Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Subspace-Optimal Data Mining on Spatially Adaptive Sparse Grids

Maximilian Luz

Course of Study:	Informatik
Examiner:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Supervisor:	Dipl.-Inf. David Pfander
Commenced:	May 15, 2017
Completed:	November 15, 2017
CR-Classification:	F2.1, G.1.0, G.1.2

Abstract

Continued improvements in technology lead to an ever-growing amount of data generated, for example, by scientific measurements and simulations. Data-mining is required to gain useful knowledge from this data, however, can be challenging especially due to the size and dimensionality of these problems. The use of regular grids for such applications is often limited by the curse of dimensionality, a phrase used to describe an exponential dependency of the computational complexity of a problem on the dimensionality of this problem. For many higher-dimensional problems, e.g. with 28 dimensions, regular grids cannot be used to compute results with the desired accuracy in a reasonable amount of time, even if the memory required to store and process them is available.

With spatially adaptive sparse grids, this problem can be overcome, as they lessen the influence of the dimensionality on the size of the grid, furthermore, they have been successfully applied for many tasks, including regression on large data sets. However, the currently preferred and in practice highly performant streaming-algorithm for regression on spatially adaptive sparse grids employs many unnecessary operations to effectively utilize modern parallel computer architectures, such as graphics processing units (GPUs).

In this thesis, we show that the implementation of a by computational complexity more promising subspace-linear algorithm on the GPU is able to out-perform the currently preferred streaming-algorithm on many scenarios, even though the this algorithm does not utilize modern architectures as well as the streaming-algorithm. Furthermore, we explore the construction of a new algorithm by combining both, streaming- and subspace-linear algorithm, which aims to process each subgrid of the grid with the algorithm deemed most efficient for its structure.

We evaluated both of our algorithms against the highly optimized implementation of the streaming-algorithm provided in the SG++ framework, and could indeed show speed-ups for both algorithms, depending on the experiments.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API application programming interface. 25

CAS compare-and-swap. 56

CG conjugate gradient. 36

CPU central processing unit. 23, 25, 26, 40

DR5 data release 5. 51

DRAM dynamic random-access memory. 28

FP32 32 bit floating point. 48

FP64 64 bit floating point. 48

GCN Graphics Core Next. 25

GPGPU general purpose graphics processing unit. 25

GPU graphics processing unit. 23

HBM2 2nd generation high bandwidth memory. 28, 65

HPC high performance computing. 28

LHC Large Hadron Collider. 53

MSE mean square error. 36

NaN Not-A-Number. 42

SDSS Sloan Digital Sky Survey. 51

SIMD single instruction multiple data. 26

SIMT single instruction multiple threads. 26

SLE system of linear equations. 36

SM Streaming Multiprocessor. 26

1 Preface

1.1 Introduction and Motivation

As it gets constantly easier to collect vast amounts of data, for example due to the increased precision and speed of measurement systems, the extraction of useful information from that data is critical. As many of these data sets are high-dimensional, a discretization of the feature-space by use of regular grids would introduce a, with the dimensinality of the problem, exponentially increasing number of data points whenever an increase in acuracy is desired. Instead of regular grids, so-called sparse grids can be used to circumvent this issue. They allow us to maintain accuracy while effectively reducing the required number of grid points, and thus find application in many high-dimensional data-driven workloads, such as encountered in finance, medicine, or even astrophysics [Pfl10].

Due to the large amount of data that needs to be processed, a parallelization of the workload is essential. For highly parallel workloads, Graphics processing units (GPUs) are a desired target architecture, as they provide a significantly larger throughput of arithmetic operations than CPUs, especially when comparing their cost. However, algorithms have to be adapted or developped specifically to fully exploit the benefits of this architecture.

For regression based data mining on sparse grids using the GPU, the currently preferred algorithm is the so-called streaming algorithm. The basic concept of this algorithm is fairly simple and is explicitly tailored towards this architecture, due to which even executes a considerable amount of unnecessary operations. In the past, however, it has been proven to out-perform, in terms of algorithmic run-time complexity, more desirable solutions [HP13]. To this end, Pfander, Heinecke, and Pflüger [PHP16] proposed a new algorithm that incorporates some of the desired algorithmical properties and also allows for an efficient implementation on contemporary parallel architectures. While they were able to beat the streaming-algorithm on CPUs, their algorithm has not been implemented and tested on GPUs. Furthermore, the amount of memory required by this algorithm for spatially adaptive sparse grids is dependent on the maximum depth of these grids, and not their actual size, which poses problems for highly refined grids.

In this thesis, we explore the implementation of the subspace-linear algorithm on GPUs and aim to find a solution to its problems. The rest of Chapter 1 presents the notation used throughout this document, as well as a general introduction to the GPU-architecture and some details of the NVIDIA Tesla P100 GPU we use for allmost all of our evaluation. An introduction to sparse grids, including spatially adaptive sparse grids, and sparse-grid-based regression is given in Chapter 2, which is followed by the discussion of existing sparse grid

regression algorithms in Chapter 3. The implementation of the subspace-linear algorithm on the GPU is given in Chapter 4, as well as the construction of a new algorithm combining both, streaming and subspace-linear approaches, to acheive better results. Their implementations are evaluated in Chapter 5, followed by a discussion of this thesis in Chapter 6.

1.2 Notation

Throughout this thesis, we follow the common mathematical notations and conventions as seen in similar literature, e.g. “Acta Numerica: Sparse grids” [BG04] and “Spatially Adaptive Sparse Grids for High-Dimensional Problems” [Pfl10]. For clarification, and to avoid misconceptions, this section will reiterate some of the lesser known notations and provide the additional syntax and semantics used in this document.

We define a multi-index $\boldsymbol{\alpha}$ as tuple

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{N}_0^d$$

For multi-indices and vector-valued properties we use the component-wise operations

$$\begin{aligned}
(\boldsymbol{\alpha} + \gamma)_i &\coloneqq \boldsymbol{\alpha}_i + \gamma \\
(\boldsymbol{\alpha} \cdot \gamma)_i &\coloneqq \boldsymbol{\alpha}_i \cdot \gamma \\
(\boldsymbol{\alpha}^\gamma)_i &\coloneqq (\boldsymbol{\alpha}_i)^\gamma \\
(\gamma^{\boldsymbol{\alpha}})_i &\coloneqq \gamma^{\boldsymbol{\alpha}_i} \\
(\boldsymbol{\beta} + \boldsymbol{\alpha})_i &\coloneqq \boldsymbol{\beta}_i + \boldsymbol{\alpha}_i \\
(\boldsymbol{\beta} \cdot \boldsymbol{\alpha})_i &\coloneqq \boldsymbol{\beta}_i \cdot \boldsymbol{\alpha}_i
\end{aligned}$$

with multi-indices (and/or vectors in $\mathbb{R}^d$) $\boldsymbol{\alpha}, \boldsymbol{\beta}$ and scalar γ, based on their scalar counterparts. Note that definitions for the operators “$-$” and “$\div$” follow by substitution.

We further define the scalar product as

$$\langle \boldsymbol{\alpha}, \boldsymbol{\beta} \rangle \coloneqq \sum_{i=1}^{d} \alpha_i \beta_i,$$

the relational operators

$$\begin{aligned}
\boldsymbol{\alpha} < \boldsymbol{\beta} \quad &\Leftrightarrow \quad \forall_{1 \le i \le d} : \alpha_i < \beta_i, \\
\boldsymbol{\alpha} \le \boldsymbol{\beta} \quad &\Leftrightarrow \quad \forall_{1 \le i \le d} : \alpha_i \le \beta_i \quad \text{and} \quad \boldsymbol{\alpha} \ne \boldsymbol{\beta}, \\
\boldsymbol{\alpha} \leqq \boldsymbol{\beta} \quad &\Leftrightarrow \quad \forall_{1 \le i \le d} : \alpha_i \le \beta_i,
\end{aligned}$$

and the norms

$$|\boldsymbol{\alpha}|_\infty \coloneqq \max_{1\leq i\leq d} |\alpha_i| \qquad (L_\infty\text{- or maximum norm}),$$

$$|\boldsymbol{\alpha}|_1 \coloneqq \sum_{i=1}^{d} |\alpha_i| \qquad (L_1\text{-norm}),$$

$$|\boldsymbol{\alpha}|_2 \coloneqq \sqrt{\sum_{i=1}^{d} \alpha_i^2} \qquad (L_2\text{-norm}).$$

Additionally we define

$$k = [x]_{\text{odd}}$$

as shorthand notation to round x to the nearest odd integer $k \in \mathbb{N}$.

1.3 The Graphics Processing Unit

As a large part of this thesis focuses on the implementation and optimization of regression algorithms for GPUs, an explanation of their underlying architecture, especially due to its significant differences compared to the typical CPU-architecture, is in order. This architecture has evolved over the years from a simple graphics-only co-processor to the highly parallel general purpose graphics processing units (GPGPUs), accelerating many of today's data-driven workloads such as deep learning, classification, and regression, capable of executing up to multiple thousands of threads in parallel and even more of them concurrently. While this high level of parallelism provides a serious performance benefit over CPUs, it can only be managed by architectural decisions inherently limiting the nature of tasks that can fully exploit these devices, which we will discuss later in this section.

Since we are using NVIDIA GPUs for all of our evaluations, we will focus on their architecture specifically and use the terms coined by them. Except for nomenclature, however, most of the details discussed in this section are similar between vendors and generations, such as NVIDIA's Kepler, Maxwell, and Pascal [NVI16], or AMD's current Graphics Core Next (GCN) [AMD17] devices.

Aside from integrated hardware (which we will deliberately ignore hereafter), GPUs are independent devices with their own memory, memory controller and instruction set. To differentiate between these entities we commonly refer to the system hosting the GPU as *host* and the GPU itself as *device*. To fully utilize the GPGPU capabilities, both, host and device, have to be programmed respectively. Frameworks, such as OpenCL [SGS10] or CUDA [NBGS08] (NVIDIA only) provide access from host-code to the GPU by giving programmers the means to write functions — so-called *kernels* — compile them to device-specific machine code, execute them on the GPU, synchronize them, and transfer the required memory back and forth. The last of which may already be a first bottleneck for some applications as this is limited by the bandwidth of the interface between CPU and GPU (at the time of

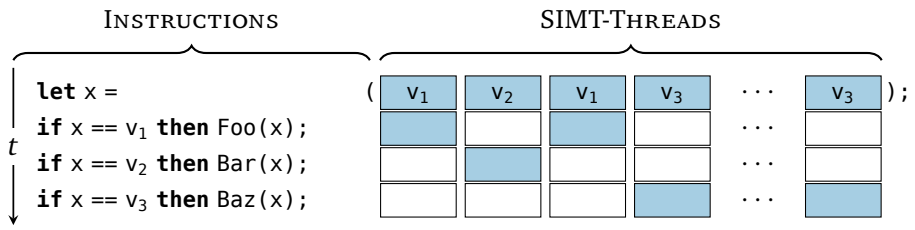

Figure 1.1: SIMT lockstep execution. Multiple hardware-threads execute the same vector instruction at cycle t. If branching occurs (last three instructions), the diverging paths are executed subsequently. Threads not on the current path are stalled (white) while only threads on the path are executing the operations (colored).

writing commonly PCIe Gen. 3 $\times$16 with a theoretical maximum of 15.75 GB/s or PCIe Gen. 3 $\times$8 with 7.88 GB/s, depending on the availability of PCIe lanes and devices connected). Both CUDA and OpenCL provide asynchronous application programming interfaces (APIs), allowing to mitigate some of the latency introduced by memory transfers through overlapping them with host- and even device-code execution.

Contemporary GPU-architecture is largely based on the single instruction multiple threads (SIMT) principle, similar to the single instruction multiple data (SIMD) principle found on CPUs. This means that batches, so-called *warps*, of multiple (for NVIDIA architectures 32) threads are executed in parallel on a single Streaming Multiprocessor (SM), all executing the same instruction. To allow for branching, instructions can be predicated on a thread-level basis, with the predicate deciding if the operation is going to be executed on the current thread. If the control flow diverges, i.e. some threads of a warp execute an operation based on a predicate and others do not, the threads of this warp not executing the operation are effectively stalled for that period of time (see Figure 1.1), yielding a scheme of execution commonly referred to as *lockstep execution*. Due to this, the cost of managing hundreds of threads is reduced to managing only one instruction path per SM at a time, however, this also means that if-else branches are serialized and thus take more time to execute on the GPU. Depending on the problem at hand, some of the control flow can be eliminated by clever use of arithmetic instructions, or even by grouping input data, such as data points, influencing the branching using, for example, space-filling curves.

In terms of memory, each SM has a large set of registers to store intermediate results, partitioned between all threads on the SM. Additionally, SMs have a limited amount of *shared memory* directly on the SM, a level one (L1) cache, and an instruction cache. On NVIDIA architectures, they are connected to multiple parallel level two (L2) caches. Each L2-cache is connected to a memory controller, and each memory controller to a part of the *global memory*. Loads and stores of multiple warps accessing global memory are typically coalesced into one or more memory transactions per warp whenever possible [NVI17]. Further performance problems can occur when single threads require too many registers. If too many registers are used, some of the registers are *spilled* to *local memory*, a per-multiprocessor reserved space in global memory often cached in the L1- or L2-cache, or alternatively the number of threads scheduled on a single SM may be reduced. Even

with caching, this can severely impact the performance as neither caches, local memory, or shared memory are as fast as registers. It may, however, be beneficial to pre-fetch global to shared memory, effectively reducing pressure on the L2-cache and slower global memory.

To mitigate some of the memory, and even instruction latencies, GPUs allow to schedule multiple warps on a single SM. With this, the SM can switch between warps to hide latencies. Following Little's Law, the number of instructions N required to hide a latency of L clock cycles for a throughput of T instructions per clock is $N = L \cdot T$. As contemporary NVIDIA Pascal devices are able to schedule two instructions per warp per clock cycle, this leads, assuming a latency L of 11 clock cycles and maximum throughput for all instructions, to 22 required warp instructions per warp, and thus 704 required independent operations per warp [NVI17]. These operations can be either provided by 704 other threads (22 other warps), or alternatively instruction level parallelism, i.e. subsequent independent instructions.

Another limitation of GPUs are function calls and recursion. Due to the huge amount of parallelism, maintaining an execution-stack on GPUs is difficult, often not viable, and, for NVIDIA devices has been first introduced with compute capability 2.x [NVI17]. If a stack is required, it cannot be stored in registers and thus is slow. Furthermore, large depths of recursion are impossible, as this would require a significant amount of memory per thread.

In summary, GPUs require a highly parallel, non-diverging, and preferably non-memory-driven workload. While some memory-related latencies can be eliminated by increasing the number of concurrent threads, transfer from host to device memory may also be an issue.

1.3.1 CUDA

CUDA [NBGS08; NVI17] is NVIDIA's proprietary GPGPU API. In contrast to OpenCL, it allows the integration of device code directly into C++ host code by employing a separate compiler. This provides several benefits, such as ease of use and an exhaustive language front end supporting most of the modern C++ features, however, also has some drawbacks. It is not (directly) possible to employ code-generation at run-time, which would allow certain optimizations such as parameter-specific loop unrolling. We chose CUDA over OpenCL, as it provides better access to hardware-specific instructions, such as true 64 bit atomic floating-point operations for devices with compute-capability 6.0 or higher, as well as superior support for performance profiling on NVIDIA devices.[1]

The actual programming model of CUDA is largely identical to the one of OpenCL. Programmers write, except for indexing of memory, seemingly single-threaded code, that is then executed in a parallel fashion on the GPU, with each scheduled thread receiving a unique combination of identifiers. This can be somewhat seen as programming a parallel *for* loop over a complete set of indices. Additionally, threads are grouped into so-called *thread-blocks*, with only the threads inside a thread block being capable of accessing the

[1] NVIDIA's development tools only support CUDA, which would leave us solely with the performance measurement functionalities provided by the OpenCL framework itself.

same shared memory, which can then be arranged in a grid-like fashion. Both thread-blocks and the full grid can have up to three dimensions, depending on the programmers needs, however, the only difference in dimensionality is the way single threads in the thread-block and single thread-blocks in the grid are indexed. The size of thread-blocks and grid can be specified at run-time by the programmer, but is limited by the executing hardware. For newer devices, this is a maximum grid size of $2^{32}-1$ in x-dimension and a maximum of 1024 threads per block.

Furthermore, CUDA provides a wide variety of device-code instructions for arithmetic operations, thread synchronization and intra-thread-block communication, as well as the standard memory-transfer and device-synchronization host-code procedures.

1.3.2 The NVIDIA Tesla P100

The Tesla P100 [NVI16] is, for non-selected entities, NVIDIA's latest available professional flagship GPGPU, designed for high performance computing (HPC) acceleration. It is based on the Pascal architecture and features 56 SM, each containing 32 64 bit and 64 32 bit CUDA cores, $2 \times 32\,768$ 32 bit registers, and two warp schedulers. Additionally, each SM contains 64 kB of shared memory and a separated L1 cache. One SM is capable of executing four warp-instructions per clock cycle, two for each scheduler, can have up to 64 resident warps, and execute up to two of them in parallel. The SMs share a 4096 kB L2 cache, connected, via eight memory controllers, to the 16 GiB of HBM2 DRAM main memory, divided into four stacks, leading to a theoretical peak bandwidth of 180 GB/s per stack.

2 Sparse Grids

As computers are, inherently and due to practical limitations, constrained to computations on a finite set of numbers, discretization of the problem space is a fundamental part of all algorithms and methods dealing with (our approximation of) physical reality or other continuous data. In computational science, the choice of the discretization scheme to be used for this task is generally restricted by three pragmatic reasons: We want the scheme, i.e. transformation of the problem into a discrete and thus approximated form, to be easily applicable, the error of this approximation to be small, and the transformed problem to be computable in an adequate amount of time. The straightforward choice is thus a grid-based approach. However, most of said approaches fall victim to the so-called *curse of dimensionality*, a phrase originating from Bellman [Bel61] nowadays often used to describe the exponential increase of grid points $n = \mathcal{O}(N^d)$ relative to the dimension d of the grid. This exponential relation renders them, due to memory and computational limitations, effectively unusable for moderate- and higher-dimensional problems, such as data mining, regression, and various other scenarios extensively discussed by Bungartz and Griebel [BG04], Buse [Bus15], Heinecke [Hei14], and Pflüger [Pfl10]. Spars grids are the result of a cost to benefit optimization designed to reduce the number of grid points while keeping the error introduced by these changes small [BG04], and a viable, well-established solution to mitigate this problem.

This chapter aims to introduce the general aspects of sparse grids used throughout this thesis. We will, in the given order, look at the definition of sparse grids and their composition from regular grids, the hierarchical hat basis functions, spatially adaptive sparse grids, and finally conclude with an overview of regression on sparse grids.

A more thorough and in-depth discussion of sparse grids, their origin, derivation, and use, can again be found in the literature by Bungartz and Griebel [BG04], Bungartz, Pflüger, and Zimmer [BPZ08], Buse [Bus15], Heinecke [Hei14], and Pflüger [Pfl10].

