the outermost loop are once again independent, which leads to a straightforward parallelization approach. Furthermore, the algorithm is mostly branch-free, even though the pruning of nodes and edges occurs within a branch. Overall, there are $\mathcal{O}(mk)$ branches to be processed. Because the calls to `prune_edge` are cheap and the density estimation evaluations have a higher complexity, the overhead for the conditionals is low.

Algorithm 12: The k-nearest-neighbor graph is pruned by evaluating the density estimation and removing nodes and edges with density below a threshold t. The i-th node is denoted by g_i, $g_{i,j}$ refers to the j-th edge of the i-th node.

Input : k-nearest-neighbor graph g as neighborhood list, dataset $T^{(c)} = \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$, density estimation $\hat{f}(\mathbf{x}) = \sum_{j=1}^N \alpha_j \phi_j(\mathbf{x})$, threshold t

Output: pruned k-nearest-neighbor graph g

```
for i ∈ {1, ..., m} do
    if f̂(x_i) < t then
        prune_node(g_i)
        continue
    p_1, ..., p_k ← load_midpoints(T, g_i)
    for j ∈ {1, ..., k} do
        if f̂(p_j) < t then
            prune_edge(g_{i,j})
```

8.4. Connected Component Detection

Algorithm 13 detects the connected components in the pruned k-nearest-neighbor graph. First, the pruned k-nearest-neighbor graph is converted to an undirected graph. To that end, we simply add all inverted edges. Note that the nodes of the resulting graph can vary in the number of edges they have. In the undirected graph, we perform depth-first searches starting from every node that was not already visited. The starting node of a depth-first search and all nodes that are reachable from it form a cluster. All nodes that

were reached during the searches are marked as visited.

Algorithm 13: A $\mathcal{O}(km)$ algorithm to detect the connected components in the pruned k-nearest-neighbor graph. E_i are the edges of the node with index i.

Input : pruned directed k-nearest-neighbor graph G
Output: connected components/clusters C

Function `search_cluster` $(i, E, \mathbf{v})$**:**
  if v_i **then return** $\emptyset$
  $v_i \leftarrow$ true
  return $\{i\} \cup (\bigcup_{(i,j)\in E_i}$ `search_cluster`$(j, E, \mathbf{v}))$
$G' = (V', E) \leftarrow$ `make_undirected` (G) `// adds inverted edges`
$\mathbf{v} \leftarrow$ (false, ..., false) `// keeps track of visited nodes,` $|\mathbf{v}| = m$
for $i \in \{1, \ldots, |V'|\}$ **do**
  $C_i =$ `search_cluster`$(i, E, \mathbf{v})$
`drop_empty` (C)

As this algorithm has a complexity of $\mathcal{O}(km)$, it can be significantly faster processed than any of the other clustering steps. Because of this, a shared-memory-parallel processor-only implementation is used. To avoid NUMA-related issues, our implementation only uses the processor cores of a single socket. As this algorithm is not performance-critical compared to the other sparse grid clustering steps, we do not describe the implementation of the connected component search in more detail. However, we report the runtimes of the connected component detection in the evaluation section for one of the supercomputers.

8.5. Implementation Methodology

So far, we have primarily focused on an algorithmic perspective. In the following, we first describe the general parallelization approach and the OpenCL-based node-level implementation. Afterwards, we introduce the MPI-based distributed implementation.

8.5.1. Parallelization Approach and Node-Level Implementation

Analogously to the steps of the algorithm, sparse grid clustering consists of four OpenCL kernels. The first two compute the sparse grid density estimation: one calculates the right-hand side and one computes the matrix-vector products. The third implements the k-nearest-neighbor graph creation and the fourth performs the graph pruning.

In Alg. 14, we display the loop structure of the compute kernel that computes the right-hand side of the density estimation to illustrate our parallelization approach. The outer loop iterating the grid points is split three times for three layers of parallelization: inter-node, thread-level and vector-level. As iterations of the outer loop are independent in all four OpenCL kernels, we use an analogous loop splitting approach for the other compute kernels. In our OpenCL implementation, $\text{chunk}_{\text{par}}$ becomes the OpenCL work-size. Consequently, this level of parallelization is mapped to a one-dimensional grid of work-groups. As we do not make use of the explicit vector types in OpenCL (`float4`, `float8`, ...), the vectorization level is implicit in the OpenCL implementation. Most modern OpenCL platforms, especially those that target GPUs, can do vectorization across work-items. Our algorithms were designed to be (mostly) branch-free in their innermost loops. As a result, all four compute kernels vectorize effectively on the hardware platforms we considered.

Algorithm 14: The loop structure of the parallel, vectorized and distributed calculation of **b**. It is assumed that the dataset and sparse grid have been padded so that all chunk sizes are evenly divisible.

Input : dataset $T^{(c)} = \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^m$, sparse grid basis functions $\phi_1 \dots \phi_N$
Output: **b**

```
1  chunk_dis ← N/nodes // per-node chunk size
2  chunk_par ← ... // For thread-like parallelization, e.g., 256
3  chunk_vec ← ... // SIMD width per core, e.g., 32
4  for i_dis = 0; i_dis < N; i_dis += chunk_dis do
5      for i_par = i_dis; i_par < i_dis + chunk_dis; i_par += chunk_par do
           // implemented by the auto-vectorizer
6          for i_vec = i_par; i_vec < i_par + chunk_par; i_vec += chunk_vec do
               // this loop gets vectorized (and thereby removed)
7              for i = i_vec; i < i_vec + chunk_vec; i_vec += 1 do
8                  b_i ← 0
9                  for j = 0 ... m do
10                     b_i += ∏_{d=0}^{dim-1} φ_i^(d)(x_j^(d))
11                 b_i ← (1/m) b_i
```

Apart from parallelization and vectorization, all algorithms use the shared memory of the OpenCL devices. Thereby, performance is improved on devices that implement the shared memory physically, i.e., GPUs. The usage of the shared memory is analogous for all four compute kernels: the data that is streamed in the innermost loop is cached in

Kernel	FP ops./complexity	Arith. int. (ws=1)	Arith. int. (ws=128)
dens. right-hand side	$N \cdot m \cdot d \cdot 6$	1.5 F/B	192 F/B
dens. matrix-vector	CG-iter. $\cdot N^2 \cdot d \cdot 14$	1.2 F/B	149 F/B
create graph	$m^2 \cdot d \cdot 3$	1.0 F/B	129 F/B
prune graph	$m \cdot N \cdot (k+1) \cdot d \cdot 6$	4.5 F/B	576 F/B

Table 8.1.: The complexity and the arithmetic intensity of the clustering OpenCL kernels. Through local memory caching and a sufficiently large work-group size, the algorithms become compute-bound.

the shared memory. Though for reasons of readability not shown directly, we illustrate our approach again using the density kernel depicted in Alg. 14. In the algorithm, j iterates the data points (Line 9). As all work-items process the same data points (but a different grid point), data points can be shared through the shared memory. This entails work-group-level synchronization to ensure that indeed the work-items of a work-group always work on the same values for j.

Table 8.1 lists the complexity and highlights the benefit of using the shared memory. Compared to the machine balance of modern node-level architectures, the arithmetic intensity of a work-group with a single item would be too low to fully utilize all available resources (see 3.1 for the machine balance of modern hardware platforms). However, as the streamed data gets shared across the work-group, the algorithms are compute-bound on all commonly used node-level architectures. For this calculation, sharing of data across a work group with 128 work-items was assumed.

To give a reasonable upper bound for the performance of these compute-bound compute kernels, we consider the instruction mix. Using the same approach as was used for the unified streaming algorithm in Sec. 7.1.4, we consider full-speed instructions, FMAs, and half-speed instructions. As Tab. 8.2 shows, every compute kernel can be implemented with at least one FMA instruction. For example, the innermost loop of the density matrix-vector multiplication kernel can be implemented with 12 instructions including two FMA instructions. Assuming that non-FMA instructions run at half the speed of FMA instructions, the instruction mix limits the achievable performance to $\frac{2}{12} \cdot 1 + \frac{10}{12} \cdot 0.5 = 58.3\%$ of the peak performance. As computing the absolute value can be embedded into operands on AMD GPUs, the achievable performance on devices of this vendor is slightly higher.

Kernel	Hottest Inst.	FMAs	Peak lim. (%)	Peak lim. AMD (%)
dens. right-hand side	5	1	60%	62.5%
dens. matrix-vector	12	2	58.3%	59.1%
create graph	2	1	75%	75%
prune graph	5	1	60%	62.5%

Table 8.2.: The peak limit implied by the instruction mix in the innermost and therefore hottest loop if it is assumed that FMA instructions run at double the speed of all other instructions. All numbers apply to single-precision arithmetic.

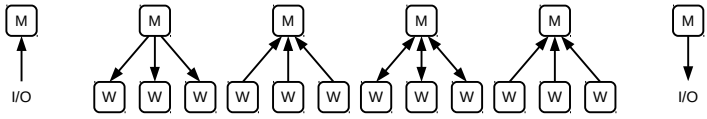
8.5.2. Distributed Implementation

In order to use sparse grid clustering on distributed machines such as supercomputers, we implemented a manager-worker scheme using MPI. In the previous section, we showed the loop structure of the density right-hand-side kernel (Alg. 14) and described how we use the independence of the outer loop iterations to obtain all layers of parallelism. This includes the inter-node parallelism required for distributed computing. In the example in Alg. 14, the iterations over N are split into equally sized chunks, so that every node computes an index range of the same size. This approach is a form of static load balancing. Our clustering implementation supports both single and double-precision arithmetic. For the communication of floating-point data, 8 B per value are used in both cases. Because the datasets can be large, all indices are 8 B as well.

We hold a copy of the sparse grid and the dataset on all nodes which limits the maximum size of the dataset that can be processed. This approach has the advantage that after an initial communication phase, the remainder of the algorithm requires little communication. In some steps only the cheap communication of index ranges is needed.

To explain our distributed approach, we describe it from the perspective of the manager node. The steps of the algorithm, as it is executed by the manager node, are shown in Fig. 8.3. We list the cost of the communication steps in bytes in Tab. 8.3. The assignment of index ranges is omitted due to the low cost of these communication steps.

At the beginning, the manager node reads the dataset and creates a sparse grid of the requested discretization level. It then broadcasts the dataset and the created grid to all worker nodes. As soon as the data is distributed, the manager node triggers the computation of the density estimation. To that end, it assigns index ranges to the worker nodes for the computation of $\mathbf{b}$, i.e., the computation of the right-hand side of the density system of linear equations. The manager node then waits until it has received all chunks of $\mathbf{b}$ from the worker nodes. The next step is the computation of the density



(a) Load dataset (b) Broadcast grid and dataset (c) Receive $\mathbf{b}$ (d) Broadcast $\mathbf{v}$, receive $\mathbf{v}'$ (each CG step) (e) Receive pruned subgraphs (f) Find components

Figure 8.3.: The manager node controls the clustering process. It does not execute the clustering kernels itself, except for the connected component search. The manager perspective highlights the major communication tasks.

Communication Step	Bytes Transferred	Repeated per
Broadcast dataset	$md \cdot 8\,\mathrm{B}$	worker
Broadcast grid	$2Nd \cdot 8\,\mathrm{B}$	worker
Aggregate $\mathbf{b}$	$N \cdot 8\,\mathrm{B}$	-
Broadcast $\mathbf{v}$	$N \cdot 8\,\mathrm{B}$	worker and CG it.
Recv. $\mathbf{v}'$	$N \cdot 8\,\mathrm{B}$	worker and CG it.
Broadcast $\boldsymbol{\alpha}$	$N \cdot 8\,\mathrm{B}$	worker
Recv. pruned kNN graph	$mk \cdot 8\,\mathrm{B}$	-

Table 8.3.: The communication cost in bytes transferred for the steps of the distributed clustering algorithm. Under realistic conditions ($N \ll m$), the initial broadcast of the dataset is most expensive with regard to communication.

matrix-vector product. At the beginning of each CG iteration, the input vector $\mathbf{v}$ is sent to all nodes. Then, the nodes get index ranges assigned for the computation of $\mathbf{v}'$. At the end of each CG step, the manager node gathers all chunks of $\mathbf{v}'$. These steps are repeated for every CG iteration, which of course leads to further communication. After the last CG step, the computation of the density estimation is finished, as the surpluses of the sparse grid function are now available on the manager node. As the density estimation is used by the nodes, the surpluses $\boldsymbol{\alpha}$ of the sparse grid function get broadcasted to the nodes.

The next step is the computation and the pruning of the approximate k-nearest-neighbor graph. Both these steps, from an algorithmic perspective, are implemented as a single distributed operation. Each node receives an index range and computes the k-nearest-neighbors for the assigned range of data points. Then, each node prunes the subgraph that it just computed using the density estimation. Having assigned the index ranges, the manager node waits until it has received all parts of the pruned k-nearest-neighbor graph from the workers. In the final step, the manager node computes the connected components of the pruned graph and returns them as detected clusters.

8.6. Parameter Selection

The sparse grid clustering exposes parameters that need to be chosen carefully, as they determine the computed solution. We first describe how the parameters of the clustering algorithm can be chosen in general, so that a "good" clustering result is obtained. Then, we introduce the Adjusted Rand Index (ARI) as the metric we use in the next sections to assess the quality of the results of our clustering algorithm.

Table 8.4 summarizes the parameters that need to be chosen for sparse grid clustering. The number of parameters increases in case a spatially-adaptive sparse grid is used, as refinement and coarsening need to be set up. Given some measure of quality for the created clustering, we provide an approach for automatically choosing the regularization parameter λ and the clustering threshold t. In contrast to the other parameters, these parameters are more difficult to choose a priori.

For optimal results, the density function should outline the clusters for a certain globally applicable density value that can be chosen as the threshold value t. Unfortunately, the density estimation does not guarantee a maximum or minimum (not even positivity). If a too low value for t is chosen, clusters cannot be appropriately separated. However, if it is too high, too much of the k-nearest-neighbor graph gets removed. The obtained

Name	Purpose
Regular and spatially-adaptive sparse grids	
n	sparse grid discretization level
k	connectivity of the k-nearest-neighbor graph
λ	regularization for smoothness of the density estimation
ϵ	controls error of CG [136]
t	graph pruning threshold value
Spatially-adaptive sparse grids only	
r_p	the number of grid points to refine per step
r_s	the number of refinement steps
t_{ws}	threshold for weight-support pruning
t_{supp}	minimum grid points on support for support refinement
$n_{\max}$	maximum grid level for support refinement

Table 8.4.: The parameters needed to control sparse grid clustering. If a spatially-adaptive grid is used, additional parameters need to be specified.

clusters might therefore be smaller than expected or it might even occur that clusters get split. Furthermore, as the regularization parameter λ determines the shape and number of high-density regions, a careful choice it critical as well.

For comparison, as long as k is chosen with a value that is too high, the connectivity of the connected components is increased, but the same connected components might be obtained in the end. A similar observation can be made for choosing the discretization level n too high. Assuming that a proper value for λ was chosen, superfluous grid points will get a surplus value close to zero.

To automatically select λ and t, we use a two-dimensional adaptive full-grid search similar to the one used by `libsvm` [28]. Every grid point of this search algorithm is associated with a run of the clustering algorithm configured with a specific (λ, t)-tuple. For both parameters, a minimum and maximum value of the search range has to be specified. Additionally, the number of intervals per parameter needs to be supplied. This implies a two-dimensional grid that gets evaluated, i.e., the clustering gets executed for all (λ, t)-tuples. After an initial grid evaluation, we use a greedy heuristic to span a grid around the so far best candidate value and its neighbors. This approach is iterated with smaller and smaller intervals. The greedy heuristic is stopped as soon as the interval size is below a threshold that needs to be supplied. To tune λ, we use a logarithmic grid search approach (with base 10), as a wide range of values is plausible for this parameter. For the threshold t, a linear range proved to be sufficient.

Adjusted Rand Index

To assess the quality of the clustering results, we use the adjusted Rand index (ARI), which is a standard measure in statistics for comparing a pair of partitions [81]. For an interpretation of the ARI we briefly consider the Rand index [128]. For two partitions, it sums up two cases. First, it counts all pairs of data points where the pairs are together in one set in both partitions. Second, it adds all pairs of data points where the pairs are in different sets in both partitions. These are the two cases were the partitions "agree", i.e., the clustering is the same. By using the total number of pairs $\binom{m}{2}$ for normalization, the (unadjusted) Rand index is obtained. The ARI is a variant of the Rand index that is corrected for chance.

To calculate the ARI, we assume a dataset $T^{(c)}$ and two partitions P and Q of the dataset (results of clustering). From that, we compute a $|P| \times |Q|$ contingency matrix M with components $M_{i,j} = |P_i \cap Q_j|$. Further, $a_i := \sum_{j=1}^{|Q|} M_{i,j}$ sums up rows in the contingency table, whereas $b_j := \sum_{i=1}^{|P|} M_{i,j}$ does the same for columns in the contingency table. The ARI can now be defined as

$$ARI(P,Q) := \frac{\sum_{i=1}^{|P|}\sum_{j=1}^{|Q|}\binom{M_{i,j}}{2} - \left(\sum_{i=1}^{|P|}\binom{a_i}{2}\sum_{j=1}^{|Q|}\binom{b_j}{2}\right)/\binom{m}{2}}{\frac{1}{2}\left(\sum_{i=1}^{|P|}\binom{a_i}{2} + \sum_{j=1}^{|Q|}\binom{b_j}{2}\right) - \left(\sum_{i=1}^{|P|}\binom{a_i}{2}\sum_{j=1}^{|Q|}\binom{b_j}{2}\right)/\binom{m}{2}}. \tag{8.10}$$

8.7. Evaluation

To evaluate sparse grid clustering, we first describe the experimental setup, i.e., the parameterization of the algorithm and the datasets. Then, we investigate performance and performance portability using regular sparse grids, first for the node-level platforms and then for two supercomputers. Finally, we show that spatial adaptivity can be used to further improve the runtimes, especially for a 20-dimensional dataset.

8.7.1. Datasets and Experimental Setup

To assess performance and clustering quality of the sparse grid clustering algorithm, we use several synthetic datasets. These synthetic datasets were created by randomly drawing from Gaussian distributions that were placed inside the domain $[0.1, 0.9]^d$ with means μ_i. The standard deviations σ were chosen so that the clusters are unlikely to overlap. For the same reason, a minimum distance of between the cluster centers μ was enforced. By ruling out overlapping clusters, we know a reference solution by con-

Name	Size	Clust.	σ	Dim.	Center Dist.	Noise	Type
1M-10C	1M	10	0.05	10	$7 \cdot \sigma$	2%	node
1M-100C	1M	100	0.05	10	$7 \cdot \sigma$	2%	node
10M-100C	10M	100	0.05	10	$7 \cdot \sigma$	2%	node
10M-3C	10M	3	0.12	10	$3 \cdot \sigma$	0%	dist.
100M-3C	100M	3	0.12	10	$3 \cdot \sigma$	0%	dist.
G5D-100C-5D	0.2M	100	0.05	5	$7 \cdot \sigma$	2%	adap.
G5D-100C-10D	0.2M	100	0.05	10 (5d distr.)	$7 \cdot \sigma$	2%	adap.
G5D-100C-20D	0.2M	100	0.05	20 (5d distr.)	$7 \cdot \sigma$	2%	adap.

Table 8.5.: The datasets used in the node-level, distributed and spatial-adaptivity experiments. Data points were drawn randomly from the Gaussians that represent the clusters. In case of the node-level experiments, noise was added to make reconstruction of the clusters more difficult.

struction. Therefore, it is possible for our algorithm to reconstruct this reference cluster mapping. The datasets used in the node-level and distributed clustering experiments are shown in Tab. 8.5. All datasets, except those used for adaptivity experiment, have a dimensionality of 10. However, in the adaptivity experiments we used 5-dimensional Gaussians embedded in a 5-, 10- and 20-dimensional dataset. To illustrate the 10-dimensional structure of the datasets and that the clusters are not trivially separable, we show projections onto the coordinate planes of the 1M-10C and the 1M-100C datasets in Fig. 8.4. For these illustrations, we did not use the whole datasets, but sampled 2000 points from the data sets. Because more clusters need to be separated, the 100 cluster datasets are more difficult to learn than the 10 cluster datasets. The figure illustrates this, as it is much easier to visually separate the clusters of the 1M-10C dataset compared to doing the same for the 1M-100C dataset.

Peherstorfer et al. showed that sparse grid clustering is competitive with other clustering approaches. Their experiments included real-world datasets [114]. In this work, we use synthetic datasets for two reasons. First, clustering has no well-defined general solution [47]. Due to the construction of our synthetic datasets, we have a reference solution available. By fitting to the reference solution, we can show the "correctness" of our clustering approach relative to the reference solution. Second, it is possible for data points of the dataset to reside in a structure of lower dimensionality than the dataset. Due to the construction of our synthetic datasets, the dimensionality of the embedded structures matches that of the dataset itself, except for the datasets used in the adaptivity experiments. Thus, we can guarantee that an actual higher-dimensional problem is solved. For a real-world dataset, this would require further analysis.

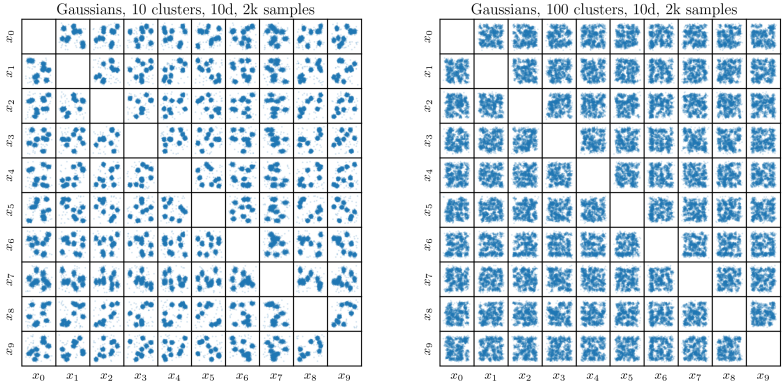


Figure 8.4.: Projections onto the coordinate planes of the 1M-10C (left) and the 1M-100C (right) dataset. For these illustrations 2000 data points were uniformly sampled from each dataset.

Table 8.6 shows the parameters used for setting up the sparse grid clustering algorithm in both the node-level and the distributed case and for the experiments with spatial adaptivity. In the node-level case, the parameters were chosen so that the ARI suggests a high-quality solution to make our performance assessment more realistic, whereas the distributed experiments were designed to purely show the achievable performance with our approach. For the distributed experiments the threshold was chosen relative to the maximum surplus values. In case of the adaptivity experiments the size of the grid was varied, as different adaptivity criteria are employed. Therefore, we cannot provide N in the table. For the node-level and distributed experiments the level stated is the level of the regular grid used. In all experiments conducted, we used single-precision arithmetic.

8.7.2. Node-Level Clustering on Regular Sparse Grids

In this section, we present the performance and performance portability result for our sparse grid clustering algorithm on the node-level. In order to assess the performance, we conducted the node-level experiments described in the previous section and summarized in Tab. 8.6. The experiments were conducted on five hardware platforms of which four are GPU-based. Two of the GPUs are Nvidia devices, the Tesla P100 and the GTX 1080 TI, and two are AMD devices, the FirePro W8100 and the Vega VII. As the final platform, we used a dual-socket node with two Intel Xeon E5-2680v3 processors. The

Name	λ	Threshold t	Level	N	CG ϵ	k	ARI	Type
1M-10C	10^{-5}	667	6	76k	1E-2	6	1.0	node
1M-100C	10^{-6}	556	7	0.4M	1E-2	6	0.89	node
10M-10C	10^{-5}	1167	7	0.4M	1E-2	6	1.0	node
10M-100C	10^{-6}	1000	7	0.4M	1E-2	6	0.90	node
10M-3C	10^{-6}	$0.7 \cdot \max(\boldsymbol{\alpha})$	7	0.4M	1E-3	5	-	dist.
100M-3C	10^{-6}	$0.7 \cdot \max(\boldsymbol{\alpha})$	8	1.9M	1E-3	5	-	dist.
G5-100C-5D	10^{-5}	55	-	-	1E-2	6	≥ 0.99	adap.
G5-100C-10D	$3 \cdot 10^{-8}$	$1.3 \cdot 10^{4}$	-	-	1E-2	6	≥ 0.99	adap.
G5-100C-20D	$7.8 \cdot 10^{-13}$	$7.6 \cdot 10^{8}$	-	-	1E-2	6	≥ 0.99	adap.

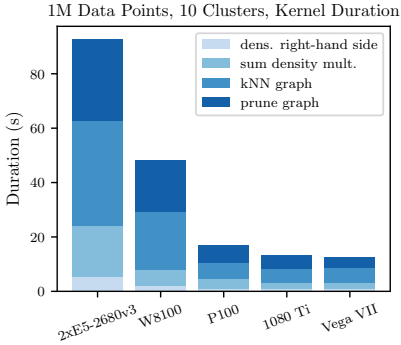
Table 8.6.: The parameters of the sparse grid clustering algorithm for the node-level and distributed performance experiments with regular sparse grids as well as the datasets for experiments with spatial adaptivity.

specifications of these platforms are summed up in Tab. 3.1. In all cases, four runs were averaged to reduce variation due to frequency scaling and power saving measures implemented the hardware platforms.

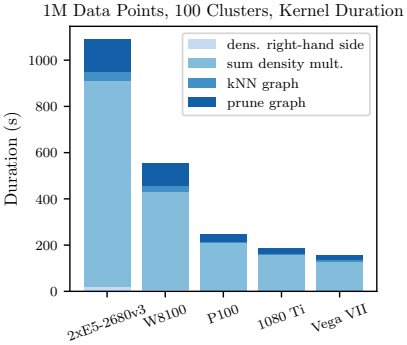
We show the results of the node-level experiments with the one million data points datasets in Fig. 8.5. On the fastest platform, the Vega VII GPU, it took 13 s to process the 1M-10C dataset. Using the same platform, the cluster analysis of the 1M-100C required 158 s. As the 1M-100C has more clusters to detect, a higher discretization level $l = 7$ was required to obtain a similar ARI. Due to the larger sparse grid the cost of the density matrix-vector product is significantly higher compared to the 1M-10C experiments. The fraction of time spend in the individual compute kernels is similar across all five platforms. For both datasets, the differences in measured runtimes can be explained by the differences in raw compute power of the devices.