2.1 Sparse Grid Composition

Throughout this thesis, we follow the common restrictions to functions of type $f : \Omega \to \mathbb{R}$ with the d-dimensional hypercube $\Omega = [0, 1]^d$ as domain and boundary $f|_{\partial\Omega} = 0$. Note that the limitation of the domain to Ω is not a loss generality: Functions $f^* : \mathbb{R}^d \to \mathbb{R}$ can be normalized to fit, at least with the relevant part of their feature space, in Ω.

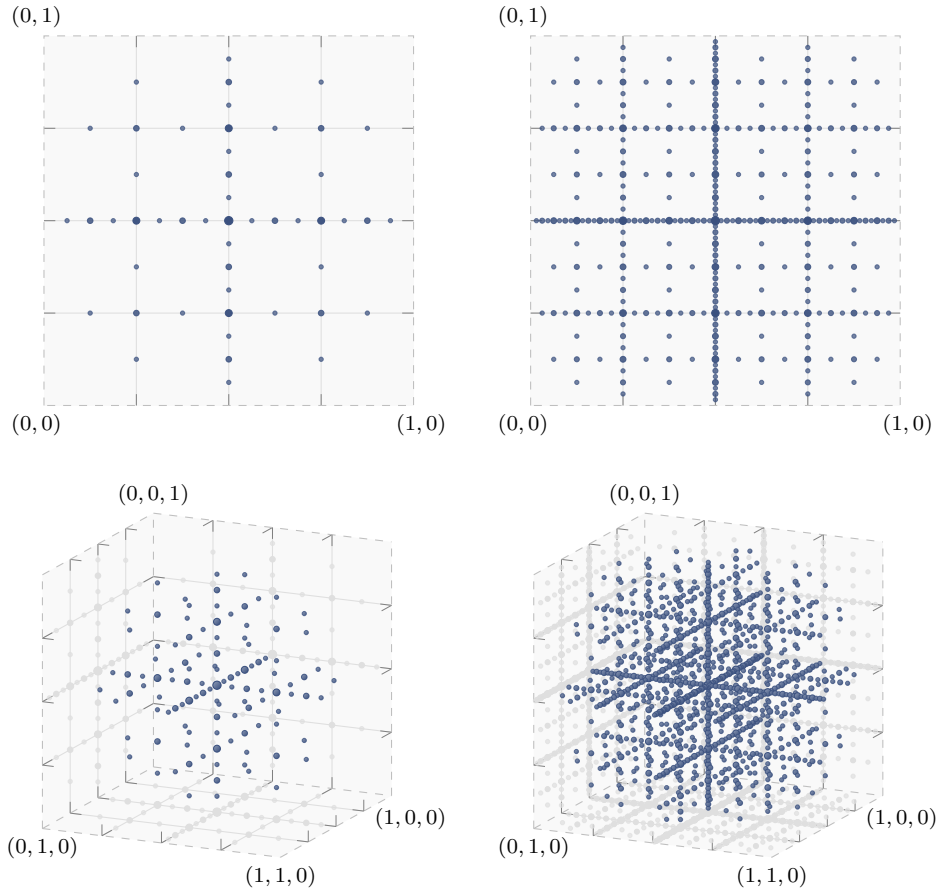

Figure 2.1: Regular sparse grids in two (top) and three dimensions (bottom), with a depth of $n = 4$ on the left and a depth of $n = 6$ on the right. Note the projections of the points (gray) to the xy-, xz-, and yz-planes in the three-dimensional figures forming the respective two-dimensional grid.

A sparse grid on the hypercube Ω can be reached via a special hierarchical decomposition of a regular grid in this domain. To this extent we define the index set

$$I_{\mathbf{l}} \coloneqq \left\{ \mathbf{i} = (i_1, i_2, \ldots, i_d) \;\middle|\; 0 < i_k < 2^{l_k},\ i_k \text{ odd} \right\} \tag{2.1}$$

with $\mathbf{l} \in \mathbb{N}^d$ as level-vector. Using this set we can define regular grids,

$$\Omega_{\mathbf{l}} \coloneqq \left\{ \mathbf{x}_{\mathbf{l},\mathbf{i}} = \mathbf{i} \cdot \mathbf{h}_{\mathbf{l}} = \mathbf{i} \cdot 2^{-\mathbf{l}} \;\middle|\; \mathbf{i} \in I_{\mathbf{l}} \right\}, \tag{2.2}$$

the so-called subgrids, and using these create a composition of the full regular grid

$$\Omega_n^{(\infty)} \coloneqq \bigcup_{|\mathbf{l}|_\infty \leq n} \Omega_{\mathbf{l}} = \{ \mathbf{x}_{\mathbf{l},\mathbf{i}} \in \Omega_{\mathbf{l}} \mid |\mathbf{l}|_\infty \leq n \} \tag{2.3}$$

with spacing $h = 2^{-n}$. Note that by using odd index values, there are no duplicate points on all combined subgrids. Similarly, we can now define sparse grids as

$$\Omega_n^{(1)} \coloneqq \bigcup_{|\mathbf{l}|_1 \leq d+n-1} \Omega_{\mathbf{l}} = \{ \mathbf{x}_{\mathbf{l},\mathbf{i}} \in \Omega_{\mathbf{l}} \mid |\mathbf{l}|_1 \leq d + n - 1 \}, \tag{2.4}$$

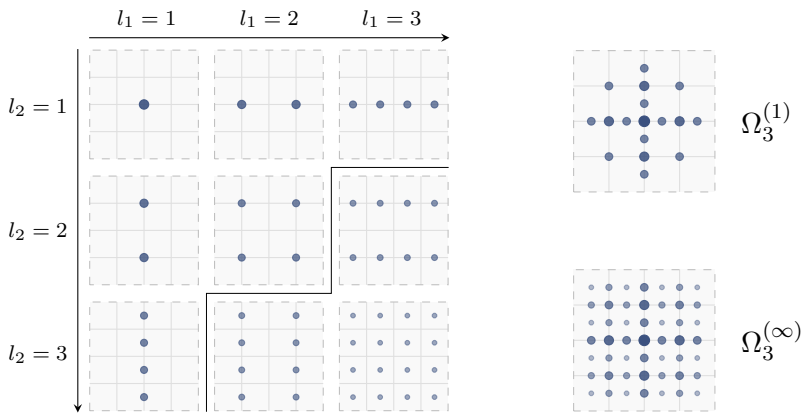

Figure 2.2: Tableau of two-dimensional subgrids for $\Omega_3^{(\infty)}$ and $\Omega_3^{(1)}$, the difference being indicated by the line separating them (left). The composition of the grids (right) is highlighted by opacity and size of the grid points.

see Figure 2.1 for an illustration. The choice of subgrids is based on the contribution of the function space spanned by the associated hierarchical basis functions, the so-called subspace, to the interpolant versus their cost, i.e. the number of points on the subgrid, and is elaborated in more detail by Bungartz and Griebel [BG04]. An example decomposition of sparse grids and regular grids into their subgrids can be seen in Figure 2.2.

2.2 Hierarchical Basis Functions

As elaborated previously, sparse grids are a hierarchical arrangement of isotropic and anisotropic regular subgrids. Therefore, they need to be combined with a set of hierarchical basis functions, such as the piecewise linear, piecewise polynomial, Mexican hat, or B-spline basis. In the context of this thesis, we limit ourselves to the use of the hierarchical piecewise multilinear basis functions, commonly referred to the *hierarchical hat basis functions*. For a detailed overview of these and various other functions see Pflüger [Pfl10].

The d-dimensional hierarchical hat basis is a composition of the standard one-dimensional hat function

$$\varphi(x) = \max(1 - |x|, 0). \tag{2.5}$$

To complement the previously defined hypercube $\Omega = [0,1]^d$, we expand this function, using dilation and translation, to a set of functions spanning the interval $[0,1]$ defined as

$$\varphi_{l,i}(x) \coloneqq \varphi\left(2^l x - i\right), \tag{2.6}$$

with level $l \in \mathbb{N}$ and index $i \in \mathbb{N}$. To extend the above to d dimensions, we use a tensor-product-based approach, resulting in

$$\varphi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) \coloneqq \prod_{k=1}^{d} \varphi_{l_k,i_k}(x_k). \tag{2.7}$$

We again limit the index vectors to $\mathbf{i} \in I_\mathbf{l}$ (see Equation (2.1)). This limitation creates, for a fixed level $\mathbf{l}$, a set of non-overlapping multi-dimensional basis functions, which is an important aspect referenced later on in this thesis, and the key difference to the similarly defined nodal basis. For a subgrid $\Omega_\mathbf{l}$ we can now obtain the corresponding *subspace* of basis functions, also called the *hierarchical increments*,

$$W_\mathbf{l} \coloneqq \operatorname{span}\left\{\varphi_{\mathbf{l},\mathbf{i}}\left(\mathbf{x}\right) \mid \mathbf{i} \in I_\mathbf{l}\right\} \tag{2.8}$$

$$= \operatorname{span} \mathcal{W}_\mathbf{l}, \tag{2.9}$$

illustrated in Figure 2.3. The discrete approximation space $V_n^{(\infty)}$ for the full grid and $V_n^{(1)}$ for the sparse grid are the direct sum of subspaces, again selected by the corresponding norms and limits:

$$V_n^{(\infty)} = \bigoplus_{|\mathbf{l}|_\infty \leq n} W_\mathbf{l}, \tag{2.10}$$

$$V_n^{(1)} = \bigoplus_{|\mathbf{l}|_1 \leq n+d-1} W_\mathbf{l} \quad . \tag{2.11}$$

With this, we can now write a sparse grid function $u \in V_n^{(1)}$ as

$$u\left(\mathbf{x}\right) = \sum_{|\mathbf{l}|_1 \leq n+d-1} \sum_{i \in I_\mathbf{l}} \alpha_{\mathbf{l},\mathbf{i}} \varphi_{\mathbf{l},\mathbf{i}}\left(\mathbf{x}\right) \tag{2.12}$$

$$= \sum_{i=0}^{N} \alpha_i \varphi_i\left(\mathbf{x}\right) \tag{2.13}$$

$$= \langle \mathbf{b}_\mathbf{x}, \boldsymbol{\alpha} \rangle, \tag{2.14}$$

where N denotes the number of grid points, $\boldsymbol{\alpha}$ the vector of sparse grid function coefficients, also referred to as *surpluses*, and $\mathbf{b}_\mathbf{x}$ the vector

$$\mathbf{b}_\mathbf{x} = \left(\varphi_0\left(\mathbf{x}\right), \varphi_1\left(\mathbf{x}\right), \ldots, \varphi_N\left(\mathbf{x}\right)\right)^\mathsf{T}. \tag{2.15}$$

An exemplary interpolation on sparse grids with the resulting sparse grid function and its surpluses can be found in Figure 2.4.

Having presented the basics of sparse grids we can now give a short overview of their benefit, which can be found in Table 2.1.

Grid Type	**Number of Points**	**Error of Interpolant**
Regular Grid	$\mathcal{O}\left(2^{d \cdot n}\right)$	$\mathcal{O}\left(h_n^2\right)$
Sparse Grid	$\mathcal{O}\left(2^n \cdot n^{d-1}\right)$	$\mathcal{O}\left(h_n^2 \cdot n^{d-1}\right)$

Table 2.1: Benefits of sparse grids: In relation to the corresponding regular grids the number of points is considerably lower, a factor improving relative to increasing dimension, while the error is only slightly worse. Note that $h_n = 2^{-n}$. The error terms are given with respect to both L^2- and maximum-norm [BG04].

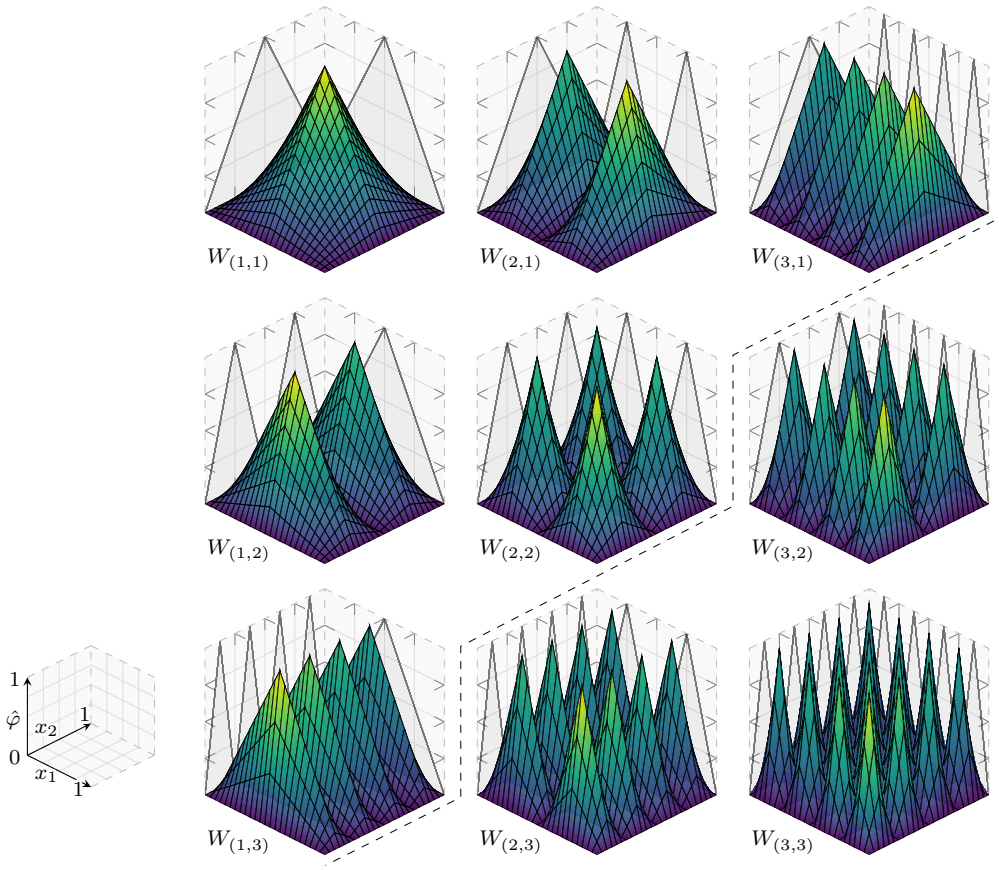

Figure 2.3: Two dimensional hierarchical hat basis functions, grouped by the corresponding subspaces, for $V_3^{(\infty)}$. The difference between $V_3^{(1)}$ and $V_3^{(\infty)}$ is indicated by the dashed line between subspaces. Note the corresponding one-dimensional basis functions plotted on their respective planes.

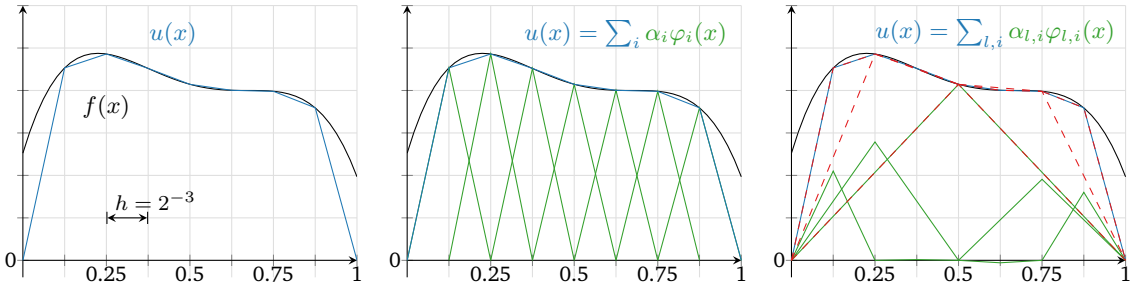

Figure 2.4: One-dimensional interpolation $u(x)$ (blue) of $f(x)$ (black) by use of the piecewise linear (nodal) hat basis functions (center, green) and the hierarchical hat basis functions (right, green). The partial sums of the hierarchical basis functions are indicated by the dashed lines.

2.3 Spatially Adaptive Sparse Grids

While regular sparse grids reduce memory and computational requirements over normal regular grids drastically, up to the point of being suitable for higher-dimensional problems, they too do not consider the structure of the underlying dataset. This leads to either a potential spending of many points in areas where only few are needed (to retain accuracy where a higher number of points is needed) or a loss of accuracy when not enough points are spent in a certain area (to reduce computational cost of the overall grid). Adaptive sparse grids allow us to selectively choose the areas, both spatial and dimensional, in which we want to use more and the areas in which we want to use less points. In the following chapters we will focus on spatially adaptive sparse grids exclusively, for dimensionally adaptive sparse grids see Buse [Bus15].

We define spatially adaptive sparse grids analogous to regular sparse grids (see Sections 2.1 and 2.2), with the exception that we base everything on the index sets

$$\tilde{I}_{\mathbf{l}} \coloneqq \{i \in I_{\mathbf{l}} \mid P(\mathbf{l}, \mathbf{i}) \text{ is true}\} \tag{2.16}$$

with predicate P, instead of the sets $I_{\mathbf{l}}$. In the following we will refer to the components of spatially adaptive sparse grids, e.g. their subspaces and subgrids, that have been redefined by use of $\tilde{I}_{\mathbf{l}}$ with a tilde. The definition of spatially adaptive sparse grids is not complete yet, as they have to obey certain restrictions, thus P may not be chosen arbitrarily. To express these restrictions, we begin by defining the relation $\mathbf{l}'$ *is parent of* $\mathbf{l}$ as $\mathbf{l}' \leq \mathbf{l}$ holds, expand this relation to subgrids and subspaces by comparing their respective level, and define the relation *is direct parent of* as a stricter version thereof, where additionally $|\mathbf{l}|_1 - |\mathbf{l}'|_1 = 1$. Furthermore, let

$$\hat{\mathcal{W}}_{\mathbf{l},\mathbf{i}} \coloneqq \left\{ \varphi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) \;\middle|\; \mathbf{l}' \leq \mathbf{l},\ \mathbf{i}' \in I_{\mathbf{l}'},\ \varphi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}_{\mathbf{l},\mathbf{i}}) \neq 0 \right\} \tag{2.17}$$

be the set of all non-zero basis functions of the full hierarchical basis for each parent level $\mathbf{l}'$ of $\mathbf{l}$. Analogous to their previous definitions we now expand the relation *is parent of* to points by support of the basis functions on parent subspaces, i.e. $\mathbf{x}_{\mathbf{i}',\mathbf{l}'}$ *is parent of* $\mathbf{x}_{\mathbf{i},\mathbf{l}}$ iff $\varphi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) \in \hat{\mathcal{W}}_{\mathbf{l},\mathbf{i}}$. For a spatially adaptive sparse grid $\tilde{\Omega}_n^{(1)}$ we then require

$$\mathbf{x}_{\mathbf{l},\mathbf{i}} \in \tilde{\Omega}_n^{(1)} \;\Rightarrow\; \hat{\mathcal{W}}_{\mathbf{l},\mathbf{i}} \subseteq \bigcup_{|\mathbf{l}|_1 \leq d+n-1} \tilde{\mathcal{W}}_{\mathbf{l}} \tag{2.18}$$

$$\Rightarrow\; \forall \varphi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) \in \hat{\mathcal{W}}_{\mathbf{l},\mathbf{i}} : \mathbf{i}' \in \tilde{I}_{\mathbf{l}'} \tag{2.19}$$

$$\Rightarrow\; \forall \varphi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) \in \hat{\mathcal{W}}_{\mathbf{l},\mathbf{i}} : \mathbf{x}_{\mathbf{l}',\mathbf{i}'} \in \tilde{\Omega}_n^{(1)} \tag{2.20}$$

i.e. we require that, if a point $x_{\mathbf{i},\mathbf{l}}$ is present in a (spatially adaptive) sparse grid, parent points have to be present in that grid as well (Equation (2.20)). Equivalently all basis functions, that can potentially contribute to this point, of all parent subspaces have to be present in the grid (Equations (2.18) and (2.19)) Remember that the sets $\tilde{\mathcal{W}}_{\mathbf{l}}$ and $\tilde{\Omega}_n^{(1)}$ are based on the filtered index set $\tilde{I}_{\mathbf{l}}$ instead of $I_{\mathbf{l}}$, and thus modified for adaptive grids, but otherwise identical to the sets $\mathcal{W}_{\mathbf{l}}$ or $\Omega_n^{(1)}$ of non-adaptive grids respectively. Note that for

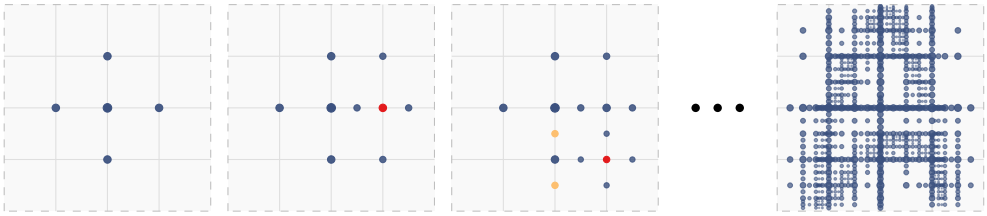

Figure 2.5: Two-dimensional spatially adaptive sparse grid (right) created from the regular grid $\Omega_2^{(\infty)}$ (left). The illustrated refinement process (left to right) expands a single point (red) and adds all its children, as well as its parents if they have not been part of the grid before (yellow).

the hierarchical hat basis functions there can at most be one direct parent for a point per dimension, as there is only one supporting basis function for a point per subspace.