In Fig. 8.6, we provide the results of the node-level experiments for the 10M-10C and 10M-100C datasets. As expected, the runtimes are longer compared to those for the smaller datasets. Again, the Vega VII GPU achieved the lowest runtimes compared to the other devices. On this device, processing the 10M-10C dataset took 782 s, whereas the 10M-100C dataset required 843 s. The runtimes are similar in both cases, as a sparse grid of the same discretization level was used. Due to less regularization, the density estimation requires slightly more time for the 10M-100C dataset. Again, the five platforms displayed runtimes that differ in accordance with their specifications.

Despite the differences in runtime of the kernels in the four experiments, in three out of four experimental setups the achieved performance was nearly the same. Only in the 1M-10C experiment lead the smaller sparse grid to slightly lower performance of the density

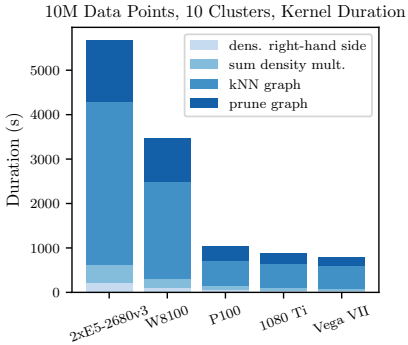


(a) $\lambda = 10^{-5}, l = 6, t = 667$

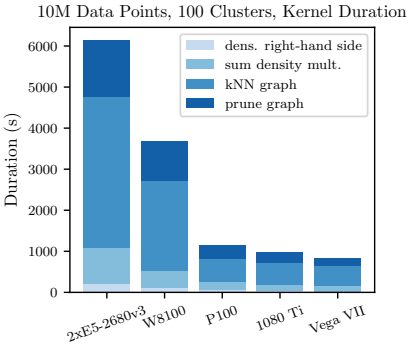


(b) $\lambda = 10^{-6}, l = 7, t = 556$

Figure 8.5.: The duration of the node-level experiments with one million data points on five node-level platforms. The larger sparse grid used in the 1M-100C experiment leads to longer runtimes of the density estimation.



(a) $\lambda = 10^{-5}, l = 7, t = 1167$



(b) $\lambda = 10^{-6}, l = 7, t = 1000$

Figure 8.6.: The duration of the node-level experiments with 10 million data points on five node-level platforms. Due to the size of the dataset and the quadratic scaling, the k-nearest-neighbor graph creation takes up most of the runtime.

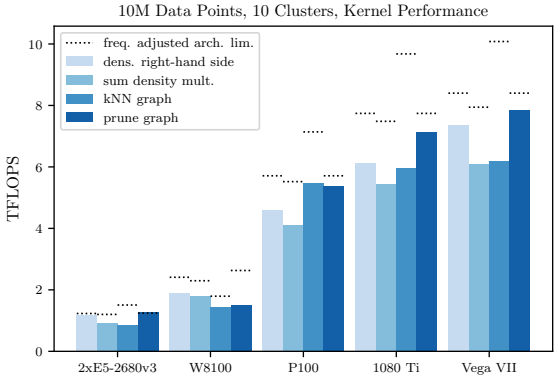


Figure 8.7.: The node-level performance of the clustering algorithm for the 10M-10C scenario using single-precision arithmetic. Due to the instruction mix of these compute-bound kernels, the achievable peak performance is limited to values below 100%. The achievable limit is indicated by the dotted lines.

estimation kernels. Because there are only minor performance differences, we show the performance for the 10M-10C experiment only. Figure 8.7 shows the performance of the compute kernels for the 10M-10C dataset on all five devices. The dotted line above each bar indicates the maximum achievable performance, given the instruction-mix limits described in Sec. 8.5.1. The limit is further adjusted by the average frequency of the devices during the experiments. As the Vega VII GPU did not allow for frequency measurements, we instead assumed the maximum boost frequency of 1.75 GHz on this platform.

Overall, at least 36% of the peak performance is achieved across all devices and compute kernels. With regard to the instruction mix limit, at least 57% of the achievable peak performance was reached in all cases. The best performance was observed for the prune graph kernel with more than 91% of the achievable peak performance measured for four of the five devices. This is likely an effect of the implicit instruction-level parallelism of this kernel, as $k + 1$ sparse grids function evaluations are processed simultaneously by each work-item. On the Xeon E5 platform, a performance 2% above the limit was measured. We attribute this to slight variations in the processor clock, as the frequencies were measured in a separate run.

On average across all compute kernels and devices, 79% of the achievable performance

could be measured. This shows the excellent performance portability of our approach. With regard to absolute performance, the Tesla P100, the GeForce GTX 1080 Ti and the Vega VII achieved the highest level of performance. Those are the most modern devices in the comparison and have the highest theoretical performance. Therefore, the observed performance nicely fits the theoretical performance expectation with no major outlier. The Vega VII GPU is the fastest device overall; it was 13% faster than the GTX 1080 Ti and 35% faster than the Tesla P100. Compared to the slower devices, the Tesla P100 achieved a speedup of 2.23 – 3.29x over the FirePro W8100 and a speedup of 4.41 – 5.49x over the dual-socket Xeon E5-2680v3 node.

While the lower performance of the FirePro W8100 is in part explained by its age, it also experienced significant throttling. This effect was strongest during the k-nearest-neighbor graph creation where the frequency was reduced from 824 MHz to only 467 MHz. However, despite the throttling behavior the FirePro W8100 still achieved a speedup of 1.67 – 1.98x compared to the dual-socket Xeon node.

Because it had the lowest specifications in this survey, the dual-socket Xeon system displayed the lowest performance compared to the GPU-based platforms. However, the achieved fraction of peak performance shows that our approach provides competitive efficiency. Therefore, the performance on this platform illustrates that our sparse grid clustering approach is performance-portable across processor and GPU architectures.

8.7.3. Distributed Clustering on Regular Sparse Grids

To show that our node-level performance portability translates to portability between two highly different supercomputers, we show results for two supercomputers with different node-level architectures. The Cray XC40 “Hazel Hen” system was located at the High-Performance Computing Center (HLRS) in Stuttgart, Germany. Hazel Hen was set up with 7712 nodes. Each node had two Intel Xeon E5-2680v3 processors with 12 cores running at a base clock of 2.5 GHz. The second system was the Cray XC40/Cray XC50 “Piz Daint” located at the Swiss National Computing Centre (CSCS) in Lugano, Switzerland. As a hybrid system, Piz Daint consisted of 1813 XC40 multi-core nodes and 5704 XC50 accelerator nodes. Each XC50 node that we used was configured with an Nvidia Tesla P100 graphics card and a single Intel Xeon E5-2690v3 (2.6 GHz, 12 cores). In the following, we present results for distributed strong-scaling experiments as summarized in Tab. 8.6.

Strong Scaling on Hazel Hen

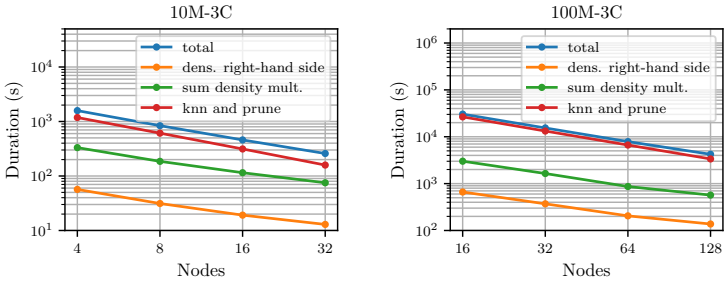
The results for the strong scaling experiment on Hazel Hen are shown in Fig. 8.8. We display the runtime of the experiments for both datasets and the performance in TFLOPS. Additionally, we provide the durations required to detect the connected components, for loading-and-transferring the dataset and for creating-and-broadcasting the sparse grid.

We first address runtime, shown in Fig. 8.8a, and parallel efficiency. The latter is computed from the measured runtimes. Clustering of the 10M-3C dataset took 259 s using 32 nodes and 1583 s using four nodes. Therefore, compared to the four-node run, a parallel efficiency of 76% could be measured. For the larger 100M-3C a run takes significantly longer with 4226 s on 128 nodes and 30 380 s using 16 nodes. Relative to the 16 nodes run, on 128 nodes a parallel efficiency of 90% was achieved, i.e., near-linear scaling. This indicates that the parallel efficiency of the 10M-3C dataset runs was limited by the amount of work available per node, despite the lower maximum node count.

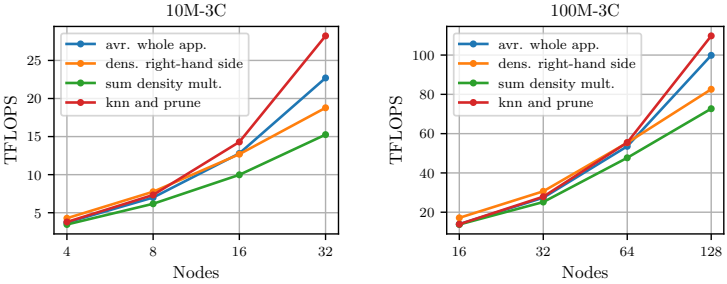
The calculation of the right-hand side of the density estimation system of linear equations depends linearly on both N and m. The density matrix-vector product even depends quadratically on N. As $N \ll m$, we expect limited scalability of the density estimation compared to the graph-related operations. This effect is observable in Fig. 8.8a. As the graph creation quadratically depends on m, the combined graph-creation-and-pruning step is the best scaling part of the algorithm.

The performance of the runs on Hazel Hen at different node counts is shown in Fig. 8.8b. On 32 nodes a performance of 23 TFLOPS was achieved for the 10M-3C dataset. For the 100M-3C dataset and using 128 nodes, a performance of 100 TFLOPS was measured or 41% of the peak performance.

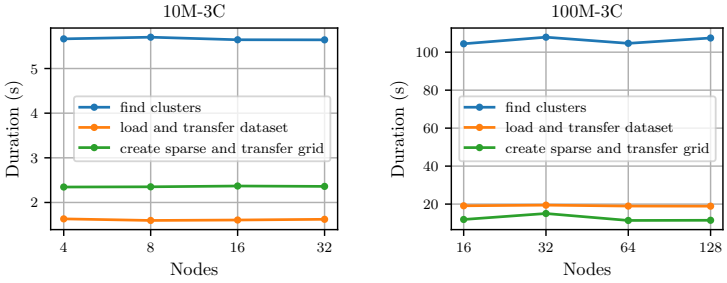
The time required for operations that are not part of the four major OpenCL compute kernels is shown in Fig. 8.8c. We consider loading and broadcasting the dataset as well as creating and broadcasting the sparse grid to be potentially expensive, because of the communication involved. As the connected component search is done by a single compute node, it is potentially expensive as well. However, the measured durations show that the operations related to the dataset and the sparse grid only take up a small fraction of the overall runtime. The connected component search is more expensive. Still, for the 10M-3C dataset using 32 nodes the connected component search only took 5.6 s or 2.2% of the total runtime. Similarly, for the 100M-3C dataset and 128 nodes it took 107 s which translates to 2.5% of the total runtime.



(a) The durations in seconds of the clustering experiments on Hazel Hen



(b) The performance in TFLOPS of the clustering experiments on Hazel Hen



(c) The time required by potentially costly operations that are not major compute kernels.

Figure 8.8.: Strong scaling results for the 10M-3C and 100M-3C datasets on Hazel Hen

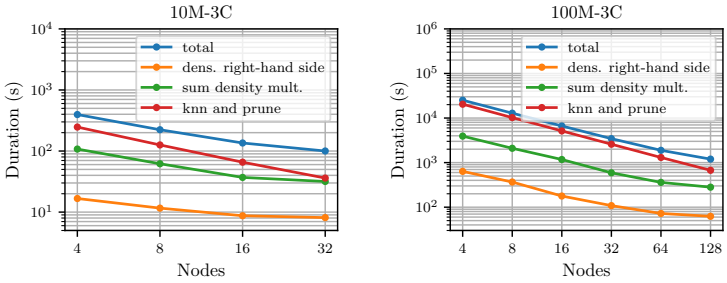
Strong Scaling on Piz Daint

The runs on Piz Daint were conducted prior to some final node-level optimizations. Therefore, the node-level performance was slightly lower compared to the node-level results presented in Sec. 8.7.2. However, this also implies that scalability might be slightly overestimated as there is more time spent computing the kernels, whereas the time required for all communication tasks is the same. Furthermore, a different approach for the connected component search was used, which is why the duration for this step is not listed. For our calculation of the duration of the experiments, we added the runtime of the connected component search as measured on Hazel Hen. We expect this to be a good approximation, as this operation is only single-socket shared-memory parallelized and Piz Daint uses a nearly identical processor.

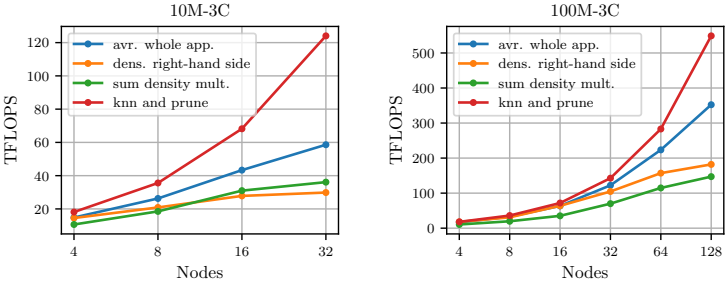
The runtimes of the experiments on Piz Daint are shown in Fig. 8.9a. On Piz Daint, the 10M-3C experiment took 100 s on 32 nodes. Compared to a baseline of four nodes, a parallel efficiency of 50% was measured. The experiment with the 100M-3C took 1198 s using 128 nodes for a parallel efficiency of 66% relative to four nodes. Compared to the Hazel Hen results, we could use four nodes instead of eight nodes as the base line due to the higher node-level performance of Piz Daint. The higher node-level performance also explains the slightly worse parallel efficiency. The GPUs in Piz Daint can complete their node-level tasks faster, which makes communication and other non-compute tasks more pronounced in the overall runtime. Furthermore, due to their massively-parallel architecture, GPUs require a higher amount of work per node to operate at their maximum performance.

Another effect of the higher node-level performance is the overall higher performance compared to Hazel Hen. On Piz Daint, a performance of 59 TFLOPS was measured for the 10M-3C dataset and 32 nodes. For the larger 100M-3C dataset and 128 nodes, 352 TFLOPS were measured. Figure 8.9b displays the performance for the varied number of nodes and further illustrates that, similar to the Hazel Hen results, the density estimation offers too little work for the individual nodes at the higher node counts. On the other hand, the performance of the creation of the k-nearest-neighbor graph-creation and the embedded graph pruning still scales nearly linearly. This step achieved a performance of 549 TFLOPS or 49% of the peak performance of 128 nodes. For the whole clustering algorithm, the performance at the highest node count corresponds to 29% of the device peak performance.

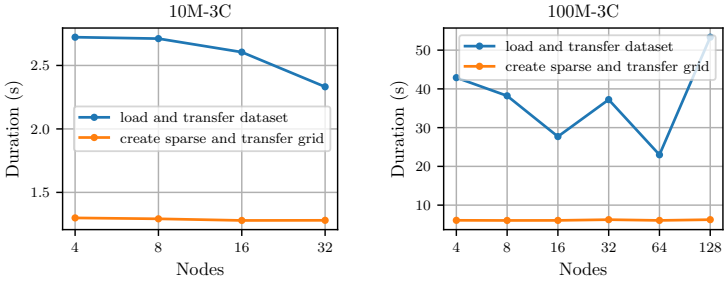
Figure 8.9c displays the time required for loading the dataset and broadcasting it to the nodes. Additionally, it shows the time required for creating the sparse grid and



(a) The durations in seconds of the clustering experiments on Piz Daint



(b) The performance in TFLOPS of the clustering experiments on Piz Daint



(c) The time required by potentially costly operations that are not major compute kernels.

Figure 8.9.: Strong scaling results for the 10M-3C and 100M-3C datasets on Piz Daint

Type	Dataset	Configuration
Support refinement	G5D-100C-D5	
Support refinement	G5D-100C-D10	$t_{\text{supp}} = 800, n_{\max} = 15$
Support refinement	G5D-100C-D20	
W.-Support coarsen	G5D-100C-D5	$t_{\text{ws}} = 7.5 \cdot 10^{-4}, n = 9$
W.-Support coarsen	G5D-100C-D10	$t_{\text{ws}} = 10^{-4}, n = 11$
Surplus refinement	G5D-100C-D5	$r_p = 800$, $r_s = 5, n = 7$

Table 8.7.: Configuration of the refinement and coarsening criteria for the G5D datasets.

transferring it to the nodes. Similar to the results from Hazel Hen, the time required is only a small fraction of the overall runtime. Fluctuations in the time required to load the dataset are explained by the high-performance filesystem being shared with other users.

8.7.4. Spatial Adaptivity for Higher Dimensions

So far, we only considered regular sparse grids for clustering. As the last step of the evaluation of sparse grid clustering, we show that a spatially-adaptive sparse grid for the density estimation enables the targeting of higher-dimensional datasets at acceptable runtimes. We compare three approaches to spatial adaptivity: support-based refinement, weight-support coarsening and surplus refinement. To highlight the differences between the adaptivity approaches, we use three datasets with 5-dimensional Gaussians embedded in a 5-dimensional, 10-dimensional and 20-dimensional dataset. The construction of these datasets and the parameters of the clustering algorithm, except for those related to adaptivity, were described in Sec. 8.7.1. As an illustration we show projections of the G5-100C-10D dataset in Fig. 8.10. All results presented in this section were computed on a Quadro GP100.

The configurations of the refinement and coarsening criteria are listed in Tab. 8.7. The parameters were chosen so that an ARI ≥ 0.99 was achieved in all cases. An attempt was made to configure the adaptivity criteria so that grid points were not added unnecessarily. Figure 8.11 shows the results of the refinement and coarsening experiments. We only display the time required for the density estimation, as the effect on the duration of the other steps is minimal.

The support refinement approach achieves the best runtimes for the more difficult datasets with 10 and 20 dimensions. It required more time to create the grid, as it iterates over the dataset to add grid points. However, the obtained grid is small enough so that

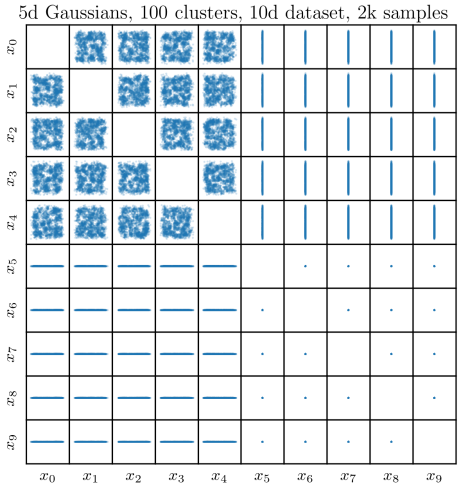


Figure 8.10.: Projections onto the coordinate planes of the G5D-100C-D10 dataset. This dataset was created by sampling 100 5d Gaussians, the additional five dimensions were set to 0.5.

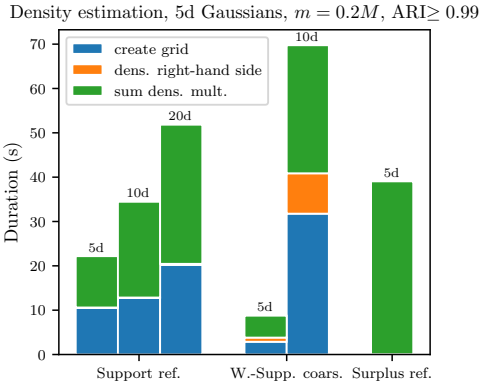


Figure 8.11.: Duration of the density estimation of a 5d structure embedded in a 5d, 10d, and 20d dataset for three different approaches for spatial adaptivity. Support refinement was the only approach that could process the 20d dataset within a reasonable timeframe. The missing results for weight-support coarsening approach and surplus are due to excessive runtimes.

solving the system of linear equations for the density estimation is fast. This approach was the only refinement approach that could successfully process the G5D-100C-D20 dataset. Furthermore, the runtime only linearly increases with the dimension.

The weight-support coarsening approach displays excellent performance for the G5D-100C-D5 dataset, as the regular grid that is coarsened is still relatively small. However, the $n = 9$ regular grid needed for the G5D-100C-D10 dataset leads to longer runtimes compared to the support refinement approach. The G5D-100C-D20 could not be processed with this approach, as the regular grid to be coarsened would be too large. This illustrates that weight-support coarsening is better suited for lower dimensional settings compared to support refinement.

The results of the coarsening approach also outline how an attempt with a regular sparse grid would fare. The G5D-100C-D5 dataset could be processed quite competitively, as only the density estimation would be slightly more expensive. However, it is more expensive for the G5D-100C-D10 dataset as well, indicating runtimes that would not be competitive with support refinement. For the G5D-100C-D20, the same reason-

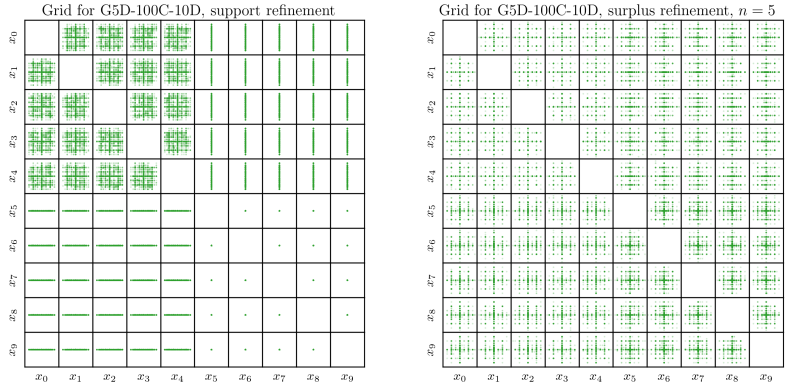


Figure 8.12.: Projections onto the coordinate planes of the refined grids created for the G5D-100C-10D datasets. Shown are the spatially-adaptive grids for support refinement (left) and surplus refinement (right). These illustrations were created from a uniformly drawn sample of 2000 grid points. In contrast to surplus refinement, the support refinement approach adapts better to the relevant dimensions.

ing as for the coarsening approach applies, i.e., the sparse grid would be too large to be processed within an acceptable timeframe.

The surplus refinement approach was slowest in processing the G5D-100C-D5 dataset. We gave up on obtaining results even for the 10-dimensional dataset, as we were unable to find a configuration that achieved an ARI ≥ 0.99 and required less than $\approx$ 5x the duration of the G5D-100C-D5 dataset. Due to the regularization and the data-point-agnostic refinement approach, surplus refinement results in a larger grid with many grid points in the non-contributing dimensions. To illustrate this issue, we show projections of sparse grids obtained through support refinement and surplus refinement for the G5D-100C-D10 dataset in Fig. 8.12. As the illustration shows, support refinement only spends grid points in the relevant dimensions.

8.8. Summary

We presented a distributed approach for clustering based on the sparse grid density estimation. By using a sparse grid density estimation, the clustering algorithm is well-suited for higher-dimensional problems. As presented, our algorithm can be used in

node-level and distributed settings and was shown to achieve excellent node-level performance in both situations. On Piz Daint, across the compute kernels a performance of 352 TFLOPS was achieved with more than 549 TFLOPS for the k-nearest-neighbor graph creation step. Furthermore, the use of OpenCL and an optimized implementation enabled performance portability across five node-level platforms from three different vendors. Finally, we showed that a spatially-adaptive sparse grid for the density estimation enables the clustering of a 20-dimensional dataset. Comparing three refinement approaches, support refinement displayed the best runtimes at a fixed ARI.

9. Conclusion

In this work, we introduced the auto-tuning framework AutoTuneTMP and applied it to three algorithms. Two further auto-tuned compute kernels are evaluated in the appendix (Sec. A.2 and Sec. B.4). We showed that AutoTuneTMP's auto-tuning approach is easy to integrate into applications, productive, maintainable and extensible. By providing templates for common optimizations, AutoTuneTMP not only tunes compute kernels, but also supports the writing of auto-tuning-enabled compute kernels. The use of JIT compilation and the approach for writing compute kernels allow AutoTuneTMP to go beyond prior approaches.

Applying our auto-tuning approach to the standard matrix multiplication algorithm, we demonstrated competitive performance with vendor BLAS libraries, achieving up to 91% of the peak performance of four devices and displaying performance portability. For this performance result, AutoTuneTMP successfully discovered high-performance parameterizations in an 11-dimensional search space.

We further auto-tuned two sparse grid regression algorithms: the subspace algorithm and the unified streaming algorithm. In evaluating these algorithms, we demonstrated that both algorithms are a strong improvement over the state-of-the-art approach and that both significantly benefit from auto-tuning. Of these two algorithms, the unified streaming algorithm proves to be more flexible as it can efficiently use GPUs. The subspace algorithm displayed higher performance than the unified streaming algorithm, but is limited to processor platforms. On a spatially-adaptive sparse grid, the subspace algorithm outperformed the streaming algorithm by a factor of up to 5.4x. While the subspace algorithm used an auto-tuned C++ implementation approach, the unified streaming algorithm showed that AutoTuneTMP can be used to auto-tune OpenCL kernels. OpenCL is a particularly relevant auto-tuning target, as it provides excellent portability. For the unified streaming algorithm, we showed that near-optimal performance was achieved across processors and GPUs. Considering the performance model provided for this algorithm, the unified streaming algorithm reached up to 96% of the achievable performance.

Our distributed sparse grid clustering algorithm contributes both to sparse grid data mining and performance portability research. Our highly-optimized OpenCL implementation achieved up to 352 TFLOPS in a strong-scaling scenario using 128 nodes of Piz Daint. In node-level scenarios, with sufficient work available per node, up to 90% of the achievable peak performance could be measured. In addition, we introduced support refinement for both sparse grid regression and sparse grid clustering. For both regression and clustering, support refinement was shown to improve performance at the same error compared to the state-of-the-art surplus refinement approach.

While there is still work to be done until auto-tuning is widely adopted, we showed in this work that auto-tuning is well-suited to solve the performance and portability challenges faced by many scientific applications. We identified JIT-compilation, which was used in all compute kernels in this work, as a key feature towards generally applicable auto-tuning. As parameters are often independent of most other parameters, we tentatively conclude that search heuristics such as line search tend to be sufficient even for large higher-dimension search spaces. Finally, we showed that performance portability is possible across major node-level architectures. Parameterized compute kernels and auto-tuning were shown to be valuable tools for bridging the gap between different architectures.