The predicate P is usually constructed by refinement in consideration of the underlying dataset, meaning one or more promising grid points are chosen and some (or all) children of this point are added to the grid, as well as, to uphold the characteristics noted above, all parents of the added points if they are not already present. An example of a two-dimensional spatially adaptive sparse grid and the construction thereof can be found in Figure 2.5. For further details, such as the choice of grid points to refine, we refer to Pflüger [Pfl10].

2.4 Regression on Sparse Grids

The general idea of regression is to create a reconstruction u of an unknown multivariate function f, based on a set of M potentially noisy samples thereof, the so-called training data set

$$T \coloneqq \{(\mathbf{x}_m, y_m) \mid \Omega \times \mathbb{R}\}_{m=1}^{M}, \tag{2.21}$$

again with hypercube $\Omega = [0,1]^d$ and restriction to $f : \Omega \to \mathbb{R}$ [1]. Note that we can thus also use regression as a form of binary classification: We can interpret regression as learning the threshold $u(\mathbf{x}) = 0$ between two classes, denoted by the values -1 and 1, where $|u(\mathbf{x}_m)|$ indicates the certainty with which $\mathbf{x}_m$ belongs to class $\operatorname{sign}(u(\mathbf{x}_m))$. This approach can even be adapted for non-binary classification problems [Pfl10].

[1] For the definition of regression on sparse grids, we actually do not have to impose this limitation and could choose $f : \mathbb{R}^d \to \mathbb{R}$. However, as we have restricted ourselves to sparse grids with $|_{\partial f} = 0$, samples outside of the defined hypercube would not make much sense. Furthermore, we hereafter assume that all sample points $\mathbf{x} \in [0,1]^d$.

To reconstruct the function u from the samples of f, so that

$$u : \Omega_n^{(1)} \to \mathbb{R}, \quad f : \Omega \to \mathbb{R}, \quad \text{and} \quad u \approx f, \tag{2.22}$$

we employ the commonly used least squares approach in combination with a Tikhonov regularization

$$u = \underset{u \in V_n^{(1)}}{\arg\min} \left(\frac{1}{M} \sum_{m=1}^{M} \left(y_m - u\left(\mathbf{x}_m\right) \right)^2 + \lambda \mathcal{C}(u) \right), \tag{2.23}$$

with λ controlling the trade-off between the mean square error (MSE) and the smoothness ensured by the weight-decay regularization functional $\mathcal{C}(u)$ [Pfl10]. For said functional we use the easy to calculate

$$\mathcal{C}(u) = |\boldsymbol{\alpha}|_2^2, \tag{2.24}$$

which exploits the intrinsic smoothness of the hierarchical basis functions and thus usually leads to similar results compared to the norm of the gradient of f, as mentioned and discussed by Pflüger [Pfl10].

The solution of Equation (2.23) in combination with (2.24) can be obtained by setting the gradient of the inner term to zero, and given by the linear system of equations

$$\left(\frac{1}{M} B B^\intercal + \lambda \mathbb{1} \right) \boldsymbol{\alpha} \;=\; \frac{1}{M} B \mathbf{y}, \tag{2.25}$$

where $\mathbf{y}$ is a vector of size M containing the potentially noisy evaluations of f, $\mathbb{1}$ the identity matrix, and B the matrix

$$B = \left(b_{\mathbf{x}_1}, b_{\mathbf{x}_2}, \ldots, b_{\mathbf{x}_M} \right) \tag{2.26}$$

$$= \begin{pmatrix} \varphi_1\left(\mathbf{x}_1\right) & \varphi_1\left(\mathbf{x}_2\right) & \cdots & \varphi_1\left(\mathbf{x}_M\right) \\ \varphi_2\left(\mathbf{x}_1\right) & \varphi_2\left(\mathbf{x}_2\right) & \cdots & \varphi_2\left(\mathbf{x}_M\right) \\ \vdots & & \ddots & \\ \varphi_N\left(\mathbf{x}_1\right) & \varphi_N\left(\mathbf{x}_2\right) & \cdots & \varphi_N\left(\mathbf{x}_M\right) \end{pmatrix} \tag{2.27}$$

containing the evaluations of the basis functions φ_i at the sample points $\mathbf{x}_m$ [Pfl10]. The solution of this system of linear equations (SLE), in turn, can be computed by applying a solver, such as the conjugate gradient (CG) algorithm [HS52; She94], which we will use throughout this thesis.

Looking at the computational cost of the process described above, we can see that the solver itself is quite cheap in comparison to the two matrix multiplications

$$\mathbf{v} \coloneqq B^\intercal \boldsymbol{\alpha} \tag{2.28}$$

$$\mathbf{v}' \coloneqq B \mathbf{v} \tag{2.29}$$

it has to calculate each iteration [PHP16]. As Equation (2.28) is nothing more than an evaluation of $u\left(\mathbf{x}_m\right)$ for the points $\{\mathbf{x}_m \mid 1 \leq m \leq M\}$ in a single operation (see Equations (2.14)

and (2.26)), we also refer to it as *multi-evaluation* and to Equation (2.29) as *transposed multi-evaluation*. The multiplication with matrix B and contents thereof imply an evaluation of all basis functions φ_i at all points of evaluation $\mathbf{x}_m$, leading to an approach with cost $\mathcal{O}(N \cdot M)$. However, with regards to the hierarchical hat basis functions introduced in Section 2.2, this approach seems, at least in theory, to be naïve, as there can only be one non-zero evaluation $\varphi_{\mathbf{l},\mathbf{i}}(\mathbf{x}_m)$ for all indices $\mathbf{i}$ for a given point $\mathbf{x}_m$ and subspace $W_\mathbf{l}$. The objective of this thesis is to develop algorithms for multi-evaluation and transposed multi-evaluation that perform not only in theory, but also in practice on the GPU, better (on regular and adaptive sparse grids) than the, at the time of writing, primarily used $\mathcal{O}(N \cdot M)$ solution.

3 Fast Regression Algorithms for Sparse Grids

With the mathematical foundation of regression on sparse grids introduced in the last section of the previous chapter, we can now look at some existing algorithms thereof. We will begin this chapter by outlining a few common properties and restrictions that influence their composition. The main sections then discusses a simple recursive strategy, followed by the commonly used streaming algorithm, expressing a straightforward approach to compute the operators $B^\intercal\boldsymbol{\alpha}$ and $B\mathbf{v}$, and finally the subspace-linear algorithm presented by Pfander, Heinecke, and Pflüger [PHP16], employing an, at least in theory, more favorable strategy.

Based on Equations (2.23) and (2.24) and the resulting system of linear equations (SLE) (2.25), we exclusively focus on the multi-evaluation $B^\intercal\boldsymbol{\alpha}$ and the transposed multi-evaluation $B\mathbf{v}$, as they pose the biggest potential for problem-specific optimization. Note that the choice of $\mathcal{C}(u) = |\boldsymbol{\alpha}|_2^2$ simplifies a potential third non-trivial operator to the identity matrix. A property exploitable by all algorithms below is the fact that the individual sparse grid function evaluations performed by $B^\intercal\boldsymbol{\alpha}$ are completely independent from each other and thus simple to parallelize. For $B\mathbf{v}$, the results are accumulated per basis function which would suggest a parallelization over the grid points instead. Due to the size of the matrix B, it is not feasible to explicitly assemble it. While we could reduce this size from $N \times M$, N being the number of grid points, M being the number of data points to evaluate, to $N \times N$ by combining B and $B^\intercal$ to $BB^\intercal$, as we generally consider N to be significantly smaller than M, a storage requirement of $\mathcal{O}\left(N^2\right)$ would still be impractical for at least larger grids. Furthermore, as B consists of basis function evaluations of all basis functions on all data points and the hat basis function has only one non-zero evaluation per point per subspace, there would be a considerable amount of unnecessary function evaluations by its explicit creation. Note that B, nevertheless, is not a sparse matrix, especially when we employ adaptive sparse grids, which aim to reduce the amount of unimportant, i.e. zero or close-to-zero, evaluations in comparison to a full sparse grid. Fortunately, these evaluations are comparatively cheap, with a cost of $\mathcal{O}(d)$, and thus can be computed on the fly, which also gives us an opportunity to select which basis functions to evaluate for which data points.

3.1 A Recursive Approach

A simple algorithm for the operator $B^\intercal\boldsymbol{\alpha}$ would be a recursive descent through the tree of subspaces and dimensions for each data point x_m. By evaluating the dimensions subsequently, we can exploit the tensor product composition of our basis functions, as we only

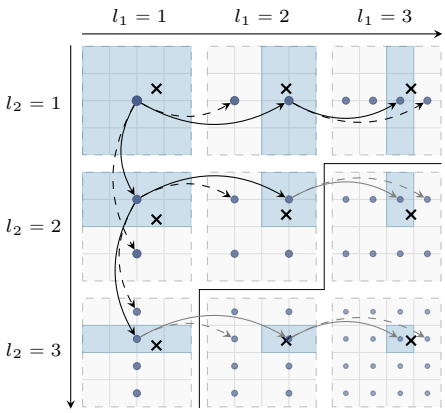

Figure 3.1: The recursive evaluation algorithm for a single data point (cross). Solid arrows indicate the paths taken, leading to the grid point and basis function which supports the data point, indicated by their highlighted domain. We begin by the recursive descent at the root ($\mathbf{l} = (1, 1)$) in the last dimension ($d = 2$). Note that, in this example, we can cache the one-dimensional results for $d = 2$. Compare Figure 2.3 for the respective basis functions. [HP13]

have to calculate the one-dimensional evaluation $\alpha_{l_d,i_d}\varphi_{l_d,i_d}(x_{m_d})$ once for dimension d and can then multiply this result with the result of the corresponding subtree, i.e. the sum of the components in the remaining $d-1$ dimensions (see Figure 3.1) [HP13]. Note that for a descent in a fixed dimension, the resulting tree of basis functions is binary and thus the direction of traversal is easy to calculate.

To compute the operator $B\mathbf{v}$, we can adapt the algorithm described above. We still employ the same traversal scheme, however, we now sum up the partial results $v_m\varphi_k(\mathbf{x}_m)$ for each basis function individually instead of calculating a weighted sum per data point [HP13].

While both algorithms are optimal in terms of multi-dimensional basis function evaluations, we only evaluate one multi-dimensional basis function per subspace per point and even cache some of the one-dimensional evaluations for $B^\intercal\boldsymbol{\alpha}$, they pose implementational challenges. Parallelization can be done straightforward over the data points, however the operator $B\mathbf{v}$ would then require synchronized access to sum up the partial results. Additionally, the recursive structure of these algorithms render them difficult to be competitively implement on the GPU[1], and their need for a multi-dimensional access scheme of the grid points further complicates matters.

[1] While CUDA supports recursive function calls, the size of the call stacks is considerably more limited than on central processing units (CPUs). Furthermore, the call stacks are, due to their size, most likely allocated in local (or possibly shared) memory, which would lead to a significant impact on performance over a primarily register using non-recursive implementation.

Algorithm 3.1 Streaming algorithm for the multi-evaluation $B^\intercal\boldsymbol{\alpha}$.

Function BASISEVAL **is**
  Input : Level vector $\mathbf{l}$, index vector $\mathbf{i}$, d-dimensional data point $\mathbf{x}_m$
  Output : $s = \varphi_{\mathbf{l},\mathbf{i}}(\mathbf{x}_m)$
  $s \leftarrow 1$
  for $k \leftarrow 1$ **to** d **do**
    $s \leftarrow s \cdot \varphi_{l_k,i_k}(x_{m_k}) = s \cdot \text{MAX}(1 - |2^{l_k} \cdot x_{m_k} - i_k|, 0)$
  return s

Procedure MULTIEVALSTREAMING **is**
  Input : Sparse grid Ω, data set T, surpluses $\boldsymbol{\alpha}$
  Output : $\mathbf{v} = B^\intercal\boldsymbol{\alpha}$
  parallel forall $\mathbf{x}_m \in T$ **do**
    $v_m \leftarrow 0$
    forall $(\mathbf{l},\mathbf{i}) \in \Omega$ **do**
      $v_m \leftarrow v_m + \alpha_{\mathbf{l},\mathbf{i}} \cdot \text{BASISEVAL}(\mathbf{l},\mathbf{i},\mathbf{x}_m)$

Algorithm 3.2 Streaming algorithm for the transposed multi-evaluation $B\mathbf{v}$.

Procedure MULTIEVALSTREAMINGTRANSPOSED **is**
  Input : Sparse grid Ω, data set T, vector $\mathbf{v}$
  Output : $\mathbf{v}' = B\mathbf{v}$
  parallel forall $(\mathbf{l},\mathbf{i},k) \in \Omega$ **do**
    $v'_k \leftarrow 0$
    forall $\mathbf{x}_m \in T$ **do**
      $v'_k \leftarrow v'_k + v_m \cdot \text{BASISEVAL}(\mathbf{l},\mathbf{i},\mathbf{x}_m)$

3.2 The Streaming Algorithm

In contrast to the recursive algorithm described in the previous section, the so-called streaming algorithm is an iterative algorithm and tries to exploit the properties of modern hardware. Due to this, it ignores most aspects of the basis functions. The unoptimized algorithm for the operator $B^\intercal\boldsymbol{\alpha}$ can be expressed as a loop over all grid points, evaluating the corresponding basis function at each point individually. Again, the data points $\mathbf{x}_m$ are evaluated in parallel, see Algorithm 3.1. $B\mathbf{v}$ can be evaluated largely identical to $B^\intercal\boldsymbol{\alpha}$, however for it we parallelize over the grid points to avoid the need for potentially costly synchronization, as the results are accumulated per grid point (see Algorithm 3.2).

The obvious drawback of the streaming algorithm is, that it essentially computes the full matrix multiplication, meaning that it evaluates every basis function at every data point, inevitably leading to a lot of zero-evaluations. Furthermore, the algorithm for $B\mathbf{v}$ parallelizes over the grid points. Generally, we consider the number of data points to be large enough to

saturate the need for parallelism on concurrent GPUs, however, we also expect the number of grid points to be noticeably smaller. This may lead to problems exploiting the full potential provided by the hardware, see Heinecke and Pflüger [HP13] for details.

Regardless of its problems, this algorithm still performs notably well in practice. An important aspect for this is the fact that all parallel threads process the same grid points in sequence (thus the name streaming), which leads to a very cache-friendly behavior. As a result, this algorithm does not depend as much on memory latency and bandwidth as the recursive algorithm described in the previous section, effectively accessing the memory in a random fashion. Additionally, a wide variety of standard hardware-based optimizations can be easily implemented, such as for example, pre-fetching of data into caches or registers and loop unrolling. The combination of structure and optimizations possible has, in the past, been proven to make up for the many unnecessary evaluations it computes in terms of performance and has led to it being the commonly used algorithm for regression on sparse grids [Hei14; HP13]. A key aspect in these ascertainments is the fact that the overhead created by those unnecessary evaluations is expectably lower on spatially adaptive sparse grids, as these grids, in general, only contain comparatively few points per subgrid.

3.3 The Subspace-Linear Algorithm

Following the idea of the recursive approach, the so-called subspace-linear algorithm, introduced by Pfander, Heinecke, and Pflüger [PHP16], aims to avoid unnecessary zero-evaluations, while in contrast to it being iterative. The general idea of the subspace-linear algorithm is to linearize the tree of subspaces and iterate through the resulting list, evaluating one multi-dimensional basis function per subspace per data point. We again parallelize over the data points, this time however for both $B^\intercal\boldsymbol{\alpha}$ and $B\mathbf{v}$.

For each subspace, we store its level-vector $\mathbf{l}$ and the corresponding set of surpluses $\boldsymbol{\alpha}_\mathbf{l}$. When we process a subspace for a given data point $\mathbf{x}_m$, we can calculate the index of the closest grid point, and thus the supporting basis function, by using the relation $\mathbf{x}_{\mathbf{l},\mathbf{i}} = \mathbf{i} \cdot 2^{-\mathbf{l}}$ (see Equation (2.2)) in combination with this data point and rounding the result to the nearest odd:

$$i_k = \left[x_{m_k} \cdot 2^{l_k}\right]_{\text{odd}}. \tag{3.1}$$

Having calculated the index vector we can now use it to evaluate the corresponding basis functions and retrieve the respective surplus from the set of surpluses associated with this subspace. Let us, for a moment, assume that we are dealing with full (potentially anisotropic) regular subgrids, and thus regular sparse grids.. The maximum number of index-values for a subgrid with level $\mathbf{l}$ in dimension k can then be given by 2^{l_k-1}, which is directly related to this subgrid. Thus we can effectively linearize the index vector $\mathbf{i}$, allowing us to store the surpluses of a subspace in a single one-dimensional array (see Algorithm 3.3) [PHP16]. For spatially adaptive sparse grids, however, the subgrids do not have to be regular. To overcome this obstacle, we could, for the vector of surpluses, treat the grid as if it were regular and insert a value of zero as replacement for each missing surplus. A better choice

Algorithm 3.3 Common functions for subspace-linear multi-evaluation.

Function IndexVector **is**
  Input : Level vector $\mathbf{l}$, d-dimensional data point $\mathbf{x}_m$
  Output : Index vector $\mathbf{i}$
  for $k \leftarrow 1$ **to** d **do**
    $i_k \leftarrow \text{RoundToNextOdd}(x_{m_k} \cdot 2^{l_k})$
  return $\mathbf{i}$

Function LinearIndex **is**
  Input : Level vector $\mathbf{l}$ and index vector $\mathbf{i}$, both d-dimensional
  Output : Linearized index $\hat{\imath}$
  $\hat{\imath} \leftarrow 0$
  for $k \leftarrow 1$ **to** d **do**
    $\hat{\imath} \leftarrow \hat{\imath} \cdot 2^{l_k - 1} + \lfloor i_k \cdot 2^{-1} \rfloor$
  return $\hat{\imath}$

for further optimization is the replacement of the missing surpluses with the IEEE 754 special Not-A-Number (NaN) value, due to which we can then differentiate between existing and non-existing grid points. This, of course, requires a check for NaN valued surpluses.

For both operators, $B^\intercal \boldsymbol{\alpha}$ and $B\mathbf{v}$, adaption of the process described above is straightforward. For $B^\intercal \boldsymbol{\alpha}$ we again accumulate the result per data point, for $B\mathbf{v}$ we use the (zero-initialized) surplus arrays to accumulate the partial results of the corresponding basis function, which we afterwards need to gather to create the resulting vector $\mathbf{v}'$.

A, for adaptive sparse grids, important algorithmic improvement to this algorithm is subspace skipping. From our sparse grid consistency definitions in Section 2.3 (Equations (2.18) to (2.20)) follows that, if a point has support on a grid point $\mathbf{x}_{\mathbf{l},\mathbf{i}}$, it has to have support on all parent subspaces with level $\mathbf{l}' \leq \mathbf{l}$. In theory, we can now use this property to skip all subspaces with level $\mathbf{l}' \geq \mathbf{l}$ once we encounter the first surplus value of NaN on level $\mathbf{l}$ for the data point we are currently processing. Keeping track of all subspaces to skip is in practice however not easily possible. To this end, Pfander, Heinecke, and Pflüger [PHP16] propose to sort the list of subspaces we employ lexicographically by their respective level vector, and then only skip the subsequent child-subspaces in this list by use of a pre-calculated index (per subspace), leading to Algorithms 3.4 and 3.5. Compare Figure 3.1 for the similarity between the list of sorted subspaces and the recursive approach.

Pfander, Heinecke, and Pflüger [PHP16] also describe another improvement to this algorithm. Similar to the recursive algorithm described in Section 3.1, we can cache intermediate results, this time not only the one-dimensional basis function evaluations, but also the index components. As our subspaces are sorted lexicographically according to their level vector, we expect that most of the time only few components of the level vector change from one subspace to the next in the list. Due to the index-component and basis function evaluation for a fixed dimension d on a given subspace together only depending on the level component of that dimension, we can re-use all one-dimensional intermediate results,

Algorithm 3.4 Subspace-Linear algorithm for the multi-evaluation $B^\intercal \boldsymbol{\alpha}$.

Procedure MULTIEVALSUBSPACELINEAR **is**
  Input : Sparse grid Ω with list of subspaces S, data set T, surpluses $\boldsymbol{\alpha}$
  Output : $\mathbf{v} = B^\intercal \boldsymbol{\alpha}$

  SCATTER$(\boldsymbol{\alpha}) \to S$
  parallel forall $\mathbf{x}_m \in T$ **do**
    $v_m \leftarrow 0$
    $k \leftarrow 1$
    while $k \leq |S|$ **do**
      $\left(\mathbf{l}, \boldsymbol{\alpha}_\mathbf{l}, \hat{k}\right) =$ LOAD(S, k)
      $\mathbf{i} \leftarrow$ INDEXVECTOR$(\mathbf{l}, \mathbf{x}_m)$
      $s \leftarrow$ LOAD$(\boldsymbol{\alpha}_\mathbf{l},$ LINEARINDEX$(\mathbf{l}, \mathbf{i}))$
      if $\neg$ISNAN(s) **then**
        $v_m \leftarrow v_m + s \cdot$ BASISEVAL$(\mathbf{l}, \mathbf{i}, \mathbf{x}_m)$
        $k \leftarrow k + 1$
      else
        $k \leftarrow \hat{k}$

Algorithm 3.5 Subspace-Linear algorithm for the transposed multi-evaluation $B\mathbf{v}$.

Procedure MULTIEVALSUBSPACELINEARTRANSPOSED **is**
  Input : Sparse grid Ω with list of subspaces S, data set T, vector $\mathbf{v}$
  Output : $\mathbf{v}' = B\mathbf{v}$

  SCATTER$((0, 0, \ldots, 0)^\intercal) \to S$ `// Initialize existing surpluses to zero`
  parallel forall $\mathbf{x}_m \in T$ **do**
    $k \leftarrow 1$
    while $k \leq |S|$ **do**
      $\left(\mathbf{l}, \boldsymbol{\alpha}_\mathbf{l}, \hat{k}\right) =$ LOAD(S, k)
      $\mathbf{i} \leftarrow$ INDEXVECTOR$(\mathbf{l}, \mathbf{x}_m)$
      $s \leftarrow$ REFERENCE$(\boldsymbol{\alpha}_\mathbf{l},$ LINEARINDEX$(\mathbf{l}, \mathbf{i}))$
      if $\neg$ISNAN(LOAD(s)) **then**
        ATOMICADD$(s, v_m \cdot$ BASISEVAL$(\mathbf{l}, \mathbf{i}, \mathbf{x}_m))$
        $k \leftarrow k + 1$
      else
        $k \leftarrow \hat{k}$
  $\mathbf{v}' \leftarrow$ GATHER(S) `// Collect result from surpluses`

i.e. the index-components and basis function evaluation, for dimensions in which the level-vector components kept unchanged. Thus we can effectively apply caching for those.

While this algorithm, in comparison to the streaming algorithm, again uses a theoretically more favorable approach by not brute-force style evaluating all basis functions, it also introduces some overhead. We need to create the arrays for the surpluses, sort the subspaces, and for each data point and subspace calculate the index of the surplus. Note that all of these additional costs, however, do not impact the asymptotic run time class of the algorithm in a negative way. Both, the multi-dimensional basis function evaluations and the index-vectors can be calculated in $\mathcal{O}(d)$ each, creating the surplus arrays can be done in $\mathcal{O}(N)$, and the subspaces can be sorted in $\mathcal{O}(S \log S)$, with N as the number of grid points, S as the number of surpluses and d as the number of dimensions. We generally expect the number of subspaces S to be notably smaller than the number of grid points N. A more serious problem for concurrent architectures is the inevitable largely randomized memory access and the synchronized access to the surpluses required for the operator $B\mathbf{v}$. Furthermore, the algorithm, as described here, potentially requires a lot of memory as it stores every subgrid as a full grid. For highly adaptive scenarios, this can lead to a significant increase of memory required. To resolve this problem, Pfander, Heinecke, and Pflüger [PHP16] propose to compress largely unused subspaces. We aim to address this and other problems later on in this thesis.

Nevertheless Pfander, Heinecke, and Pflüger [PHP16] have shown that, at least on CPUs and with hardware-based optimizations, this algorithm is able to out-perform the streaming algorithm on all scenarios tested.

4 Fast Regression Algorithms on the GPU

In this chapter, we discuss our implementation of the subspace-linear algorithm on the GPU, and explore the combination of subspace-linear and streaming algorithms to reduce the overhead introduced per data point evaluation for the subspace-linear algorithm, as well as lessen its, on the maximum level of the grid depending, memory requirement.

4.1 Subspace-Linear Regression on GPUs

To our knowledge, the subspace-linear algorithm presented in Section 3.3 has not yet been implemented and tested on the GPU. We have already seen that the algorithm is highly parallel, as we parallelize over all data points, and thus acceleration using GPUs is the next step in achieving better performance. While the implementation of an unoptimized version is pretty straight-forward, there are a few choices to consider regarding optimization for these specific devices.

Starting with the algorithm employing subspace-skipping, as presented in Algorithms 3.4 and 3.5, we can see that index $\mathbf{i}$ of a grid point $\mathbf{x}_{\mathbf{l},\mathbf{i}}$ is only required to compute the index of the corresponding surplus and the basis function evaluation. Both of these computations can be implemented by a step-wise iteration over all dimensions, and for a single dimension require only the index component of that dimension. By moving the computation of the basis function evaluation out of the if-clause and interleaving it with the surplus-index calculation, we can effectively save registers. To explicitly store the index, we would have required one register per dimension, which can be costly considering higher-dimensional problems. The if-clause now only contains the multiplication with the respective coefficient and the summation of the result. Furthermore, this introduces more instruction-level parallelism, as the basis function evaluation is independent from the surplus-index calculation. A downside of this step is, that we now compute the basis function regardless of the existence of a grid point. Note, however, that we still only evaluate a maximum of one basis function per subspace and that, due to the lockstep execution of warps, we would experience a similar run time of the original algorithm even if only one thread of the warp would have to evaluate a basis function.

To further increase instruction-level parallelism, we do not only process one data point per thread, but multiple, i.e. a *block* of data points. This also allows us to reduce the memory-pressure if we process the subspaces coherently with respect to the block, meaning that the complete block executes computations for the same subspace and thus we only have to load the level-vector $\mathbf{l}$ of that subspace once per block instead of once per data point. A subspace

is now skipped only when all data points of the block have no support on that subspace, leading again to the drawback of more potentially unnecessary computations, which we in this case, however, can control via the size of the blocks. We can reduce memory-pressure and latency even more by pre-fetching the data point itself into registers.

We, at first, implemented the surplus scatter and gather steps on the CPU together with the rest of the preparation, but, as one drawback of the subspace-linear algorithm is the excessive memory requirement for large subspaces, the cost of memory transfer posed to be too high to do so. To counteract this, we decided to transfer only non-NaN surplus values in combination with their respective index and initialize, unpack and pack the surplus arrays on the GPU using separate kernels. For adaptive sparse grids, this effectively reduces the data to be transferred, for full subspaces, it is doubled. As we employ spatially adaptive sparse grids to reduce grid points, it is implicit that larger subspaces contain, in general, only a small subset of their potential grid points and thus this, expectedly and in practice, does not pose an issue.

For the actual implementation, we used the features provided by the SG++ library into which we also integrated our algorithm. The preparation steps, namely creating and sorting the list of subspaces as well as preparation for the surplus arrays, is done entirely on the CPU. To initialize the surpluses we employ three different kernels, two in succession for each algorithm. The first kernel initializes all surplus arrays to NaN, while the second kernel, depending on the algorithm, initializes the surpluses specified by the given indices either to the specified values, in case of $B^\intercal \boldsymbol{\alpha}$, or zero, in case of $B\mathbf{v}$. Using CUDA streams, the analog to OpenCL command queues, we can schedule the initialization kernels to run parallel to memory transfers required for the dataset and subspace list. Note that in case of the operator $B\mathbf{v}$, a write-back of the surpluses is required and can be done in the same kernel as used for the operator, as, in this case, no kernel-execution to memory-transfer parallelism can be exploited.

While both operators are fairly similar to implement, the parallelization of $B\mathbf{v}$ over the data points requires synchronized access for the summation of the result. Beginning with devices of compute capability 6.0, and thus the Pascal architecture, NVIDIA introduced true atomic 64 bit floating point (FP64) addition, a feature which has existed on previous generations for 32 bit floating point (FP32) values only and has also been restructured on the newer architecture to achieve a better performance. We use these instructions when provided by the architecture and fall back to the slower, well-known, compare-and-swap simulation of these instructions when not.

To mitigate the fact that CUDA does not support device-code compilation at run-time, we made heavy use of C++ templates. Dimensionality and block-size are specified by template-parameters, allowing loop-unrolling via a simple, recursive expanding template[1] accepting a lambda-function. Small in-place lambda functions are generally inlined by the compiler, even more so by NVIDIA's `nvcc` CUDA compiler, as it generally inlines almost any function due to the GPU architecture. We did not encounter any issues with this solution except for

[1] Recursive due to our limitation to the C++11 standard. With C++14, or alternatively a custom implementation of template-index-sets, we could reduce some pressure on the compiler by employing an iterative solution.

the obvious drawback of the required explicit template instantiation for each combination of dimensionality and block-size, leading to an increased compile-time for this specific source file and a slightly larger shared-object size.

4.2 Subspace-Optimal Regression on GPUs

There are two major drawbacks to the subspace-linear algorithm: its overhead for the index calculation and its significant memory requirement for large subspaces, the latter sometimes even prohibiting execution on the GPU due to lack of on-device memory. On spatially adaptive sparse grids, these large subspaces, in general, contain only a fraction of their potential basis functions, thus most of the memory serves no other purpose than allowing for easy indexing of the corresponding surpluses. Additionally, as the streaming algorithm evaluates all, actually existing, basis functions for a single data point, the number of unnecessary evaluations is effectively low for a subspace if it does not contain many basis functions. Consideration of the overhead the subspace-linear algorithm has per evaluation, and thus per subspace as there is only one evaluation per subspace, suggests the division of the workload between the subspace-linear algorithm, processing densely used subspaces, and the streaming algorithm, processing sparse subspaces.

As the algorithm itself, omitting the streaming and subspace-linear algorithms employed by it, is intrinsically simple, the only real degree of freedom from a design perspective is the division of the set of subspaces into the two subsets to be processed independently by the streaming and subspace-linear algorithms.

To find such a selector, selecting which subspace to process with which algorithm, let us begin by estimating the cost of evaluation for one such subspace S. For simplicity, we, for now, only look at the core of the algorithms and ignore other aspects, such as preparation steps, memory transfer and initialization. With this we can express the approximate costs

$$C_S^{\text{lin}} = C_{\text{eval}}^{\text{lin}} + C_{\text{so}}^{\text{lin}} \tag{4.1}$$

$$C_S^{\text{str}} = |S| \cdot C_{\text{eval}}^{\text{str}} \tag{4.2}$$

where

- S is the subspace to be evaluated,
- $|S|$ is the number of basis functions, i.e. grid points, in S,
- C_S^{lin} is the cost of subspace S when using the subspace-linear algorithm,
- $C_{\text{eval}}^{\text{lin}}$ is the cost of a single basis-function evaluation with summation of the result when using the subspace-linear algorithm,
- $C_{\text{so}}^{\text{lin}}$ is the cost of the overhead per subspace, e.g. loading of the skip-pointer and similar, when using the subspace-linear algorithm.
- C_S^{str} is the cost of subspace S when using the streaming algorithm, and
- $C_{\text{eval}}^{\text{str}}$ is the cost of a single basis-function evaluation with summation of the result when using the streaming algorithm.

Note that $C^{\text{lin}}_{\text{eval}}$, $C^{\text{lin}}_{\text{so}}$, and $C^{\text{str}}_{\text{eval}}$ depend on the cost of memory load operations which we, without further knowledge, cannot quantify, but we generally expect $C^{\text{lin}}_{\text{eval}} \geq C^{\text{str}}_{\text{eval}}$ to hold, due to the streaming algorithm having coherent memory access across all threads. The only dependence of Equations (4.1) and (4.2) on the subspace S is, in theory, by its size and thus we can try to find a value $|S| = p$ so that $C^{\text{lin}}_S = C^{\text{str}}_S$, ultimately allowing us to decide which algorithm to choose for the subspace by a simple comparison of its size with this threshold value. We could now further break down the individual costs by counting the respective operations and create a detailed model, however, this would only introduce more unknowns depending on both, the architecture of the GPU and the scenario (dataset, grid, etc.) used, that we cannot specify with certainty. A more practical method to deduce the threshold value p is to use basic parameter tuning.

The use of a simple grid point-based threshold p does not solve the problem of large, mostly unused subspaces. To this end, we propose the use of a secondary threshold u based on the utilization of the subspaces, i.e. the number of actually used grid points $|S|$ in relation to the maximum size of this subspace $|\hat{S}|$. This allows us to process large but sparse subspaces using the streaming algorithm, while still processing large dense subspaces with the subspace-linear algorithm. We again suggest parameter tuning to derive the optimal value for u.

Combined, both thresholds yield the selector

$$\text{alg}_{p,u}(S) = \begin{cases} \text{streaming} & \text{if } |S| < p \text{ or } \frac{|S|}{|\hat{S}|} < u \\ \text{subspace-linear} & \text{otherwise} \end{cases} \tag{4.3}$$

where

S	is the subspace to be evaluated,
$\lvert S\rvert$	is the number of basis functions, i.e. grid points, in S,
$\lvert\hat{S}\rvert$	is the maximum number of grid points that could potentially be in S,
p	is the grid-point threshold, and
u	is the utilization threshold.

5 Evaluation of Regression algorithms on the GPU

In this chapter, we present the evaluations of our, in the previous chapter discussed, implementations for the subspace-linear and subspace-adaptive algorithm. To this end, we begin by describing our evaluation process. We present the hardware as well as the data sets and scenarios on which we ran our evaluations, followed by the actual evaluations of those algorithms

All evaluations have been executed on the NVIDIA Tesla P100 GPU with 64 bit floating-point and integer precision, unless stated otherwise, employing the following scheme: For all evaluation scenarios, grids have been prepared ahead of time using the streaming algorithm for adaptive refinement. All intermediate grids as well as the base grid were stored after each refinement step of an adaptive scenario, ensuring complete comparability between different algorithms as this allows us to use the same grids for all algorithms, making the grids independent from the algorithm being evaluated. Evaluation of an algorithm and operator on a single grid then consists of a single initial run to force library initialization (in case of CUDA) and kernel compilation (in case of OpenCL), followed by five consecutive full runs, each including preparation, of the algorithm (for the specific operator), their results being averaged to reduce the impact of external influences.

5.1 Hardware Used

For our evaluations we used two test systems, shown in Tables 5.1 and 5.2. Both systems are dual-socket server systems. The NVIDIA Tesla P100 GPU is described in more detail in Section 1.3.2.

5.2 Data Sets and Scenarios

In this chapter, we will present the data sets and scenarios we will later on use for our evaluations. We employ a total of four different data sets, one containing real-world astrophysical data, one generated by simulation of particles in high energy physics, and two based on Gaussian random distributions. As appropriate, we use multiple scenarios, e.g. different parameter configurations for spatially adaptive refinement, per data set, keeping them largely identical between data sets.