Directions for Future Research

The work presented offers many opportunities for further research. AutoTuneTMP could be extended with further search strategies and additional optimization templates. Thereby, compute kernels with different characteristics from those evaluated can be targeted. Modern hardware platforms are challenging targets for auto-tuning, to a large extent because of their power-saving features and dynamic frequency variation. Both make it difficult to obtain accurate performance measurements, which are critical for auto-tuning. This could be addressed by using the hardware performance counters to account for frequency variation. Similarly, by disabling power-saving features during tuning the impact of switching from a low-power to a higher-power performance state could be mitigated. More accurate measurements of short-running compute kernels would allow for a reduction of the problem size used during tuning. Thereby, the total duration of the tuning process could be reduced.

The presented sparse grid algorithms can be further improved as well. More work on the subspace algorithm for GPUs might yield an overall faster and lower complexity regression approach. Similarly, a lower complexity k-nearest neighbor search algo-

rithm would reduce the overall complexity of sparse grid clustering. To that end, a locality-sensitive hashing algorithm primarily intended for GPUs has already been implemented [37, 22]. While early results are promising, the suitability of this algorithm requires further investigation and a distributed variant of this algorithm needs to be designed.

A. AutoTuneTMP

A.1. Metaprogramming for OpenCL Abstractions

As another example for the usefulness of template metaprogramming, we consider a template that implements the application of kernel arguments to an OpenCL kernel. In OpenCL, applying arguments requires one API call per argument and further calls for ensuring that no error occurred. The implementation of the recursive variadic template that simplifies this task as well as an example call are shown in List. A.1. The template shown not only applies an arbitrary number of arguments to the kernel, it additionally checks whether an error occurred. In case of an error, an exception is thrown. As the types of the arguments are inferred individually, the template can be used for kernels that have arguments with different types. The last line of the example shows that this template is easy to use.

Listing A.1: A recursive variadic template for applying arguments to OpenCL kernels.

```
1  namespace detail {
2  template <int32_t> void apply_arguments(cl_kernel kernel) {}
3  template <int32_t arg_num, typename T, typename... Ts>
4  void apply_arguments(cl_kernel kernel, T arg, Ts... other_args) {
5    cl_int err = clSetKernelArg(kernel, arg_num, sizeof(T), &arg);
6    check(err, err_str);
7    detail::apply_arguments<arg_num + 1>(kernel, other_args...);
8  }
9  } // namespace detail
10 template <typename... Ts> void apply_arguments(cl_kernel kernel,
11     Ts... args) {
12   detail::apply_arguments<0>(kernel, args...);
13 }
14 // example call
15 opencl::apply_arguments(kernel, arg1, arg2, arg3);
```

A.2. Auto-Tuning the STREAM Triad

The STREAM benchmark is a benchmark that measures the throughput of four memory intensive loops and was originally proposed by John McCalpin [101]. It is still often used to assess the practically achievable maximum memory bandwidth. We employ it to further illustrate the usefulness of optimization templates for creating auto-tunable compute kernels and for the applicability of AutoTuneTMP.

For the purpose of this demonstration, we consider the `triad` routine of the STREAM benchmark. It has three input vectors $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ with $N \in \mathbb{N}$ components. Further assuming a scalar constant $q \in \mathbb{R}$, it computes $a_i := b_i + qc_i$ for all $i \in \{1, \ldots, N\}$. As each component of all of the arrays is only touched once, the operation is clearly bound by the available memory bandwidth for a large enough N.

To achieve close to the practically-achievable memory bandwidth on NUMA machines, we created a NUMA-aware implementation of the `triad` routine. This implementation is displayed in List. A.2. We use a configurable thread pool, which is part of our optimization templates collection, as well as our register-blocking template. In our implementation, each thread performs the `triad` in its own array. The functional `f` that implement the per-thread routine is called with the ID of the executing thread. This is used for the thread to access the array mapped to its NUMA node. We start measuring the performance as soon as the tasks get assigned to the thread pool. The measuring ends after the synchronization function of the thread pool returns (Line 25).

This compute kernel exposes three parameters. The first parameter is the number of threads of the thread pool. As the second parameter, the affinity policy of the thread pool can be adjusted. Two options are supported: “compact” and “spread”. The “compact” policy first fills up one NUMA node before placing threads on the second NUMA node. If the policy is set to “spread”, threads are distributed so that the number of threads per NUMA node is minimal. For a portable vectorized implementation we again rely on Vc. To additionally allow for software pipelining, we use the register-blocking template. Its blocking factor is the third parameter exposed by the compute kernel.

Our target is the AMD Epyc 7551P processor. As this processor consists of four NUMA domains, achieving optimal memory throughput is slightly more challenging than on standard dual-socket platforms. The Epyc processor has two memory channels per NUMA domain. Overall, it offers a theoretical peak memory bandwidth of 171 GB/s. The values range of the parameters are summarized in Tab. A.1. As the initial parameter values, we set the register-blocking factor and the number of threads to one. The affinity

Listing A.2: An auto-tuned STREAM `triad` that uses the register-blocking and queue-thread-pool templates. It exposes three parameters.

```
using reg_arr = opttmp::register_array<Vc::double_v, REG_BLOCKING>;
opttmp::queue_thread_pool<THREADS> pool;
pool.set_affinity(AFFINITY_POLICY);
pool.start();
auto f = [&](size_t thread_id) {
  // get per-thread array
  std::vector<double, align_alloc> &a_thread = a[thread_id];
  std::vector<double, align_alloc> &b_thread = b[thread_id];
  std::vector<double, align_alloc> &c_thread = c[thread_id];
  constexpr size_t increment = block_reg * Vc::double_v::size();
  // do per-thread triad
  for (size_t i = 0; i < N_per_task; i += increment) {
    reg_arr b_reg(&b_thread[i], Vc::vector_aligned);
    reg_arr c_reg(&c_thread[i], Vc::vector_aligned);
    temp = b_reg + q_vec * c_reg;
    temp.memstore(&a_thread[i], Vc::vector_aligned);
  }
};
auto timer_start = ...;
for (size_t i = 0; i < KERNEL_THREADS; i += 1) {
  // the thread ID is set up internally, value-initialization is
  // used as placeholder
  pool.enqueue_work_id(f, {});
}
pool.finish();
auto timer_stop = ...;
```

	Affinity policy	Threads	Reg. blocking
Values	{compact, spread}	{1, 2, 4, 8, 16, 32}	{1, 2, 4}

Table A.1.: The STREAM triad kernel exposes three parameters to be tuned.

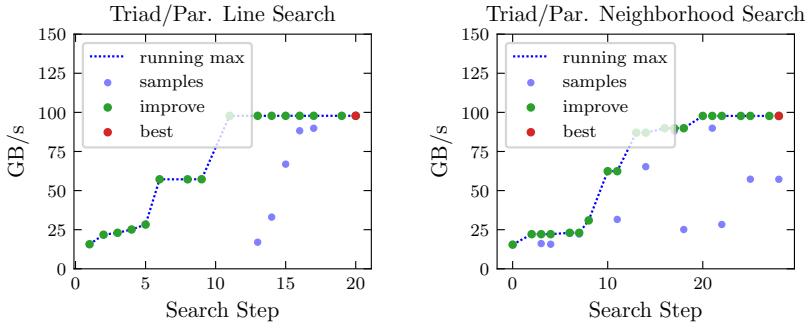


Figure A.1.: Auto-tuning the STREAM `triad` with two search strategies: parallel neighborhood search and parallel line search. Both search strategies achieved a nearly identical maximum bandwidth, though parallel neighborhood search required more search steps.

policy is set to "compact".

Results for the parallel neighborhood search and parallel line search are shown in Fig. A.1. Both search strategies achieve a maximum memory bandwidth of 98 GB/s. While both search strategies find a best parameter combination with the same performance, line search required fewer iterations. This bandwidth was computed assuming three memory references per iteration. However, as our version of Vc does not support non-temporal stores, the actual traffic between the processor and the DRAM amount to 130 GB/s. For comparison, AMD reports a slightly higher memory bandwidth of 149 GB/s for the STREAM `triad`. The highest-bandwidth run had the affinity policy set to "spread", set the number of threads to 32 and specified a register-blocking factor of 4. However, the blocking did not provide any significant speedup, whereas using at least 16 threads and the correct affinity policy were critical. This example further illustrates the applicability of AutoTuneTMP. And it shows the usefulness of auto-tuning affinity, through a parameterized thread pool abstraction, in a memory-bound case.

B. Sparse Grids

B.1. The Unified Streaming Algorithm and Double Precision

In Sec. 7.3.3, we provided auto-tuning results for the unified streaming algorithm using single-precision arithmetic. In the following, we describe auto-tuning of the unified streaming algorithm using double-precision arithmetic using the same experiments as for single-precision arithmetic.

First, we address the speedup over the PVN parameter combination. Figure B.1 shows the results of the experiments for three platforms. Fewer platforms were used, as fewer platforms support double-precision arithmetic (efficiently). Overall, speedups between 1.5x and 4.9x were achieved. The results for the Tesla P100 and the FirePro W8100 GPUs are in line with the single-precision results. On the Vega VII GPU, auto-tuning still improves performance, but less so compared to the single-precision results. The Vega VII GPU only offers quarter-rate double-precision performance. Due to the thereby improved machine balance, memory is less of a bottleneck and there is less of an opportunity for improved performance.

Next, we consider the contribution of individual parameters. For consistency, we performed the same PVN comparison as described in Sec. 7.3.3. Figure B.2 shows the results of the experiments for the DR5-Supp-D and Fried1-D experiments. Generally, the same parameters contribute and the contribution are similar compared to the single-precision experiments. The differences between the two setups are again explained by the different sizes of sparse grids. We show the best parameter values obtained through auto-tuning for the Tesla P100 in the next section.

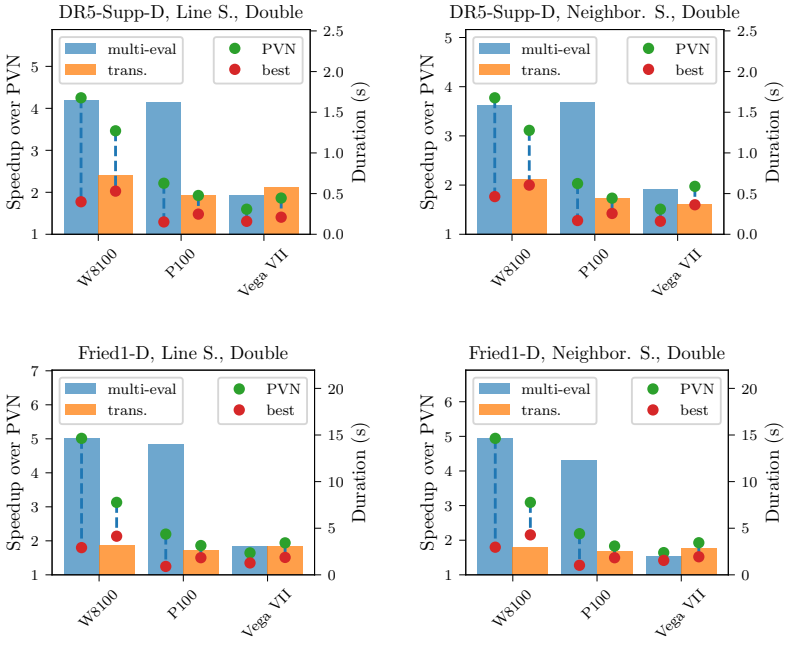


Figure B.1.: The performance improvements of auto-tuning the unified streaming algorithm for both datasets (upper) and two search strategies: line search (left) and neighborhood search (right). In contrast to Sec. 7.3.3, double-precision arithmetic was used. Speedups between 1.5x and 4.9x were obtained over PVN initial parameter values (bars). The durations shown are for a single operation execution (dots).

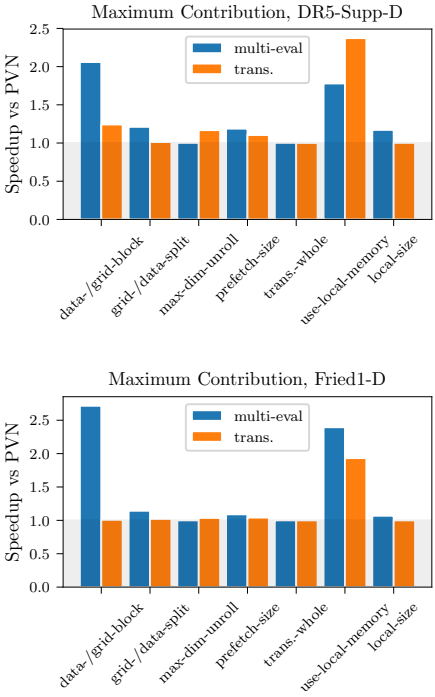


Figure B.2.: Contribution of individual parameter to the performance of the best parameter combination obtained through line search in the double-precision experiments. The graph shows the maximum benefit of a parameter for any of the hardware platforms used. Results are shown for the DR5-Supp-D setup (top) and the Fried1-D setup (bottom).

Table B.1.: Auto-tuned parameter values of the unified streaming algorithm for the P100 platform and the DR5 dataset.