CPU	2 × Intel Xeon E5-2620 @ 2.00 GHz
GPU	NVIDIA Tesla P100
Architecture	Pascal
Memory	16 GiB @ 4 × 180 GiB/s
CUDA Cores	3584 Cores @ 1328 MHz
CUDA Compute Capability	6.0
GFLOPS 64 bit	5300

Table 5.1: Primary Evaluation System

CPU	2 × Intel Xeon E5-2650 @ 2.60 GHz
GPU	NVIDIA Tesla K20Xm
Architecture	Kepler
Memory	6 GiB @ 250 GiB/s
CUDA Cores	2688 Cores @ 732 MHz
CUDA Compute Capability	3.5
GFLOPS 64 bit	1311

Table 5.2: Secondary Evaluation System

5.2.1 Data Sets

The Fifth Data Release of the Sloan Digital Sky Survey (DR5)

Data release 5 (DR5) of the Sloan Digital Sky Survey (SDSS) [AAA+07] is an astrophysical collection of survey quality photometric data taken through June 2005 at the Apache Point Observatory in New Mexico. It contains spectroscopic data for 674 741 galaxies, 90 596 quasars, and 215 781 stars, collected over an area of 5713 square degree. This data is given by the magnitudes of five bands, u, g, r, i, and z, spanning the range of wavelengths from 300 nm to 1000 nm, with the corresponding redshift of that stellar object.[1] Cosmological redshifts are a phenomenon largely based on the Doppler effect, i.e. that occurs when the wavelength of electromagnetic waves (including light) is shifted towards the red spectrum based on movement of the source of these waves away from the observer. While there are multiple other factors influencing this shift, such as gravity, they are commonly used to determine, amongst others, velocities and parameterize masses of stars. As cosmological redshifts and many properties derivable from them are difficult to obtain, it has become popular to estimate them based on photometric quantities, e.g. the ones provided by SDSS DR5. This makes the data set a five-dimensional real-world scenario.

For the evaluation of our algorithms, we use a filtered version of the spectroscopic subset of DR5, as found in the literature by Heinecke and Pflüger [HP13] and Pfander, Heinecke, and

[1] For comparison: The visible light spectrum ranges from approximately 400 nm to 700 nm.

Pflüger [PHP16], containing 371 908 data points. Further details regarding the selection of entries, as well as insights into the structure of this data set are given by Pflüger [Pfl10]. The data set itself is interesting as its combination with spatial refinement results in highly irregular adaptive sparse grids.

The HIGGS Data Set

High energy particle collisions, such as created by the Large Hadron Collider (LHC), contain the desired exotic particles only in a small subset of the observed collisions. The Higgs boson, for instance, can only be observed in approximately 300 of 10^{11} collisions [BSW14], and it is thus necessary to classify the created data as significant or insignificant with respect to the desired particles. The HIGGS data set [BSW14] is a binary classification that has been created using monte carlo simulations and is intended as a benchmarking data set for the accuracy of this classification of processes, specifically the detection of Higgs bosons.

The data set consists of 28 features, 21 kinematic properties measurable by particle detectors in the accelerator and seven manually derived combinations thereof, developed to aid classification. The original data set contains 11 million data points. To reduce load- and evaluation times to, for our purposes, more acceptable levels, we truncated it to 5 million data points and normalized it to fit in the hypercube $[0, 1]^{28}$.

This data set is particularly interesting as it is a fairly high-dimensional data set that, similar to the DR5, results in the creation of very irregular adaptive sparse grids with a large amount of subspaces when combined with spatial refinement.

The Gaussian Data Sets

The Gaussian data sets are custom generated data sets based on multiple multivariate Gaussian (normal) distributions. For a d-dimensional Gaussian data set, we scatter d Gaussian distributions in the hypercube $[0, 1]^d$, their center being determined by a uniform random distribution. To enforce smoothness, we require a minimum distance between their center points of 32 times the standard derivation σ of the Gaussian distributions, which is at maximum 0.1 and is decreased automatically if this distance cannot be ensured. The data points are divided equally between the individual distributions. As target value we chose the probability density of the respective Gaussian distribution at the sample point.

For our evaluations, we generated two instances of this data set type, one five-dimensional and one ten-dimensional, each containing one million data points. Both data sets lead to somewhat more regular grid structures, compared to the DR5 and HIGGS data sets, when combined with adaptive refinement.

Level	# Points	# Subgrids	Largest Subgrid (#Pt.)
2	11	6	2
3	71	21	4
4	351	56	8
5	1471	126	16
6	5503	252	32
7	18 943	462	64
8	61 183	792	128
9	187 903	1287	256
10	553 983	2002	512

Table 5.3: Statistics for the regular sparse grid scenario on the SDSS DR5 data set.

5.2.2 Scenarios

For reference and comparison of GPU architectures, we use regular sparse grids on the SDSS DR5 data set with varying maximum level, increasing from two to ten. Details for this scenario can be found in Table 5.3. This is the only scenario for regular sparse grids, as the focus of this thesis lies on spatially adaptive sparse grids.

To analyze the performance of our algorithms, we chose multiple scenarios with different base levels for refinement, whenever the dimensionality of the corresponding data set allowed us to do so in a reasonable manner. All grids of the scenarios were created with the OpenCL streaming algorithm provided by SG++ and stored prior to the evaluation, allowing us to create results comparable in between algorithms.

For scenarios with a base level of two, we used the same parameters as provided by Pfander, Heinecke, and Pflüger [PHP16] to allow for comparability, while only slight modifications to this configuration were made for other scenarios. The scenarios with base level two have been chosen to create mostly irregular grids, whereas the other scenarios were created to provide an increase in regularity over those. An overview of the most important parameters can be found in Table 5.4, details regarding the resulting grids and their structure can be found in Appendix A.

5.3 The Subspace-Linear Algorithm

For the performance evaluation of our implementation of the subspace-linear algorithm, we use the data sets and scenarios described in Section 5.2 and the hardware described in Section 5.1. We compare our implementation with the parameter-tuned and highly optimized streaming algorithm provided by SG++ [HP13].

Dataset	Base Level	# Refinements	# Points Refined	Max. CG Iter.	λ
DR5	2	30	80	120	1×10^{-5}
DR5	5	20	100	150	1×10^{-6}
DR5	7	15	100	150	1×10^{-6}
Gaussian (5D)	2	30	80	120	1×10^{-5}
Gaussian (5D)	5	20	100	150	1×10^{-6}
Gaussian (5D)	7	10	100	150	1×10^{-6}
Gaussian (10D)	2	20	80	120	1×10^{-5}
Gaussian (10D)	5	10	100	150	1×10^{-6}
HIGGS (5M)	2	15	80	120	1×10^{-5}

Table 5.4: Choice of parameters for the spatially adaptive sparse grid scenarios.

5.3.1 A Note on Regular Sparse Grids and Atomics

Although our main interest is in spatially adaptive sparse grids, let us begin this section by verifying some of our expectations with respect to regular sparse grids, using the SDSS DR5 data set. As the streaming algorithm performs $N \cdot M$ evaluations (N being the number of grid points, M being the number of data points), and thus exponentially more unnecessary evaluations with increasing grid level, we expect a continuously growing speed-up for the subspace-linear algorithm and a sufficiently large base level. We can indeed verify this for the operator $B^\intercal \boldsymbol{\alpha}$, see Figures 5.1 to 5.3, however, we can also observe that the run-time of our implementation of the subspace-linear algorithm is seemingly linear with regards to the grid points (Figure 5.2). Looking at the device kernel execution time only (Figure 5.4), we can see that the run-time of the core algorithm is more as expected. Considering at the composition of the run-time for the complete implementation (Figures 5.5 and 5.6), we can conclude that, for regular sparse grids, it is dominated by the preparation step. This step is indeed linear with regards to the number of grid points. The implementation of the subspace-linear algorithm in SG++ requires us to iterate over all grid points to retrieve the set of subgrids assembling the sparse grid, as SG++ essentially only stores a list of grid points for a grid. Note, however, that for adaptive refinement, the preparation step has generally to be executed only once for possibly hundreds of solver iterations, reducing its impact on the run-time severely. Memory transfers are, in this scenario, dominated by the data set transfer from host to device and thus negligible.

We are able to present a speed-up of approximately one for grids with level two to five and a continued gain for grids with a larger level, leading up to a speed-up of 7.5 for sparse grids with level ten, expecting further increase beyond. Comparing these numbers with the maximum possible speed-ups for the respective grids, derivable from Table 5.3, makes it evident that there is still a considerable amount of overhead involved, even when preparation is completely neglected. The level five grid, for example, has a theoretical maximal speed-up of approximately 11.7, as there are (approximately) 11.7 points per subgrid, but only a speed-up of approximately 1.2 can be observed. For level ten grids, we achieve, again without preparation, a speed-up of approximately 48, with a theoretical

maximum of (approximately) 278 (compare Table 5.3). While the reached numbers are small in comparison, the non-contiguous memory accesses required for this algorithm, as well as their latency, make it unlikely to reach this speed-up in practice.

We can also verify that, at least for the operator $B^\intercal\boldsymbol{\alpha}$, processing multiple evaluations in a single thread can lead to a slightly improved performance, likely due to the reduction of load operations required for the level vector (as a block shares the same level, thus it only needs to be loaded once for a block instead of once per thread) and increased instruction level parallelism.

For the transposed multi-evaluation $B\mathbf{v}$ Figures 5.7 to 5.9, we can observe mostly the same, except for two key differences. First off, the speed-up for smaller grid levels is significantly larger, ranging from two to three (Figure 5.8) where it is one for non-transposed the multi-evaluation (Figure 5.3). This is due to the fact that the streaming algorithm, for $B\mathbf{v}$, parallelizes over the grid points (whereas the subspace-linear algorithm parallelizes over the data points for both operators), and thus is not able to fully utilize the massively parallel architecture of the GPU for small grid sizes. The second difference is the exclusively negative impact of blocking, i.e. the grouping of multiple data point evaluations in a single thread (Figure 5.9). This is most likely caused by the use of an atomic addition to accumulate the results per grid point. A further impact of the atomic operation seems to be the decline in kernel execution time from grids with level three to grids with level five (Figure 5.9), which can be explained by more wide-spread atomic accesses for increased grid size, in other words, the congestion emerging from multiple atomic accesses to the same location is likely to be reduced by the increase of those locations.

Interestingly, no other impacts of atomics can be observed for the transposed multi evaluation on the NVIDIA Tesla P100 GPU. We can attribute this fact to the 64 bit hardware implementation of these atomics on the Pascal architecture, a serious benefit of this architecture over older architectures, such as Fermi and Kepler, where such operations where partly implemented in software [NVI16]. If we compare our evaluations on the Tesla P100 (Pascal architecture) with evaluations on the Tesla K20Xm (Kepler architecture), we can clearly observe this difference. The operator $B^\intercal\boldsymbol{\alpha}$ is on the K20Xm largely identical in run-time behavior to the P100 (compare Figures 5.10 and 5.11 with Figures 5.1 to 5.6), whereas a serious performance impact can be observed for the operator $B\mathbf{v}$ (compare Figures 5.12 to 5.14 with Figures 5.7 to 5.9) due to software-emulated atomics. Interestingly, we can see a decline in overall execution time with increasing grid size up to grids of level six(Figure 5.12). An explanation of this behavior is the increase of operations, due to which warps can be scheduled more freely, reducing access to the same elements at the same time, and thus reducing contention of the atomic operation. When looking at the device kernel times for $B\mathbf{v}$ on the K20Xm (Figure 5.14), we can observe that the block-size severely impacts the results as well. With increasing block-size, the execution time is reduced. This is due to the inherent nature of processing multiple evaluations in a single thread: Instead of processing all atomic additions in parallel, the additions of a block are processed subsequently, reducing the number of threads accessing the same elements atomically, again effectively reducing contention. Note, that we had to emulate the 64 bit floating point additions on the Kepler

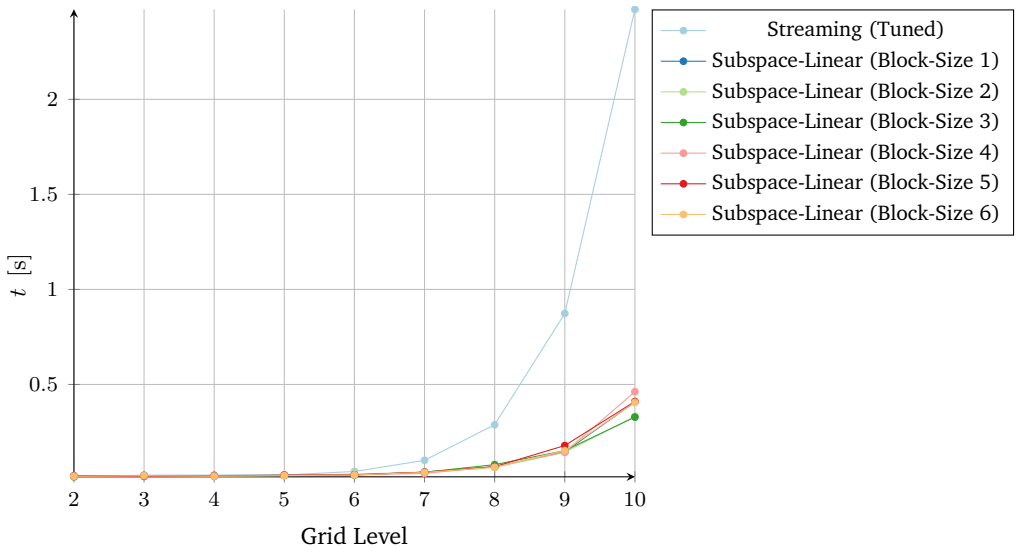

Figure 5.1: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

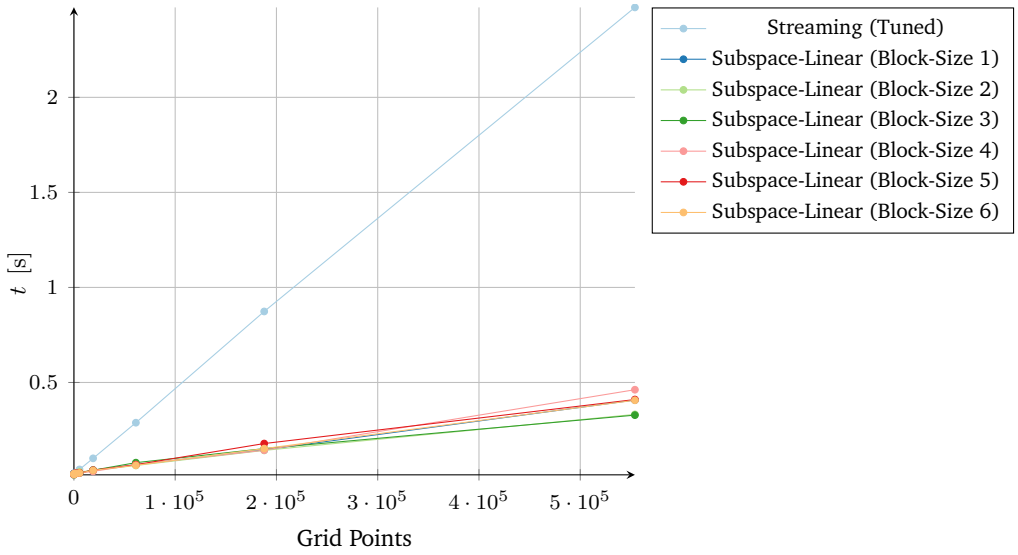

Figure 5.2: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid points.

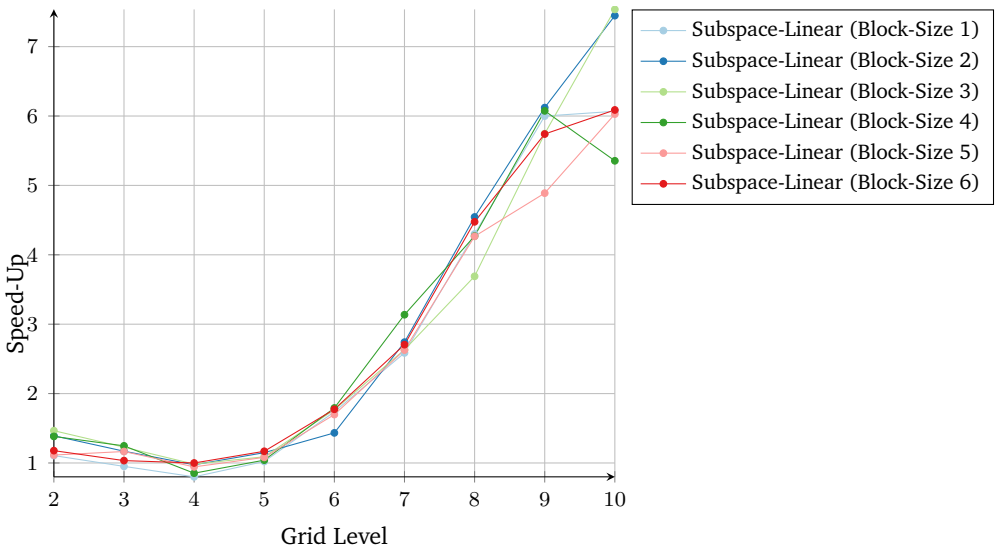

Figure 5.3: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

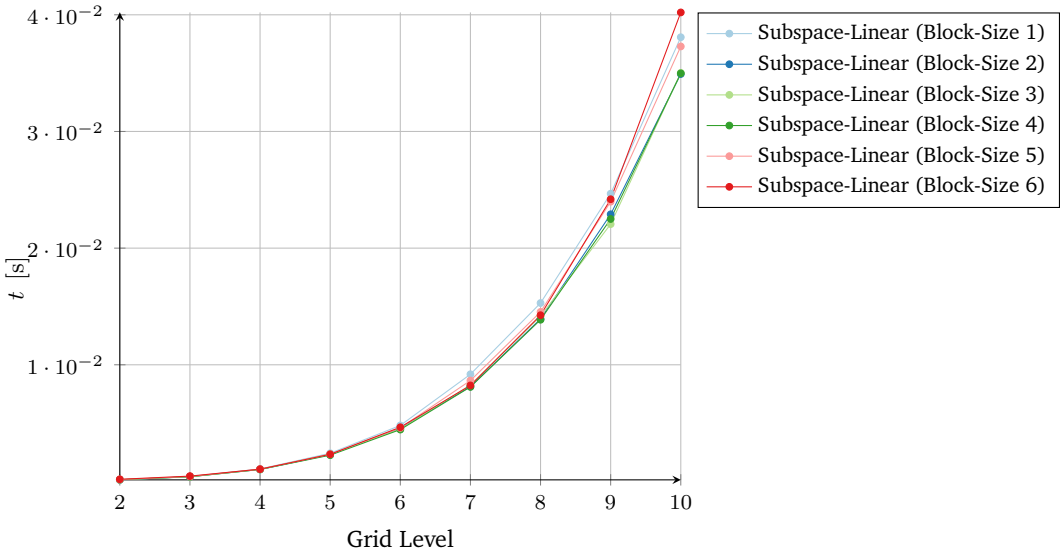

Figure 5.4: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

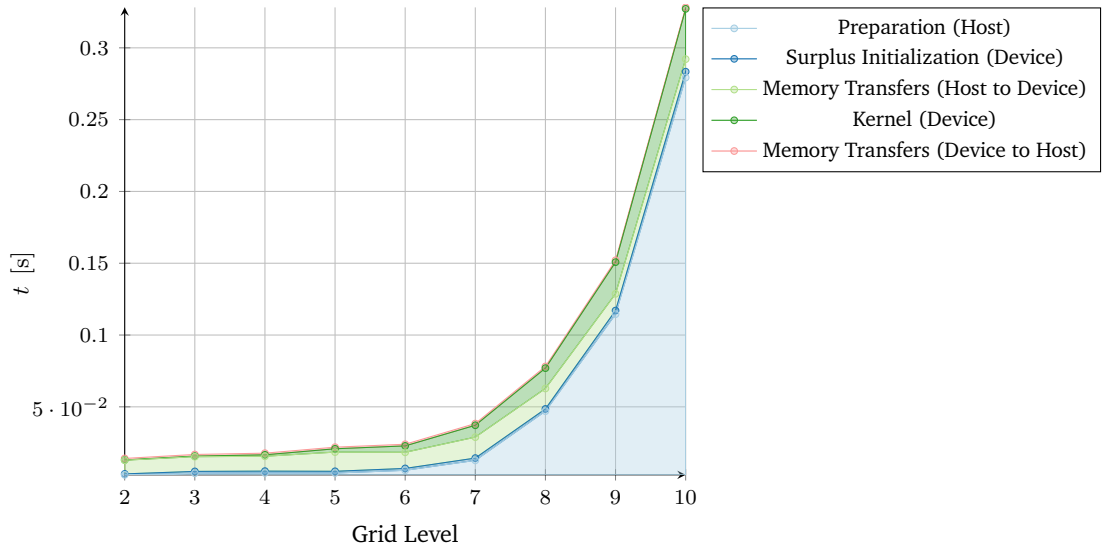

Figure 5.5: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of three) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

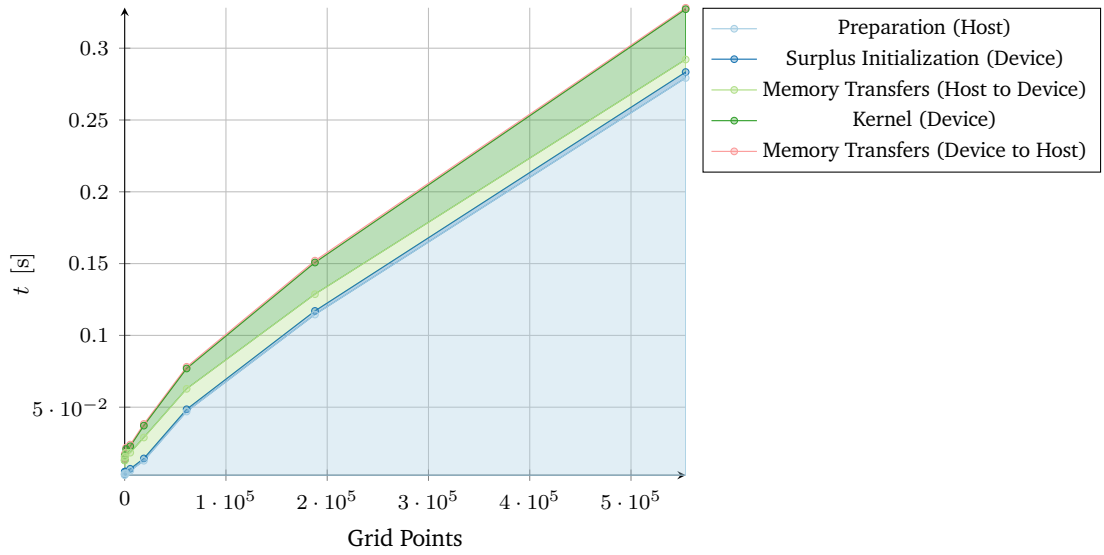

Figure 5.6: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of three) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid points.

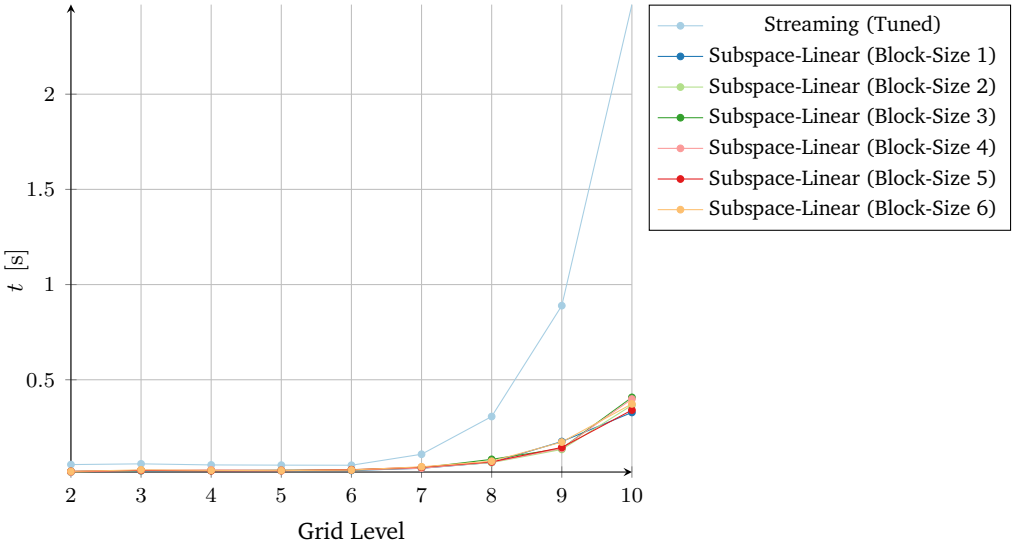

Figure 5.7: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

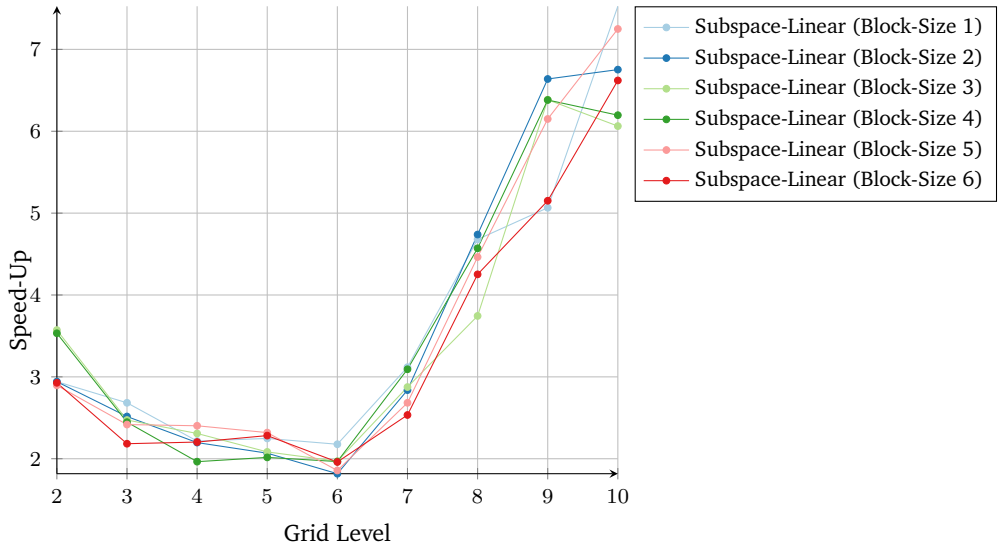

Figure 5.8: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

and other pre-Pascal architectures using a compare-and-swap (CAS)-loop as these architectures do not provide such operations, due to which the problem of contention is especially severe. For the streaming algorithm, this problem is solved by parallelization over the grid points instead of the data points, which is not directly applicable to the subspace-linear algorithm. The parallelization over grid points, however, can also become a problem for the streaming-algorithm on smaller grids, as a certain minimum of parallel tasks is required to fully utilize the GPU architecture. Nevertheless, the subspace-linear algorithm for $B\mathbf{v}$ is still faster than the streaming algorithm on large regular grids.

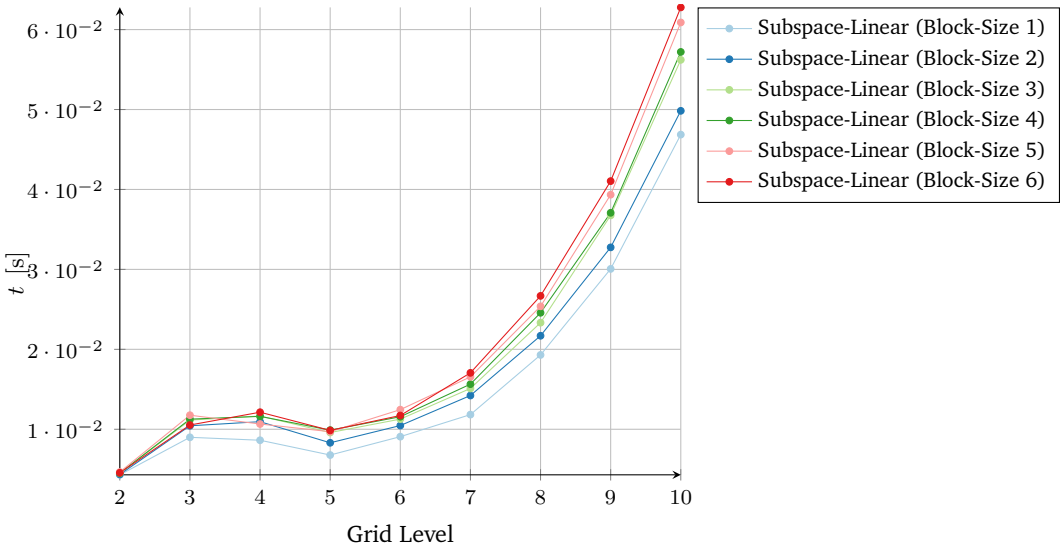

Figure 5.9: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

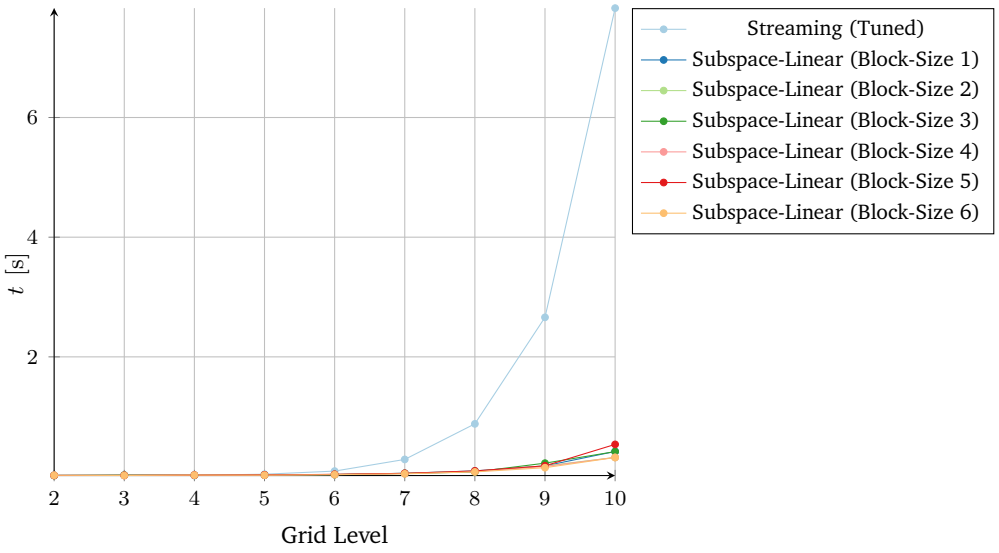

Figure 5.10: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^{\mathsf{T}}\boldsymbol{\alpha}$ on the NVIDIA Tesla K20Xm, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

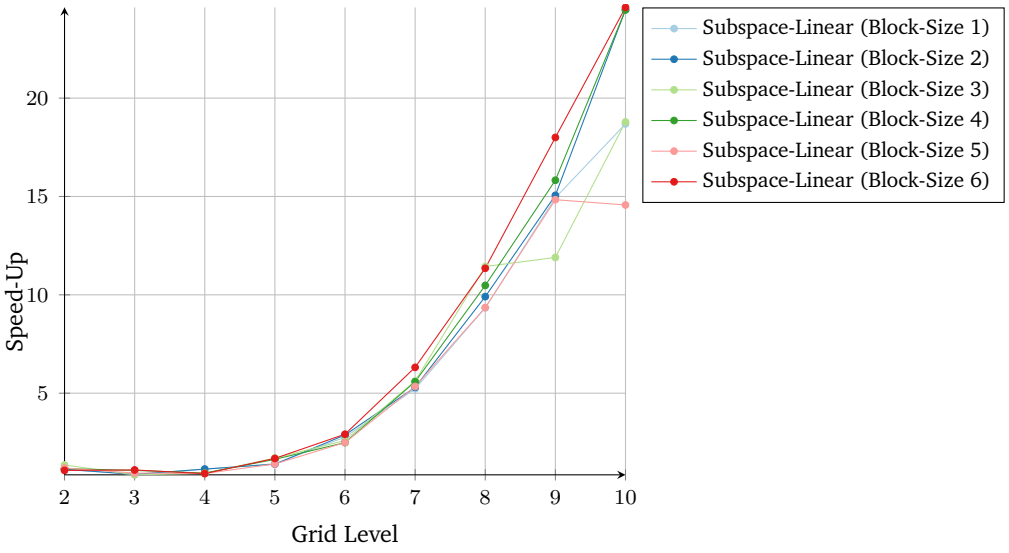

Figure 5.11: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla K20Xm, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

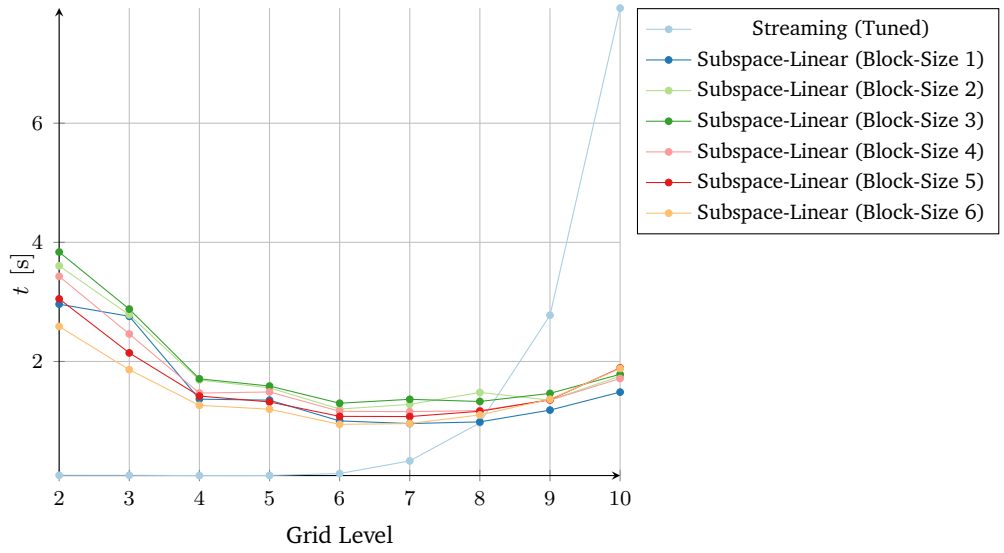

Figure 5.12: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla K20Xm, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

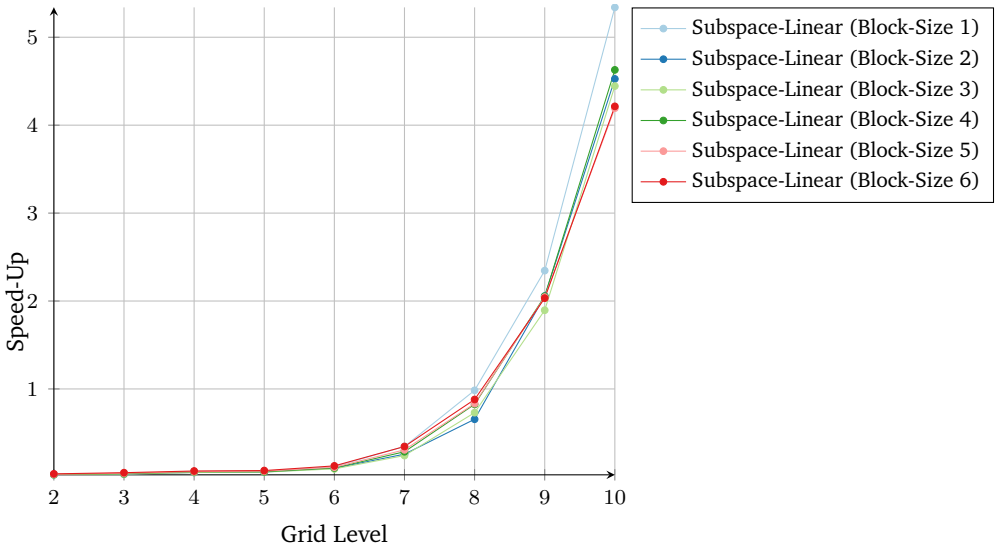

Figure 5.13: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla K20Xm, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

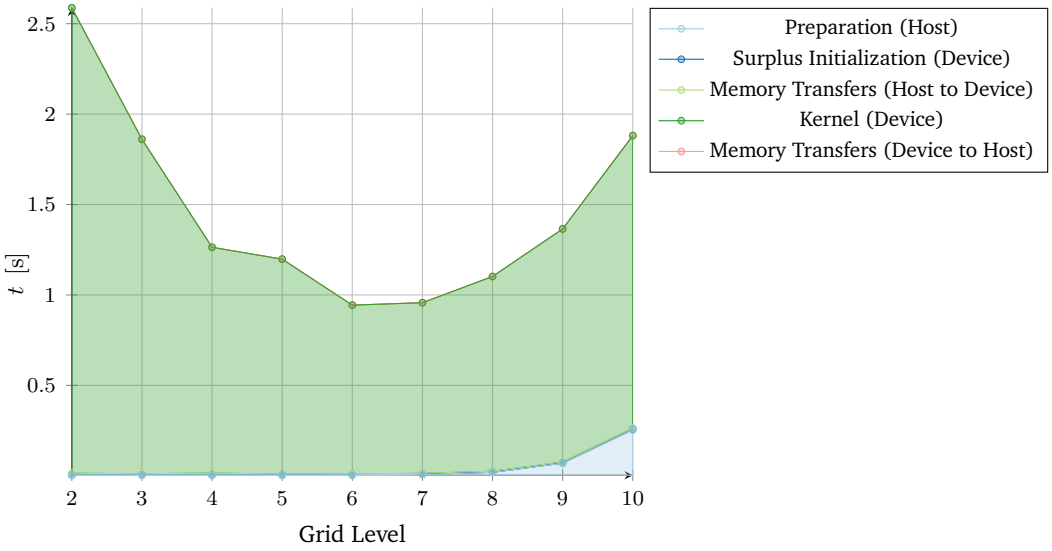

Figure 5.14: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B\mathbf{v}$ on the NVIDIA Tesla K20Xm, using regular sparse grids, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid level.

5.3.2 Spatially Adaptive Scenarios

On the SDSS DR5 Data Set with a Level Two Base Grid

Let us now look at some more interesting spatially adaptive scenarios. The first scenario is again based on the SDSS DR5 dataset and has been created through surplus-based refinement, beginning with a level two regular sparse grid. We ran 30 refinement steps, each refining 80 points with a maximum number of 120 solver iterations, combined with a regularization factor λ of 1×10^{-5}. This scenario is based on the one chosen by Pfander, Heinecke, and Pflüger [PHP16] and results in a very irregular grid structure, with only up to approximately 9.5 grid points per subspace on average (see Table A.1 in Appendix A for detailed statistics). Due to this, the scenario is relatively close to what we would expect a real-world worst case scenario for the subspace-linear algorithm to look like. Furthermore, the high number of refinement steps leads to a fairly large maximum level of the adaptive grid, resulting in big subgrids, with a maximum of 2^{24} grid points for the largest subgrid. We chose the number of refinement steps explicitly to force this issue, which allows us to observe the behavior of the subspace-linear algorithm when using a significant part of the available GPU memory, as its memory usage depends on the size of the respective full subgrids and not on the number of points actually used on these subgrids. Ultimately, this choice leads to a memory requirement of more than 16 GiB for refinement steps 29 and 30, exceeding the memory available on the P100, due to which it cannot be executed on this GPU for those grids.