(a) Exp.: DR5-Supp-S

Operator	Multi-Evaluation			Transposed		
Strategy	Line	Neighbor	Monte C.	Line	Neighbor	Monte C.
transfer-whole-dataset/ transfer-whole-grid	True	True	True	True	True	True
data-block/grid-block	8	1	8	1	1	2
grid-split/data-split	8	8	8	1	1	8
local-size	256	256	64	64	64	256
max-dim-unroll	2	1	1	4	1	4
prefetch-size	64	32	32	64	16	128
use-local-memory	True	True	True	True	True	True

(b) Exp.: DR5-Supp-D

Operator	Multi-Evaluation			Transposed		
Strategy	Line	Neighbor	Monte C.	Line	Neighbor	Monte C.
transfer-whole-dataset/ transfer-whole-grid	True	True	True	False	True	False
data-block/grid-block	8	4	8	1	1	1
grid-split/data-split	8	1	8	8	4	4
local-size	64	128	64	256	64	128
max-dim-unroll	1	4	1	10	1	1
prefetch-size	64	16	64	16	64	32
use-local-memory	True	True	True	True	True	True

B.2. Auto-Tuned Parameter Values of the Unified Streaming Algorithm

To give an idea of the parambeter values obtained through auto-tuning of the unified streaming algorithm, we provide the best parameter values for the three search strategies used in the experiments. We limit ourselves to the Tesla P100 platforms due to the high number of parameter values per device. Table B.1 lists the best parameters discovered for the DR5 experiments. In Tab. B.3, the same is shown for the Friedman1 experiments.

Table B.3.: Auto-tuned parameter values of the unified streaming algorithm for the P100 platform and the Friedman1 dataset.

(a) Exp.: Fried1-S

Operator	Multi-Evaluation			Transposed		
Strategy	Line	Neighbor	Monte C.	Line	Neighbor	Monte C.
transfer-whole-dataset/ transfer-whole-grid	True	True	True	False	False	True
data-block/grid-block	8	8	8	8	1	8
grid-split/data-split	8	8	8	4	1	8
local-size	256	256	128	256	256	256
max-dim-unroll	10	2	10	1	1	1
prefetch-size	128	128	32	32	32	64
use-local-memory	True	True	True	True	True	True

(b) Exp.: Fried1-D

Operator	Multi-Evaluation			Transposed		
Strategy	Line	Neighbor	Monte C.	Line	Neighbor	Monte C.
transfer-whole-dataset/ transfer-whole-grid	True	True	True	True	True	False
data-block/grid-block	8	8	8	2	2	4
grid-split/data-split	8	1	8	2	4	8
local-size	64	64	64	64	64	128
max-dim-unroll	10	2	1	1	2	10
prefetch-size	64	64	32	16	16	128
use-local-memory	True	True	True	True	True	True

B.3. Auto-Tuned Parameter Values of the Subspace Algorithm

In this section, we provide the best auto-tuned parameter values discovered through auto-tuning of the subspace algorithm using three search strategies. Similar to the last section, we again limit ourselves to a single device due to the amount of results. We show the parameters of the dual-socket Xeon Gold 5120 platform. Table B.5 and Tab. B.6 show the best parameter values for the DR5-Supp-D and Fried1-D experiments, respectively.

Table B.5.: Auto-tuned parameter values for the DR5-Supp-D experiment on the dual-socket Xeon Gold platform.

Operator	Multi-Evaluation			Transposed		
Strategy	Monte C.	Neighbor	Line	Monte C.	Neighbor	Line
reuse-intermediates	1	1	1	1	1	1
enable-subspace-skipping	0	0	0	1	0	0
list-ratio	0.3	0.1	0.2	0.1	0.1	0.3
chunk-size	512	512	512	512	512	512
unroll-vectorization	1	1	1	1	0	0
vector-padding	4	4	4	8	4	4

Table B.6.: Auto-tuned parameter values for the Fried1-D experiment on the dual-socket Xeon Gold platform.

Operator	Multi-Evaluation			Transposed		
Strategy	Monte C.	Neighbor	Line	Monte C.	Neighbor	Line
reuse-intermediates	1	1	1	1	1	1
enable-subspace-skipping	1	1	0	1	0	0
list-ratio	0.2	0.1	0.1	0.3	0.1	0.2
chunk-size	128	64	64	256	16	256
unroll-vectorization	1	0	0	0	0	1
vector-padding	4	4	4	8	4	4

Listing B.1: A template for converting an array from array-of-structs into a struct-of-arrays representation.

```
// std::vector<double> from AoS (abcabcabc) to SoA (aaabbbccc)
opttmp::struct_of_array_data<DIMS, ENTRIES> data_SoA(data);
// pointer to component, for continuous access in same dimension
double *double = data_SoA.pointer(d, i);
```

B.4. An Auto-Tunable High-Level Multi-Evaluation

As a further demonstration of AutoTuneTMP's optimization template collection, we consider a common sparse grid operation. Following the approach described in Sec. 7.1, we implemented a streaming multi-evaluation operator for linear basis functions.

For our implementation, we again use OpenMP for parallelization, Vc for vectorization and the register-blocking template from AutoTuneTMP's optimization template collection. For a vectorized algorithm that does not require (simulated) gather instructions, we convert the dataset, as read from a file, from array-of-struct into struct-of-array representation. This is implemented by yet another template of our template collection. An example of how this template is used is shown in List. B.1.

The implementation of this compute kernel is displayed in List. B.2. For parallelization, the dataset is split into chunks processed by individual threads. Similar to the OpenCL kernels, each thread processes a chunk of data points at a time. The chunk-size is controlled by the *block-data* parameter and was realized through the `register_array` class template. By using this class and Vc, all floating-point operations in the inner loops are perfectly vectorized. Per iteration of the innermost loop, only two scalar memory references (for reading `level` and `index`) are required. For a SIMD-width s, there are therefore $6 \cdot s \cdot \textit{block-data}$ operations per 16 B read from memory. Consequently, for larger blocking sizes the algorithm quickly becomes compute-bound. For auto-tuning, the blocking parameter was set to a value range from one to 15. As the second parameter, we once again allow the auto-tuner to enable or disable SMT by choosing the appropriate number of threads for the parallel region. Given these two parameters, the associated optimization task has a low difficulty. However, the implementation itself provides further evidence for how an optimized implementation can closely mirror a scalar implementation given adequate high-level optimization tools.

To evaluate this algorithm, we used the synthetic Friedman1 dataset as described in Fig. 7.3.1. Thus, the dimensionality of the multi-evaluation task is 10. To fully saturate a processor-node, we generated a dataset with $2 \cdot 10^5$ data points. Furthermore, we chose

Listing B.2: An auto-tuned multi-evaluation algorithm for linear basis functions implemented with AutoTuneTMP's optimization templates.

```
using namespace opttmp::vectorization;
using reg_array = register_array<double_v, DATA_BLOCKING>;
double_v one(1.0); double_v zero(0.0);
constexpr size_t step = DATA_BLOCKING * double_v::size();

#pragma omp parallel for num_threads(OMP_THREADS)
for (size_t i = 0; i < dataset_size; i += step) {
  reg_array result(0.0);
  for (size_t j = 0; j < alpha.size(); j += 1) {
    reg_array eval(alpha[j]);
    for (size_t d = 0; d < DIMS; d += 1) {
      reg_array data(data_SoA.pointer(d, i), Vc::vector_aligned);
      double_v level(levels[j * DIMS + d]);
      double_v index(indices[j * DIMS + d]);
      // abs-expression converted to DATA_BLOCKING-many FMAs
      eval *= max(one - abs(data * level - index), zero);
    }
    result += eval;
  }
  result.memstore(&results[i], Vc::vector_aligned);
}
```

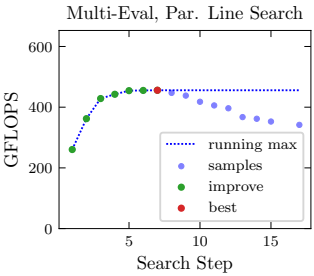


Figure B.3.: Auto-tuning the sparse grid multi-evaluation implemented with the optimization template collection. Due to the essentially one-dimensional search problem, a high-performance parameterization is detected easily. The best parameterization reaches 93% of the achievable performance.

a regular sparse grid of level 7.

We tuned the compute kernel on an AMD Epyc 7551P processor with parallel line search as the search strategy. The results of this auto-tuning experiment are shown in Fig. B.3. In this experiment, we achieved up to 456 GFLOPS or 70% of the peak performance of the device. On this hardware platform, the vector units impose an upper bound of 75%. We therefore achieve 93% of the achievable peak performance. The upper bound is computed by taking into consideration that the FMA instruction counts for two operations and the max (implemented as `AND`) and the subtraction can be executed simultaneously, as they occupy different pipelines [53]. The best parameterization had SMT enabled and chose *block-data* with seven.

Bibliography

[1] *"Vega" Instruction Set Architecture*. AMD. July 2017. URL: https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf.

[2] Bilge Acun et al. „Parallel Programming with Migratable Objects: Charm++ in Practice". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'14. New Orleans, Louisana: IEEE Press, 2014, pp. 647–658. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.58.

[3] Jennifer K. Adelman-McCarthy et al. „The Fifth Data Release of the Sloan Digital Sky Survey". In: *The Astrophysical Journal Supplement Series* 172.2 (2007), pp. 634–644.

[4] *AMD APP SDK OpenCL Optimization Guide*. AMD. Aug. 2015. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf.

[5] Guilherme Andrade et al. „G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering". In: *Procedia Computer Science* 18 (2013), pp. 369–378.

[6] Jason Ansel et al. „OpenTuner: An Extensible Framework for Program Autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Aug. 2014, pp. 303–315.

[7] Jason Ansel et al. „PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'09. Dublin, Ireland: ACM, 2009, pp. 38–49. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542481.

[8] Hartwig Anzt et al. „Experiences in autotuning matrix multiplication for energy minimization on GPUs". In: *Concurrency and Computation: Practice and Experience* 27.17 (May 2015), pp. 5096–5113. DOI: 10.1002/cpe.3516.

[9] David Arthur and Sergei Vassilvitskii. „K-means++: The Advantages of Careful Seeding“. In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA'07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035. ISBN: 978-0-898716-24-5.

[10] Cédric Augonnet et al. „StarPU: a unified platform for task scheduling on heterogeneous multicore architectures“. In: *Concurrency and Computation: Practice and Experience* 23.2 (Nov. 2010), pp. 187–198. DOI: 10.1002/cpe.1631.

[11] Bahman Bahmani et al. „Scalable K-Means++“. In: *Proc. VLDB Endow.* 5.7 (Mar. 2012), pp. 622–633. ISSN: 2150-8097.

[12] E. Bajrovic et al. „Tuning OpenCL Applications with the Periscope Tuning Framework“. In: *2016 49th Hawaii International Conference on System Sciences (HICSS)*. Jan. 2016, pp. 5752–5761. DOI: 10.1109/HICSS.2016.711.

[13] M. Bauer et al. „Legion: Expressing locality and independence with logical regions“. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[14] David Beckingsale et al. „Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code“. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 307–316. DOI: 10.1109/IPDPS.2017.38.

[15] Nathan Bell and Jared Hoberock. „Thrust: A productivity-oriented library for CUDA“. In: *GPU computing gems Jade edition* 2 (2011), pp. 359–371.

[16] Richard Bellman. *Adaptive Control Processes: A Guided Tour*. 'Rand Corporation. Research studies. Princeton University Press, 1961.

[17] Janki Bhimani, Miriam Leeser, and Ningfang Mi. „Accelerating K-Means Clustering with Parallel Implementations and GPU Computing“. In: *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE. 2015, pp. 1–6.

[18] Jeff Bilmes et al. *The PHiPAC v1.0 matrix-multiply distribution*. Tech. rep. University of California at Berkeley, Oct. 1998.

[19] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006. ISBN: 0387310738.

[20] Robert D. Blumofe et al. „Cilk: An Efficient Multithreaded Runtime System“. In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69. ISSN: 0743-7315. DOI: 10.1006/jpdc.1996.0107.

[21] Christian Böhm et al. „Density-based Clustering Using Graphics Processors“. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. CIKM'09. Hong Kong, China: ACM, 2009, pp. 661–670. ISBN: 978-1-60558-512-3.

[22] Marcel Breyer. „Ein hoch-performanter (approximierter) k-Nächste-Nachbarn Algorithmus für GPU“. Bachelor's Thesis. University of Stuttgart, Apr. 2018.

[23] Hans-Joachim Bungartz and Michael Griebel. „Sparse Grids“. In: *Acta Numerica* 13 (2004), pp. 1–123.

[24] Hans-Joachim Bungartz, Dirk Pflüger, and Stefan Zimmer. „Adaptive Sparse Grid Techniques for Data Mining“. In: *Modelling, Simulation and Optimization of Complex Processes 2006, Proc. Int. Conf. HPSC, Hanoi, Vietnam*. Ed. by H.G. Bock et al. Springer-Verlag, Aug. 2008, pp. 121–130. ISBN: 9783540794080.

[25] Gerrit Buse. „Exploiting Many-Core Architectures for Dimensionally Adaptive Sparse Grids“. PhD thesis. München: Institut für Informatik, Technische Universität München, May 2015. ISBN: 9783843920926.

[26] Antal Buss et al. „STAPL: Standard Template Adaptive Parallel Library“. In: *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. SYSTOR'10. Haifa, Israel: ACM, 2010, 14:1–14:10. ISBN: 978-1-60558-908-4. DOI: 10.1145/1815695.1815713.

[27] Jean-Sylvain Camier. „Improving Performance Portability and Exascale Software Productivity with the ∇ Numerical Programming Language“. In: *Proceedings of the 3rd International Conference on Exascale Applications and Software*. EASC'15. Edinburgh, UK: University of Edinburgh, 2015, pp. 126–131. ISBN: 978-0-9926615-1-9.