Looking at the absolute duration of operator $B\alpha$ (Figures 5.15 and 5.16 including preparation), we can again see a speed-up over the streaming algorithm (Figure 5.17) that tends to increase with growing grid size, this time, however, only up to a certain point, with declines in between peaks, and a severe drop after refinement step 26. For almost all refinement steps, the speedup of the subspace-linear algorithm (in combination with an appropriately chosen block-size value) over the parameter-tuned streaming algorithm exceeds one. The reason for all of this can be found in Table A.1 in Appendix A: While, as expected, the average subgrid utilization, i.e. the number of points on the subgrid in relation to its potential maximum size averaged over all subgrids, drops with increasing refinement steps from 100 % for the fully regular subgrid to only about 2.5 % for refinement step 26, the average number of points per subspace increases from 1.38 to approximately 9 for the same range. This indicates that the refinement process, for this scenario, not only extends in depth, but also in breadth, which ultimately favors the subspace-linear algorithm, at least up to refinement step 26. The intermediate declines in the speed-up relate to significant increases in the number of potential grid points of all subgrids were full, created by the addition of (potentially) larger subgrids through refinement, in turn resulting in an increase of memory required to store the surpluses. Note that this does not necessarily create a drop in the average number of points per subgrid (again see Table A.1). Beyond refinement step 26, the number of points per subspace keeps increasing, but the performance drops severely. We can observe, that for these steps, the run-time of the algorithm is severely impacted by the surplus initialization step (Figure 5.19), which consists of two parts: the initialization of all surpluses of the full subgrids (not only the used parts, as we need full grids for indexing)

to NaN (in case of $B^\intercal\alpha$) or zero (in case of $B\mathbf{v}$), and the transfer and unpacking of the surpluses for used grid points. Note that the timings for this step also include the transfer of the surpluses, and thus the time for memory-transfer from host to device depicted in the run-time composition figures (e.g. Figure 5.19) is dominated by the data set (due to which it is largely constant). As there is no significant increase in grid points (in comparison to previous refinement steps) it stands to reason that this behavior is a side-effect of the excessive memory use. Through further measurements, we could indeed verify that the initialization to NaN is the problematic step, as the time required to unpack the surpluses is almost insignificantly small. If we look at the number of maximum grid points, i.e. the number of grid points that would be on that grid if all subgrids were full, which is also the number of surpluses we need to store, we can see that the grid for refinement-step 26 requires approximately 7.56 GiB, while the grid for refinement-step 27 requires 8.68 GiB. Note that we are using 64 bit floating point numbers to store these. Interestingly, this increase represents a change from less than half to more than half of the memory available on the NVIDIA Tesla P100, being required solely to store surpluses. Lacking further knowledge about the memory related hardware of the P100, except that it contains four 4 GiB 2nd generation high bandwidth memory (HBM2) dies, each connected, via two 512 bit memory controllers, to two 512 kB slices of the L2 cache, one slice per controller, we are not able to confidently state a concrete reason for this behavior. This issue, however, could potentially be resolved by writing to multiple locations in a single initialization thread instead of only one, effectively reducing parallel memory accesses.

Most of these observations can also be made for the operator $B\mathbf{v}$. A difference between operators is, except for influence of the block-size value which will be discussed later, the behavior of the speed-up over the streaming algorithm (Figure 5.22). For the transposed multi evaluation, this tends to decrease from the base grid until approximately refinement step 10, beyond which it fluctuates around 1.25 for a block-size of one. These fluctuations can again be explained by the changes in grid properties as described previously, the decline, however, is due to the implementation of $B\mathbf{v}$ for the streaming algorithm. We could already see this behavior in the regular scenario and the explanation is still the same: due to its parallelization over the grid points for this operator, it is not able to fully saturate the GPU for small grid sizes. This time, it only seems to be more extreme as the largest speed-up of the subspace-linear algorithm can be observed on those smaller grid sizes.

Just like in the regular scenario presented above, we can see a difference in kernel execution times related to the block-size used, with a block-size of five providing the best result for $B^\intercal\alpha$ (Figure 5.18) and again a block-size of one for $B\mathbf{v}$ (Figure 5.23), again most likely being caused by the use of an atomic addition in the implementation of $B\mathbf{v}$. An interesting difference to the regular scenario is the semblance of linearity the kernel duration expresses with relation to the grid points for higher refinement steps (Figures 5.18 and 5.23). This is due to a considerably larger increase in the number of subgrids with relation to the number of gridpoints being added for each of these refinement steps, which can also be seen in the average number of grid points per subgrids (again see Table A.1 for details), meaning that, on average, a new subgrid is introduced for only a few points refined, which we ultimately have to traverse or skip with our algorithm.

Again, the preparation step takes a significant amount of time and is included in all evaluations (except for sole kernel duration, of course). We again emphasize this because we would only have to prepare the algorithms once the grid changes, and thus only once for a larger number of solver iterations. For such applications, the expected speed-up would be even higher as the streaming algorithm does not require any significant preparation.

On the HIGGS Data Set with a Level Two Base Grid

With 28 dimensions, the to five million data points truncated HIGGS data set is fairly challenging, and we expect it to provide interesting insights into the performance of the algorithm for higher-dimensional data sets. We again used a surplus-based refinement strategy refining 80 points per step with a maximum of 120 solver iterations and a regularization factor of 1×10^{-5}, this time refining 15 steps. The high dimensionality of this data set causes a significant number of subgrids to be added for each step, leading to a total of 1.9 million subgrids after the last refinement step and a decrease of the average number of points per subgrids after the first step to around 2.2 for the last 8 steps. Again, we are not able to run the subspace-linear algorithm on the last grid, as this would require approximately 34.2 GiB, however, this would also have about 4.1 million grid points and thus almost as much grid points as data points. Further details for this scenario can be found in Appendix A Table A.4. Due to the inherently large execution times of this data set, we limited some of our evaluations to ten refinement steps only

In Figures 5.24 and 5.27 we can see that the high number of subgrids and the low number of points per subspace indeed lead to a considerably worse run-time of the subspace-linear algorithm when compared to the streaming algorithm. Interestingly, we achieve a speed-up greater than one for smaller grids (base grid and the first refined grid), and more importantly, the speed-up declines even though both grids use all available points and there is an increase in the average number of points per subgrid (see Figures 5.25 and 5.28 Nevertheless there are already 435 subgrids after the first refinement step while there are only 1681 points, likely being the reason for this drop. After further refinement steps, the speedup seems to converge to approximately 0.6 for both operators and best observed block-size value, which is again one for $B\mathbf{v}$. For $B^\intercal\boldsymbol{\alpha}$ this value is three, whereas it has been five for the SDSS DR5 data set, indicating that there is more disagreement inside the block, i.e. that, on average, a block processes multiple data points that are often not supported on the same subspaces leading to an increase in run-time with increased block-size. Looking at the complete run-time composition (Figures 5.26 and 5.29) we can see that, for this scenario, the preparation, memory transfer, and surplus initialization steps are largely insignificant compared to the raw device-kernel duration.

On the SDSS DR5 Data Set with a Level Five Base Grid

The next scenario is again based on the SDSS DR5 data set. We ran 20 surplus-based refinement steps, refining 100 points each, with a maximum of 150 solver iterations and a

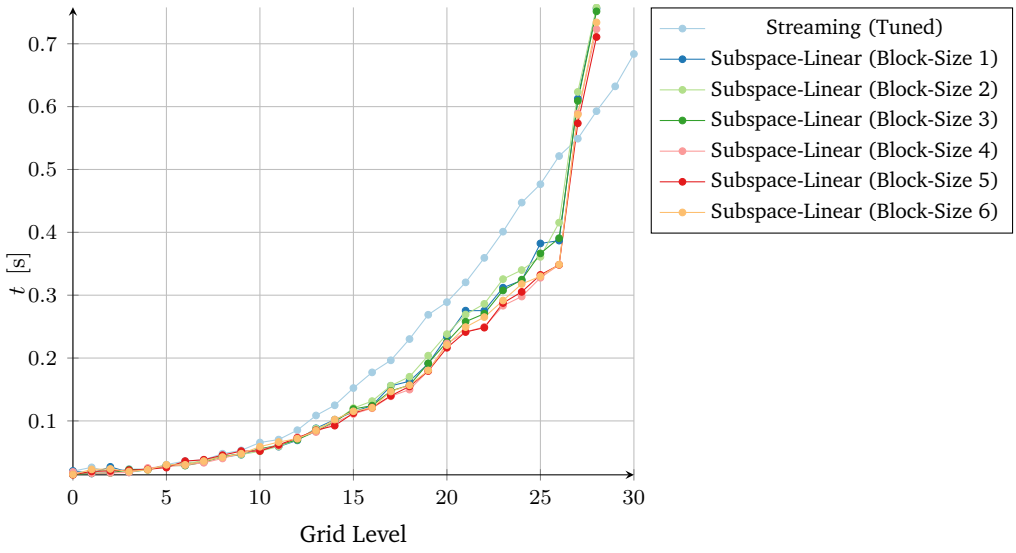

Figure 5.15: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

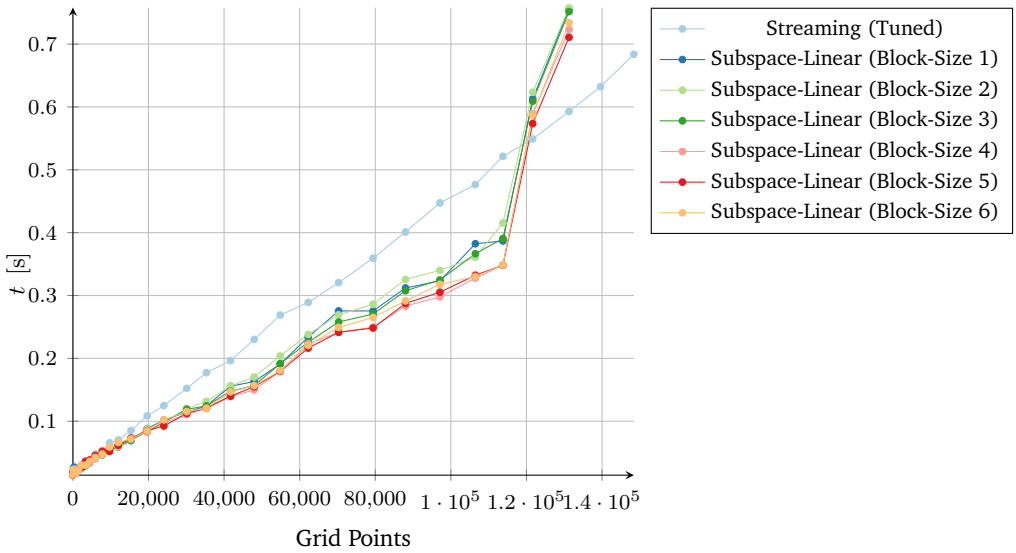

Figure 5.16: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid points.

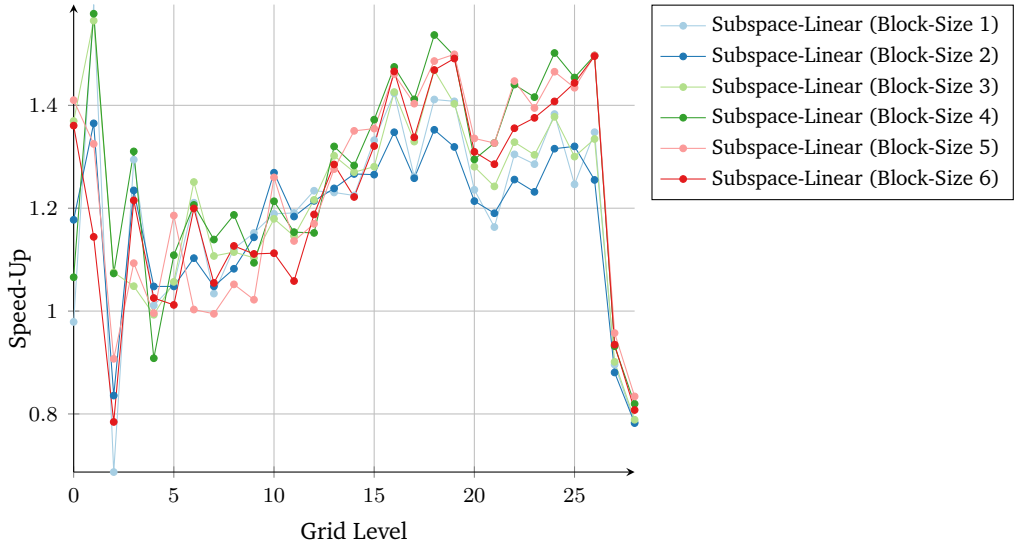

Figure 5.17: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

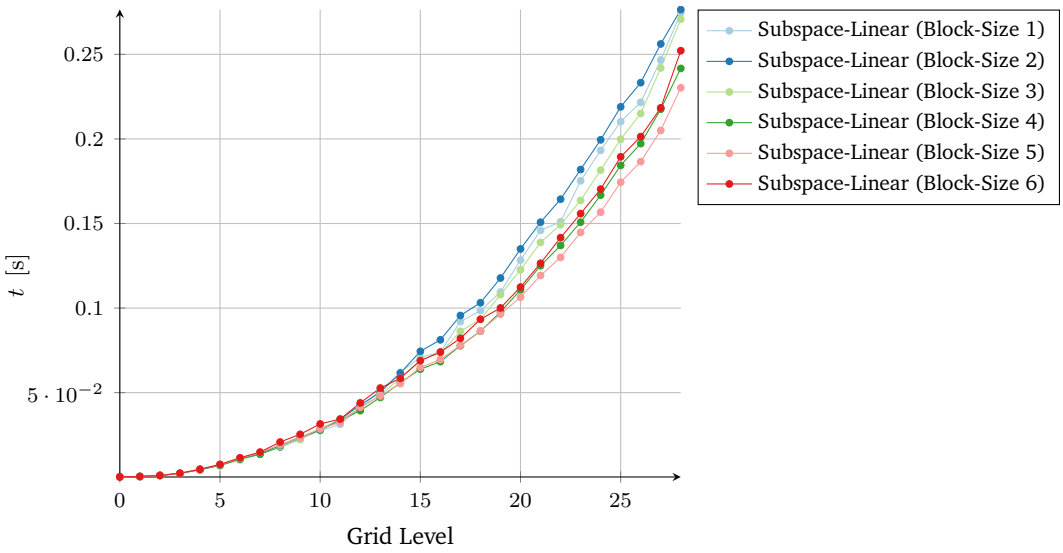

Figure 5.18: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

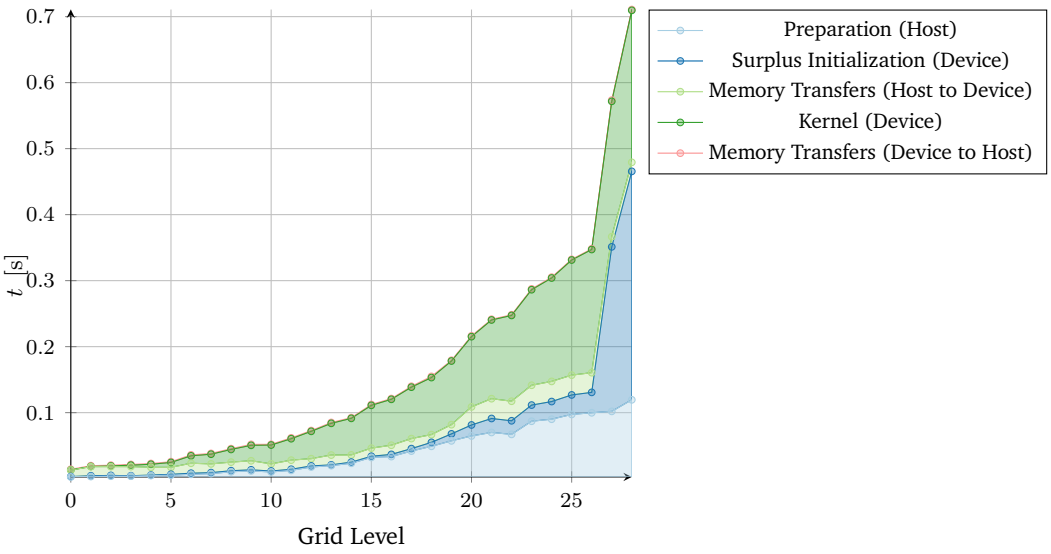

Figure 5.19: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of five) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

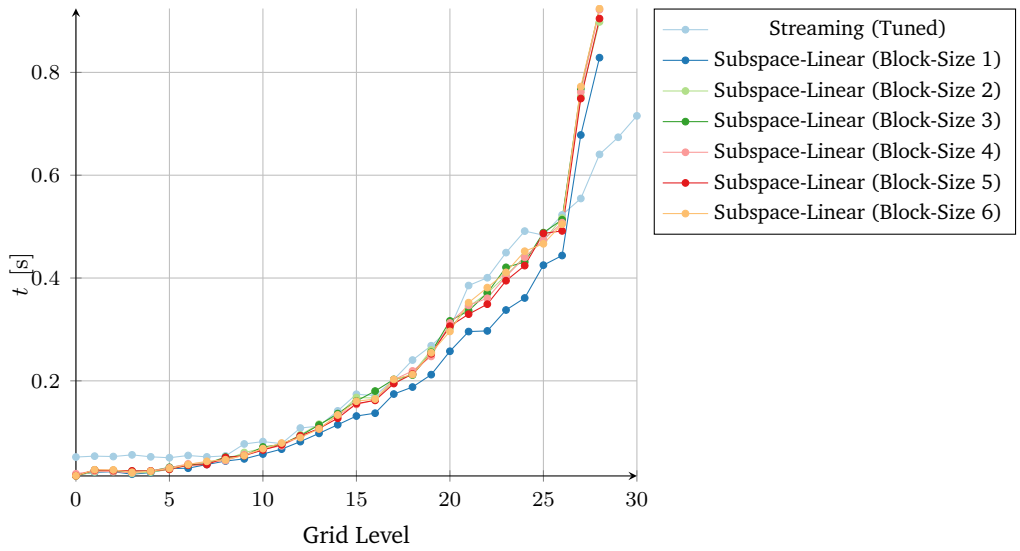

Figure 5.20: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

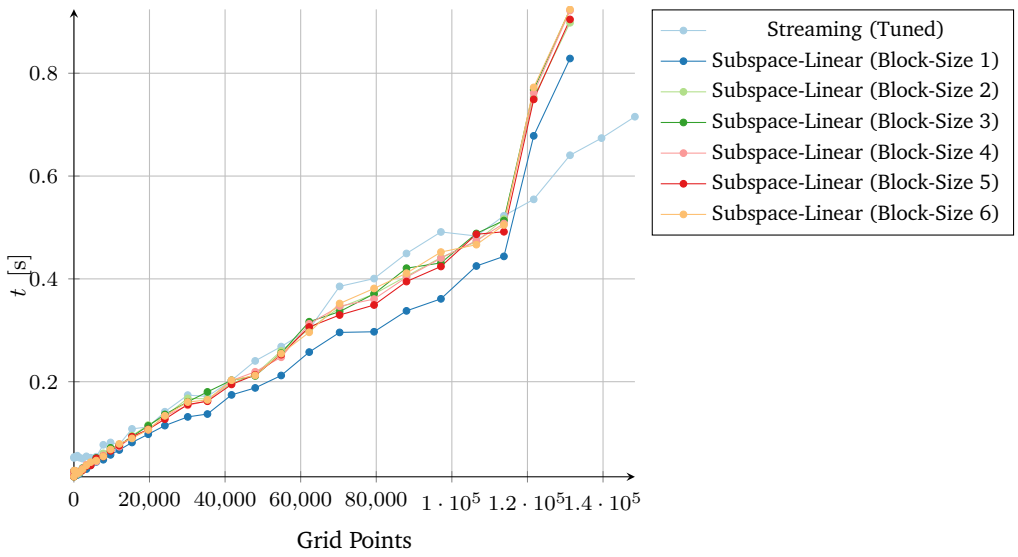

Figure 5.21: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the grid points.