[28] Chih-Chung Chang and Chih-Jen Lin. „LIBSVM: A Library for Support Vector Machines“. In: *ACM Trans. Intell. Syst. Technol.* 2.3 (May 2011), 27:1–27:27. ISSN: 2157-6904. DOI: 10.1145/1961189.1961199.

[29] Chun Chen. „Model-guided empirical optimization for memory hierarchy“. PhD thesis. 2007.

[30] Ray S. Chen and Jeffrey K. Hollingsworth. „Towards fully automatic auto-tuning: Leveraging language features of Chapel“. In: *The International Journal of High Performance Computing Applications* 27.4 (2013), pp. 394–402. DOI: 10.1177/1094342013493198.

[31] M. Christen, O. Schenk, and H. Burkhart. „PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures“. In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 676–687. DOI: 10.1109/IPDPS.2011.70.

[32] G. C. Clayton et al. „Very Large Excesses of ^{18}O in Hydrogen-deficient Carbon and R Coronae Borealis Stars: Evidence for White Dwarf Mergers“. In: *The Astrophysical Journal* 662 (June 2007), pp. 1220–1230. DOI: 10.1086/518307.

[33] *clBLAS*. https://github.com/clMathLibraries/clBLAS. Accessed: 2018-10-17.

[34] Don Coppersmith and Shmuel Winograd. „Matrix Multiplication via Arithmetic Progressions“. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 1–6. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28396.

[35] Gregor Daiß et al. „From Piz Daint to the Stars: Simulation of Stellar Mergers using High-Level Abstractions“. In: SC'19 (2019). accepted.

[36] Usman Dastgeer, Johan Enmyren, and Christoph W Kessler. „Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems“. In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. ACM. 2011, pp. 25–32.

[37] Mayur Datar et al. „Locality-Sensitive Hashing Scheme Based on P-stable Distributions“. In: *Proceedings of the Twentieth Annual Symposium on Computational Geometry*. SCG'04. Brooklyn, NY, USA: ACM, 2004, pp. 253–262. ISBN: 1-58113-885-7.

[38] Kaushik Datta et al. „Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures“. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press. 2008.

[39] Zachary DeVito et al. „Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers“. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'11. Seattle, Washington: ACM, 2011, 9:1–9:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063396.

[40] Romain Dolbeau, François Bodin, and Guillaume Colin de Verdière. „One OpenCL to rule them all?“ In: *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems*. IEEE, Sept. 2013, pp. 1–6.

[41] J. Dongarra et al. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979. DOI: 10.1137/1.9781611971811.

[42] Peng Du et al. „From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming“. In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[43] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. „Kokkos: Enabling manycore performance portability through polymorphic memory access patterns“. In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[44] Johan Enmyren and Christoph W. Kessler. „SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems“. In: *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*. HLPP'10. Baltimore, Maryland, USA: ACM, 2010, pp. 5–14. ISBN: 978-1-4503-0254-8. DOI: 10.1145/1863482.1863487.

[45] Martin Ester et al. „A Density-Based Algorithm for Discovering Clusters“. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[46] Pierre Estérie et al. „Boost.SIMD: Generic Programming for Portable SIMDization“. In: *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. WPMVP'14. Orlando, Florida, USA: ACM, 2014, pp. 1–8. ISBN: 978-1-4503-2653-7. DOI: 10.1145/2568058.2568063.

[47] Vladimir Estivill-Castro. „Why So Many Clustering Algorithms: A Position Paper“. In: *SIGKDD Explor. Newsl.* 4.1 (June 2002), pp. 65–75. ISSN: 1931-0145.

[48] Andrew Fall and Joseph Fall. „A domain-specific language for models of landscape dynamics“. In: *Ecological Modelling* 141.1 (2001), pp. 1–18. ISSN: 0304-3800. DOI: 10.1016/S0304-3800(01)00334-9.

[49] J. Fang, A. L. Varbanescu, and H. Sips. „A Comprehensive Performance Comparison of CUDA and OpenCL“. In: *2011 International Conference on Parallel Processing*. Sept. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45.

[50] Wenbin Fang et al. *Parallel Data Mining on Graphics Processors*. Tech. rep. Hong Kong Univ. Sci. and Technology, Hong Kong, China, 2008.

[51] Reza Farivar et al. „A Parallel Implementation of K-Means Clustering on GPUs“. English (US). In: *Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008*. 2008, pp. 340–345. ISBN: 1601320841.

[52] Kayvon Fatahalian et al. „Sequoia: Programming the Memory Hierarchy“. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC'06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188543.

[53] Agner Fog. *Instruction tables*. Tech. rep. Technical University of Denmark, 2018.

[54] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. Tech. rep. Technical University of Denmark, 2017.

[55] Franz Franchetti et al. „SPIRAL: Extreme Performance Portability“. In: *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106.11 (2018). preprint.

[56] Jerome H. Friedman. „Multivariate Adaptive Regression Splines“. In: *The Annals of Statistics* 19.1 (1991), pp. 1–67.

[57] Matteo Frigo and Steven G. Johnson. „FFTW: an adaptive software architecture for the FFT“. In: *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*. Vol. 3. May 1998, pp. 1381–1384. DOI: 10.1109/ICASSP.1998.681704.

[58] Junhao Gan and Yufei Tao. „DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD'15. Melbourne, Victoria, Australia: ACM, 2015, pp. 519–530. ISBN: 978-1-4503-2758-9.

[59] J. Garcke, M. Griebel, and M. Thess. „Data Mining with Sparse Grids“. In: *Computing* 67.3 (2001), pp. 225–253. ISSN: 0010-485X. DOI: 10.1007/s006070170007.

[60] Jochen Garcke. „Maschinelles Lernen durch Funktionsrekonstruktion mit verallgemeinerten dünnen Gittern“. PhD thesis. Universität Bonn, Institut für Numerische Simulation, 2004.

[61] Pawel Gepner, Victor Gamayunov, and David L. Fraser. „Effective Implementation of DGEMM on Modern Multicore CPU“. In: *Procedia Computer Science* 9 (2012). Proceedings of the International Conference on Computational Science, ICCS 2012, pp. 126–135. ISSN: 1877-0509. DOI: 10.1016/j.procs.2012.04.014.

[62] Simon Garcia De Gonzalo et al. „Revisiting Online Autotuning for Sparse-Matrix Vector Multiplication Kernels on Next-Generation Architectures“. In: *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Dec. 2017, pp. 72–80. DOI: 10.1109/HPCC-SmartCity-DSS.2017.10.

[63] Kazushige Goto and Robert A. van de Geijn. „Anatomy of High-performance Matrix Multiplication“. In: *ACM Trans. Math. Softw.* 34.3 (May 2008), 12:1–12:25. ISSN: 0098-3500. DOI: 10.1145/1356052.1356053.

[64] Scott Grauer-Gray et al. „Auto-tuning a High-Level Language Targeted to GPU Codes“. In: *Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–10.

[65] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Redmond, WA, USA: Microsoft Press, 2012.

[66] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. „A Family of High-Performance Matrix Multiplication Algorithms“. In: *Computational Science — ICCS 2001*. Ed. by Vassil N. Alexandrov et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 51–60. ISBN: 978-3-540-45545-5.

[67] Tobias Gysi et al. „STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC'15. Austin, Texas: ACM, 2015, 41:1–41:12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807627.

[68] Albert Hartono et al. „Parametric Multi-level Tiling of Imperfectly Nested Loops“. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS'09. Yorktown Heights, NY, USA: ACM, 2009, pp. 147–157. ISBN: 978-1-60558-498-0. DOI: 10.1145/1542275.1542301.

[69] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. 2nd ed. Springer Series in Statistics. Springer-Verlag New York, 2009. ISBN: 978-0-387-84858-7.

[70] Yaobin He et al. „MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data“. In: *Frontiers of Computer Science* 8.1 (2014), pp. 83–99.

[71] Markus Hegland, Giles Hooker, and Stephen Roberts. „Finite Element Thin Plate Splines in Density Estimation“. In: *ANZIAM Journal* 42 (2000), pp. 712–734.

[72] Alexander Heinecke. „Boosting Scientific Computing Applications through Leveraging Data Parallel Architectures“. PhD thesis. Technische Universität München, Jan. 2014.

[73] Alexander Heinecke and Dirk Pflüger. „Emerging Architectures Enable to Boost Massively Parallel Data Mining Using Adaptive Sparse Grids“. In: *International Journal of Parallel Programming* 41.3 (July 2012), pp. 357–399. ISSN: 1573-7640.

[74] Alexander Heinecke and Dirk Pflüger. „Multi- and Many-core Data Mining with Adaptive Sparse Grids“. In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. CF '11. Ischia, Italy: ACM, 2011, 29:1–29:10. ISBN: 978-1-4503-0698-0. DOI: 10.1145/2016604.2016640.

[75] Alexander Heinecke et al. „Demonstrating Performance Portability of a Custom OpenCL Data Mining Application to the Intel Xeon Phi Coprocessor“. In: *International Workshop on OpenCL Proceedings 2013*. Georgia Tech, May 2013.

[76] Alexander Heinecke et al. „Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor“. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 126–137. DOI: 10.1109/IPDPS.2013.113.

[77] Thomas Heller et al. „Harnessing Billions of Tasks for a Scalable Portable Hydrodynamic Simulation of the Merger of Two Stars“. In: *The International Journal of High Performance Computing Applications (IJHPCA)* (2018). accepted.

[78] J. A. Herdman et al. „Accelerating Hydrocodes with OpenACC, OpenCL and CUDA“. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Nov. 2012, pp. 465–471. DOI: 10.1109/SC.Companion.2012.66.

[79] Alexander Hinneburg and Hans-Henning Gabriel. „DENCLUE 2.0: Fast Clustering Based on Kernel Density Estimation“. In: *Proceedings of the 7th International Conference on Intelligent Data Analysis*. IDA'07. Ljubljana, Slovenia: Springer-Verlag, 2007, pp. 70–80. ISBN: 978-3-540-74824-3.

[80] Richard D Hornung and Jeffrey A Keasler. *The RAJA Portability Layer: Overview and Status*. Tech. rep. Livermore, California: Lawrence Livermore National Laboratory, Sept. 2014.

[81] Lawrence Hubert and Phipps Arabie. „Comparing partitions“. In: *Journal of Classification* 2.1 (Dec. 1985), pp. 193–218. ISSN: 1432-1343. DOI: 10.1007/BF01908075.

[82] *Intel Xeon Phi Processor Software - Optimization Guide*. Intel. Apr. 2017.

[83] Liheng Jian et al. „Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)“. In: *The Journal of Supercomputing* 64.3 (June 2013), pp. 942–967. ISSN: 1573-0484.

[84] H. Jordan et al. „A Multi-Objective Auto-Tuning Framework for Parallel Codes“. In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–12. DOI: 10.1109/SC.2012.7.

[85] Hartmut Kaiser et al. „HPX: A Task Based Programming Model in a Global Address Space“. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS'14. Eugene, OR, USA: ACM, 2014, pp. 61–611. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883.

[86] Laxmikant V. Kale and Sanjeev Krishnan. „CHARM++: A Portable Concurrent Object Oriented System Based on C++“. In: *SIGPLAN Not.* 28.10 (Oct. 1993), pp. 91–108. ISSN: 0362-1340. DOI: 10.1145/167962.165874.

[87] Shoaib Kamil et al. „An Auto-Tuning Framework for Parallel Multicore Stencil Computations“. In: *2010 IEEE International Symposium on Parallel & Distributed Processing*. Apr. 2010, pp. 1–12.

[88] T. Kanungo et al. „An Efficient k-Means Clustering Algorithm: Analysis and Implementation“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.7 (July 2002), pp. 881–892. ISSN: 0162-8828.

[89] Paul Klint, Tijs van der Storm, and Jurgen Vinju. „RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation“. In: *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28.

[90] Kazuhiko Komatsu et al. „Evaluating Performance and Portability of OpenCL Programs“. In: iWAPT 2010. 2010.

[91] Matthias Kretz. „Extending C++ for explicit data-parallel programming via SIMD vector types“. PhD thesis. 2015.

[92] Matthias Kretz and Volker Lindenstruth. „Vc: A C++ library for explicit vectorization“. In: *Software: Practice and Experience* 42.11 (2012), pp. 1409–1430. ISSN: 1097-024X. DOI: 10.1002/spe.1149.

[93] Akhilesh Kumar and Malay Trivedi. *Intel Xeon Scalable Processor Architecture Deep Dive*. June 2017.

[94] S. Lee and J. S. Vetter. „OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study“. In: *2014 First Workshop on Accelerator Programming using Directives*. Nov. 2014, pp. 1–11. DOI: 10.1109/WACCPD.2014.7.

[95] Yinan Li, Jack Dongarra, and Stanimire Tomov. „A Note on Auto-tuning GEMM for GPUs“. In: *Computational Science–ICCS 2009*. Springer, 2009, pp. 884–892.

[96] Tze Meng Low et al. „Analytical Modeling Is Enough for High-Performance BLIS“. In: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016), 12:1–12:18. ISSN: 0098-3500. DOI: 10.1145/2925987.

[97] P. Luszczek et al. „Search Space Generation and Pruning System for Autotuners“. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IPDPS'16. May 2016, pp. 1545–1554. DOI: 10.1109/IPDPSW.2016.197.

[98] Thibaut Lutz, Christian Fensch, and Murray Cole. „PARTANS: An Autotuning Framework for Stencil Computation on Multi-GPU Systems“. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (Jan. 2013), 59:1–59:24. ISSN: 1544-3566. DOI: 10.1145/2400682.2400718.

[99] Ulrike von Luxburg. „A tutorial on spectral clustering“. In: *Statistics and Computing* 17.4 (Dec. 2007), pp. 395–416. ISSN: 1573-1375.

[100] Maximilian Luz. „Subspace-Optimal Data Mining on Spatially-Adaptive Sparse Grids“. Bachelor's Thesis. University of Stuttgart, Nov. 2017.

[101] John D. McCalpin. „Memory Bandwidth and Machine Balance in Current High Performance Computers“. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* 2 (Dec. 1995), pp. 19–25.

[102] John D. McCalpin. „Memory Bandwidth and System Balance in HPC Systems“. In: *2016 ACM/IEEE Conference on Supercomputing, Invited talk* (2016). URL: https://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/.

[103] Simon McIntosh-Smith et al. „On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures“. In: *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*. ISC'14. Leipzig, Germany: Springer-Verlag New York, Inc., 2014, pp. 53–75. ISBN: 978-3-319-07517-4. DOI: 10.1007/978-3-319-07518-1_4.

[104] Renato Miceli et al. „AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications“. In: *Applied Parallel and Scientific Computing*. Ed. by Pekka Manninen and Per Öster. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 328–342. ISBN: 978-3-642-36803-5.

[105] *MPI: A Message-Passing Interface Standard*. Version 3.1. Message Passing Interface Forum. June 2015. URL: https://www.mpi-forum.org/docs/.

[106] *MPICH*. Accessed: 2018-11-13. 2018. URL: https://www.mpich.org/.

[107] G. R. Mudalige et al. „OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures“. In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339594.

[108] Alin Murarașu et al. „Fastsg: A Fast Routines Library for Sparse Grids“. English. In: *Procedia Computer Science* 9.Complete (2012), pp. 354–363.

[109] Cedric Nugteren. „CLBlast: A Tuned OpenCL BLAS Library“. In: *Proceedings of the International Workshop on OpenCL*. IWOCL'18. Oxford, United Kingdom: ACM, 2018, 5:1–5:10. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204924.

[110] *Nvidia Tesla P100*. Tech. rep. (Whitepaper). Nvidia, 2016.

[111] *Nvidia Tesla V100 GPU Architecture*. Tech. rep. (Whitepaper). Nvidia, 2017.

[112] *OpenMP Application Program Interface, Version 4.0*. Version 4.0. OpenMP Architecture Review Board. July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[113] *OpenMPI*. Accessed: 2018-11-13. 2018. URL: https://www.open-mpi.org/.

[114] Benjamin Peherstorfer, Dirk Pflüger, and Hans-Joachim Bungartz. „Clustering Based on Density Estimation with Sparse Grids“. In: *KI 2012: Advances in Artificial Intelligence*. Ed. by Birte Glimm and Antonio Krüger. Vol. 7526. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 131–142. ISBN: 978-3-642-33346-0.

[115] S. J. Pennycook and S. A. Jarvis. „Developing Performance-Portable Molecular Dynamics Kernels in OpenCL“. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Nov. 2012, pp. 386–395. DOI: 10.1109/SC.Companion.2012.58.

[116] S.J. Pennycook, J.D. Sewall, and V.W. Lee. „Implications of a metric for performance portability“. In: *Future Generation Computer Systems* (2017). ISSN: 0167-739X. DOI: 10.1016/j.future.2017.08.007.

[117] S.J. Pennycook et al. „An investigation of the performance portability of OpenCL“. In: *Journal of Parallel and Distributed Computing* 73.11 (2013), pp. 1439–1450. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2012.07.005.

[118] David Pfander, Malte Brunn, and Dirk Pflüger. „AutoTuneTMP: Auto-Tuning in C++ With Runtime Template Metaprogramming“. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 1123–1132. DOI: 10.1109/IPDPSW.2018.00172.

[119] David Pfander, Gregor Daiß, and Dirk Pflüger. „Heterogeneous Distributed Big Data Clustering on Sparse Grids“. In: *Algorithms* 12.3 (2019). ISSN: 1999-4893. DOI: 10.3390/a12030060.

[120] David Pfander, Alexander Heinecke, and Dirk Pflüger. „A New Subspace-Based Algorithm for Efficient Spatially Adaptive Sparse Grid Regression, Classification and Multi-evaluation“. In: *Sparse Grids and Applications - Stuttgart 2014*. Springer International Publishing, 2016, pp. 221–246. ISBN: 978-3-319-28262-6.

[121] David Pfander et al. „Accelerating Octo-Tiger: Stellar Mergers on Intel Knights Landing with HPX“. In: *Proceedings of the International Workshop on OpenCL*. IWOCL'18. Oxford, United Kingdom: ACM, 2018, 19:1–19:8. ISBN: 978-1-4503-6439-3. DOI: 10.1145/3204919.3204938.

[122] Dirk Pflüger. „Spatially Adaptive Sparse Grids for High-Dimensional Problems“. PhD thesis. Technische Universität München, 2010.

[123] M. Pharr and W. R. Mark. „ispc: A SPMD compiler for high-performance CPU programming“. In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[124] Leszek Plaskota. „The exponent of discrepancy of sparse grids is at least 2.1933“. In: *Advances in Computational Mathematics* 12.1 (Jan. 2000), p. 3. ISSN: 1572-9044. DOI: 10.1023/A:1018900715321.

[125] James Price and Simon McIntosh-Smith. „Analyzing and Improving Performance Portability of OpenCL Applications via Auto-tuning“. In: *Proceedings of the 5th International Workshop on OpenCL*. IWOCL 2017. Toronto, Canada: ACM, 2017, 14:1–14:4. ISBN: 978-1-4503-5214-7. DOI: 10.1145/3078155.3078173.

[126] Markus Püschel et al. „Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms“. In: *The International Journal of High Performance Computing Applications* 18.1 (Feb. 2004), pp. 21–45. DOI: 10.1177/1094342004041291.

[127] Jonathan Ragan-Kelley et al. „Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines“. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI'13. Seattle, Washington, USA: ACM, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176.

[128] William M. Rand. „Objective Criteria for the Evaluation of Clustering Methods“. In: *Journal of the American Statistical Association* 66.336 (1971), pp. 846–850. DOI: 10.1080/01621459.1971.10482356.

[129] Florian Rathgeber et al. „PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes“. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. Nov. 2012, pp. 1116–1123. DOI: 10.1109/SC.Companion.2012.134.

[130] István. Z. Reguly et al. „The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations“. In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE, Nov. 2014, pp. 58–67. DOI: 10.1109/WOLFHPC.2014.7.

[131] Arch D. Robison. „Composable Parallel Patterns with Intel Cilk Plus“. In: *Computing in Science and Engg.* 15.2 (Mar. 2013), pp. 66–71. ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.21.

[132] Arch D. Robison. *Intel Threading Building Blocks*. Talk at HPCC'07. Houston, Texas, USA, Sept. 2007.

[133] Sean Rul et al. „An Experimental Study on Performance Portability of OpenCL kernels“. In: *2010 Symposium on Application Accelerators in High Performance Computing*. 2010, pp. 1–3.

[134] Amit Sabne et al. „Evaluating Performance Portability of OpenACC“. In: *Languages and Compilers for Parallel Computing*. Ed. by James Brodman and Peng Tu. Cham: Springer International Publishing, 2015, pp. 51–66. ISBN: 978-3-319-17473-0.

[135] Nadathur Satish et al. „Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?“ In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA'12. Portland, Oregon: IEEE Computer Society, 2012, pp. 440–451. ISBN: 978-1-4503-1642-2.

[136] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. School of Computer Science, Carnegie Mellon University, 1994.

[137] Hwanjun Song and Jae-Gil Lee. „RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning“. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD'18. Houston, TX, USA: ACM, 2018, pp. 1173–1187. ISBN: 978-1-4503-4703-7.

[138] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. „Matrix Multiplication Beyond Auto-Tuning: Rewrite-based GPU Code Generation“. In: *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. Oct. 2016, pp. 1–10.

[139] Arvind K. Sujeeth et al. „OptiML: An Implicitly Parallel Domain-specific Language for Machine Learning“. In: *Proceedings of the 28th International Conference on International Conference on Machine Learning*. ICML'11. Bellevue, Washington, USA: Omnipress, 2011, pp. 609–616. ISBN: 978-1-4503-0619-5.

[140] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. „Parallel Parameter Tuning for Applications with Performance Variability“. In: *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC'05. Seattle, WA, USA: IEEE, Nov. 2005. ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.52.

[141] Hiroyuki Takizawa and Hiroaki Kobayashi. „Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing“. In: *The Journal of Supercomputing* 36.3 (June 2006), pp. 219–234. ISSN: 1573-0484.

[142] Andrew S. Tanenbaum, Paul Klint, and Wim Bohm. „Guidelines for software portability“. In: *Software: Practice and Experience* 8.6 (1978), pp. 681–698. DOI: 10.1002/spe.4380080604.

[143] Yuan Tang et al. „The Pochoir Stencil Compiler“. In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA'11. San Jose, California, USA: ACM, 2011, pp. 117–128. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989508.

[144] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. „Active Harmony: Towards Automated Performance Tuning“. In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. SC'02. Baltimore, Maryland: IEEE Computer Society Press, 2002, pp. 1–11.

[145] *The Open Group Base Specifications Issue 7, 2018 edition*. Version 1003.1-2017. IEEE and The Open Group. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[146] *The OpenACC Application Programming Interface, Version 2.5*. Version 2.5. OpenACC-Standard.org. Oct. 2015. URL: http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf.

[147] *The OpenCL Standard 2.2*. Version 2.2. Khronos OpenCL Working Group. Nov. 2019. URL: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.

[148] A. Tiwari and J. K. Hollingsworth. „Online Adaptive Code Generation and Tuning“. In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 879–892. DOI: 10.1109/IPDPS.2011.86.

[149] Ananta Tiwari et al. „A Scalable Auto-tuning Framework for Compiler Optimization“. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. May 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161054.

[150] Ananta Tiwari et al. „Auto-tuning full applications: A case study“. In: *The International Journal of High Performance Computing Applications* 25.3 (2011), pp. 286–294. DOI: 10.1177/1094342011414744.

[151] *TOP500 List - November 2018*. Accessed: 2018-11-20. URL: https://www.top500.org/lists/2018/11/.

[152] R. Tylenda et al. „V1309 Scorpii: merger of a contact binary“. In: *Astronomy & Astrophysics* 528 (Apr. 2011). DOI: 10.1051/0004-6361/201016221.

[153] Brice Videau et al. „BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications“. In: *Int. J. High Perform. Comput. Appl.* 32.1 (Jan. 2018), pp. 28–44. ISSN: 1094-3420. DOI: 10.1177/1094342017718068.

[154] Vasiliy Volkov. *Better Performance at Lower Occupancy*. Sept. 2010. URL: http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf.

[155] Richard Vuduc, James W Demmel, and Katherine A Yelick. „OSKI: A library of automatically tuned sparse matrix kernels“. In: *Journal of Physics: Conference Series* 16.1 (2005), pp. 521–530.

[156] R. F. Webbink. „Double white dwarfs as progenitors of R Coronae Borealis stars and Type I supernovae“. In: *The Astrophysical Journal* 277 (Feb. 1984), pp. 355–360. DOI: 10.1086/161701.

[157] R. Clint Whaley and Jack J. Dongarra. „Automatically Tuned Linear Algebra Software“. In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC'98. San Jose, CA: IEEE Computer Society, 1998, pp. 1–27. ISBN: 0-89791-984-X.

[158] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. „Automated empirical optimizations of software and the ATLAS project“. In: *Parallel Computing* 27.1-2 (2001), pp. 3–35. ISSN: 0167-8191.

[159] Samuel Webb Williams. „Auto-tuning Performance on Multicore Computers“. PhD thesis. EECS Department, University of California, Berkeley, Dec. 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-164.html.

[160] Samuel Williams, Andrew Waterman, and David Patterson. „Roofline: An Insightful Visual Performance Model for Multicore Architectures“. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[161] *Working Draft, Standard for Programming Language C*. N1256. ISO/IEC JTC1/SC22/WG14. Sept. 2007.

[162] *Working Draft, Standard for Programming Language C++*. N4700. The C++ Standards CommitteeISO/IEC. Oct. 2017.

[163] Q. Yi et al. „POET: Parameterized Optimizations for Empirical Tuning“. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370637.

[164] Christoph Zenger. „Sparse Grids“. In: *Notes on Numerical Fluid Mechanics* 31 (1991), pp. 241–251.

[165] Yao Zhang, Mark Sinclair, and Andrew A. Chien. „Improving Performance Portability in OpenCL Programs“. In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 136–150. ISBN: 978-3-642-38750-0.

[166] Yongpeng Zhang and Frank Mueller. „Auto-generation and Auto-tuning of 3D Stencil Codes on GPU Clusters“. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. San Jose, California: ACM, 2012, pp. 155–164. ISBN: 978-1-4503-1206-6.

[167] Weirong Zhu, Yanwei Niu, and Guang R. Gao. „Performance portability on EARTH: a case study across several parallel architectures“. In: *Cluster Computing* 10.2 (June 2007), pp. 115–126. ISSN: 1573-7543. DOI: 10.1007/s10586-007-0011-1.

[168] Jure Zupan et al. „Classification of multicomponent analytical data of olive oils using different neural networks“. In: *Analytica Chimica Acta* 292.3 (1994), pp. 219–234. ISSN: 0003-2670.