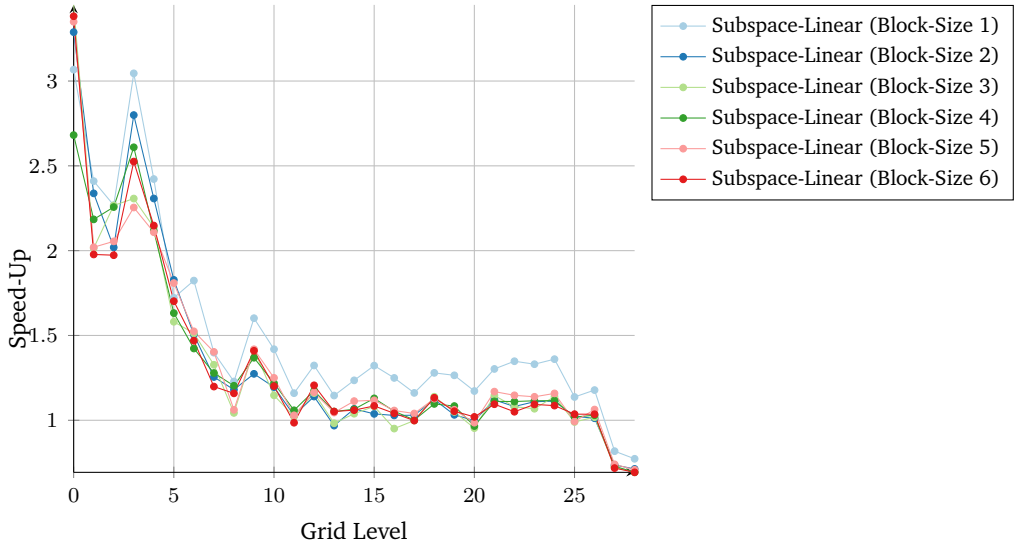

Figure 5.22: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

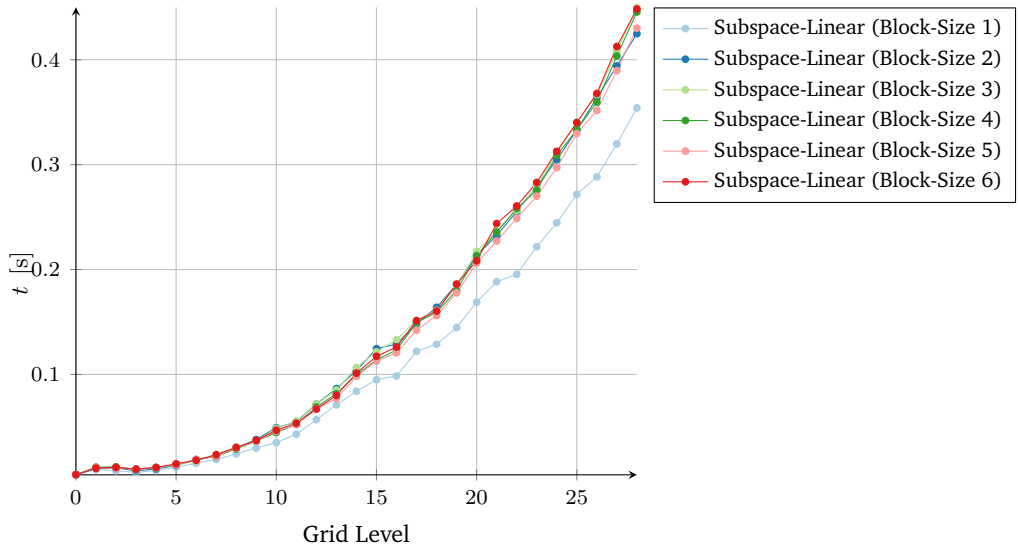

Figure 5.23: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

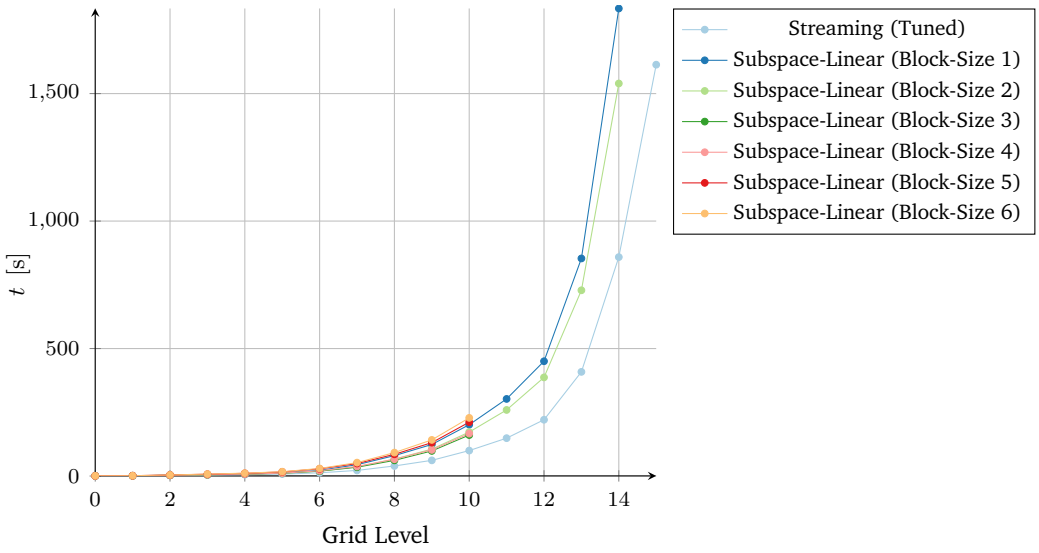

Figure 5.24: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

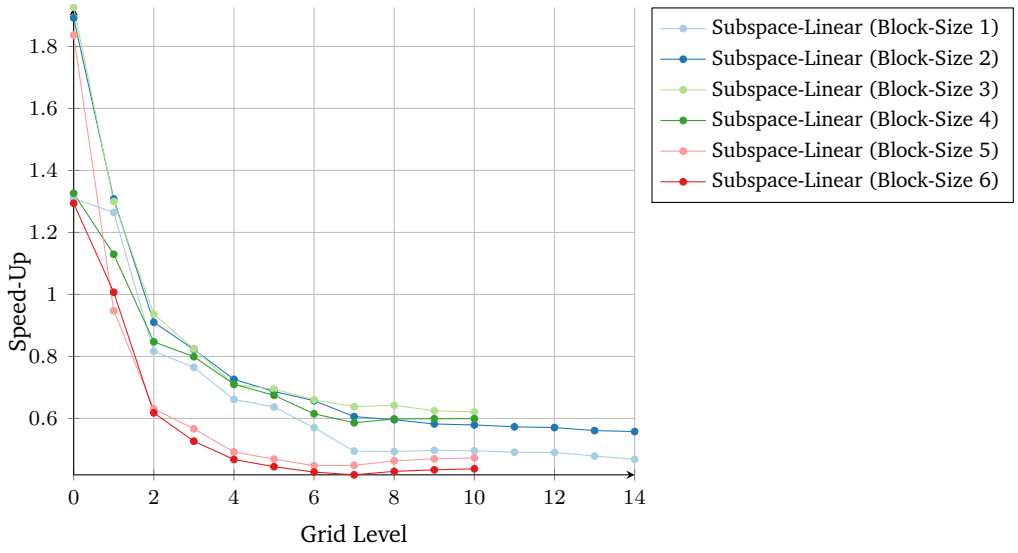

Figure 5.25: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

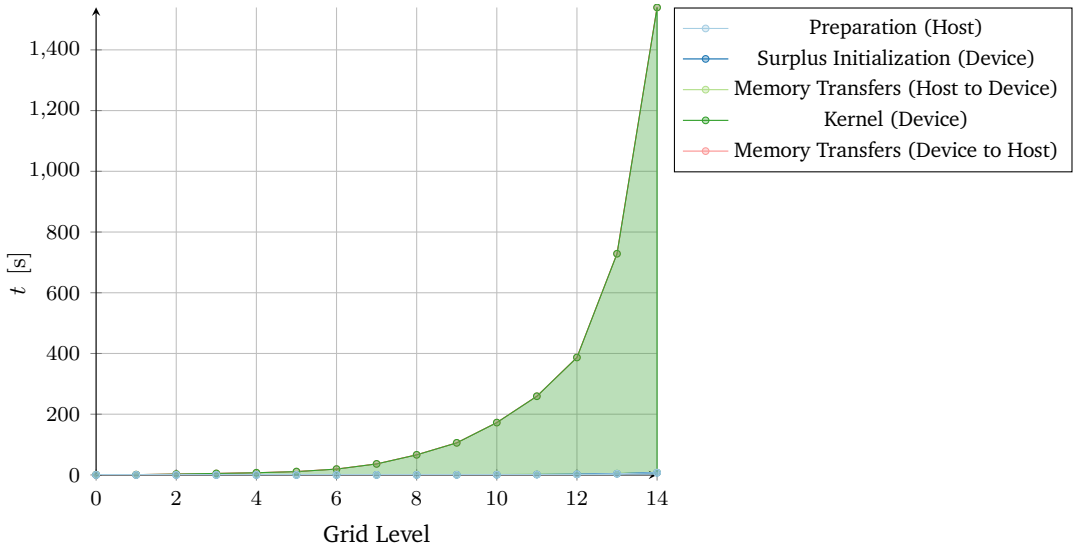

Figure 5.26: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of two) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

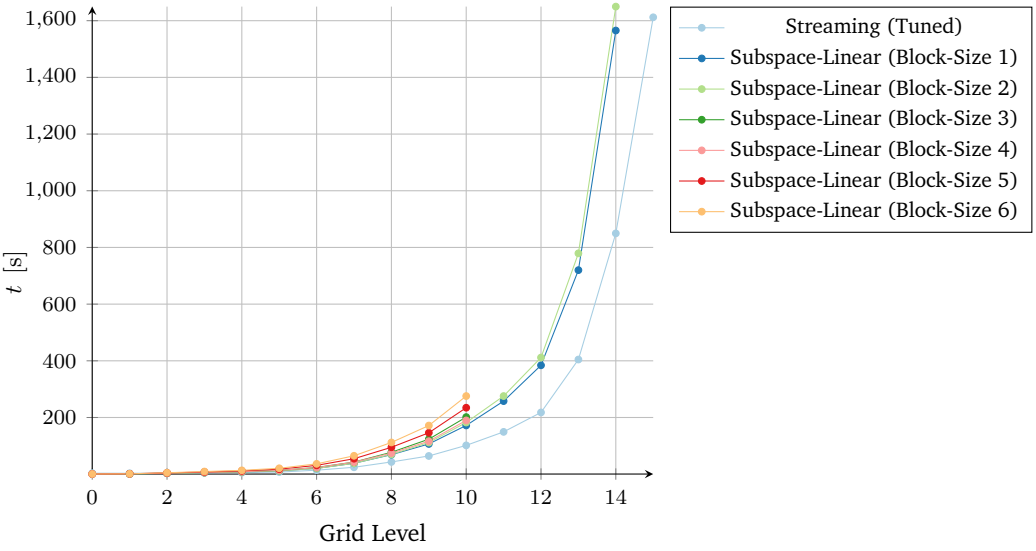

Figure 5.27: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

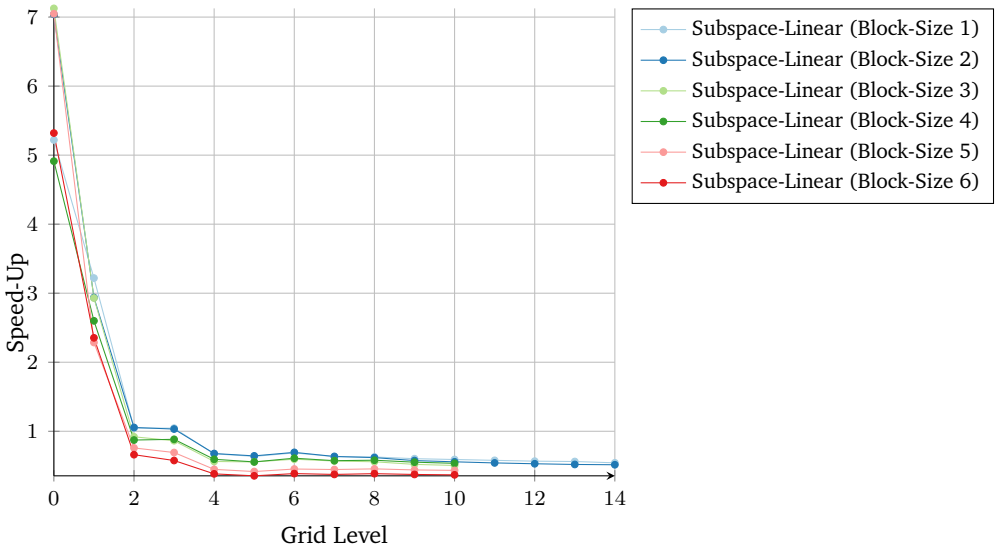

Figure 5.28: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

regularization factor of 1×10^{-6}. The behavior of this scenario is similar to the previous DR5 scenario with a base level of two (compare Figures 5.30, 5.31 and 5.32 to 5.34 with Figures 5.15 to 5.23). We can again observe the same impacts of block-size and grid structure (see again Appendix A for details), however, this time we are able to achieve a consistently higher speed-up for $B^\intercal\boldsymbol{\alpha}$ and a somewhat higher speed-up for $B\mathbf{v}$ over the streaming algorithm. This is expected, as we increased the regularity of the grid by using a higher base-level on which to start refinement, thus increasing the number of unnecessary operations the streaming algorithm makes, ultimately favoring the subspace-linear algorithm.

On the SDSS DR5 Data Set with a Level Seven Base Grid

Increasing the base-level from the previous scenario even further (all other parameters have not been modified, except for the reduction of refinement steps to 15), we can see a further increase in speed-up over the streaming algorithm (compare Figures 5.35 to 5.39 with Figures 5.30, 5.31 and 5.32 to 5.34), now constantly being over 1.5 for all refinement steps (and selected block-sizes), with a decline over those. This decline is caused by the percentage of the regular structures in the grid being reduced by the refinement steps. We can see this explicitly in the absolute duration of the algorithms (Figures 5.35 and 5.38), where this is represented as an offset in run-time.

On the Five Dimensional Gaussian Data Set with a Level Two Base Grid

Let us now look at the five-dimensional Gaussian data set, beginning again with a scenario using a base grid of two and the same parameters of the corresponding SDSS DR5 scenario, i.e. 30 refinement steps with 80 points refined per step, a maximum of 120 solver iterations and a regularization factor of 1×10^{-6}.

Compared to the DR5 scenario, we can see a higher speed-up (Figures 5.41 and 5.45), ultimately decreasing with higher refinement steps, again caused by the excessive amount memory required to store the surpluses (compare Figure 5.43 and Table A.5), and more importantly, a stronger effect of the block-size on the kernel run-time (Figures 5.42 and 5.46). This influence can be explained by the structure of the data set. The data set has been generated by subsequently adding random clusters, generated by a gaussian random distribution. Due to this, the data set is largely ordered, meaning that it is likely for points that are close to each other in the feature space and thus likely to share support on most of their subspaces, to be also close in the list of data points. For the subspace-linear algorithm, this leads to less subspaces to be processed in the block, as most data points agree on which subspaces to skip, leading to a decrease in run-time by the reduced load operations required to load the level vector (one per block instead of one per data point) and only little overhead by processing subspaces where data points have no support (as the data points of a block share most of them). Note that a higher block-size increases the possibility of warp divergence, however, the impact of this divergence would again be reduced through the locality of the points. Interestingly, this also changes the influence of the block size-with respect to the DR5 from

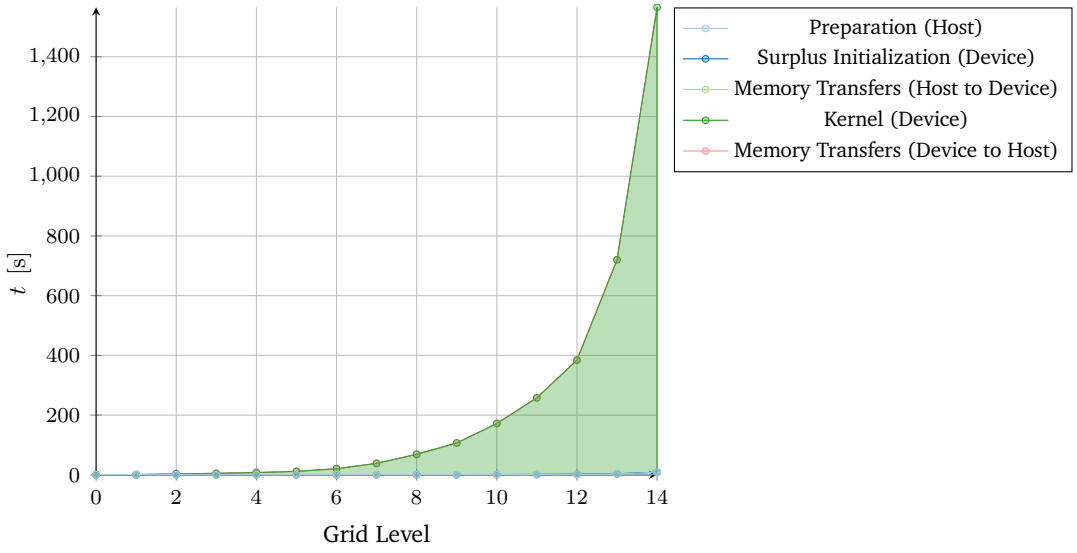

Figure 5.29: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of one) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS (5M) data set, and 64 bit precision. Plotted over the refinement steps.

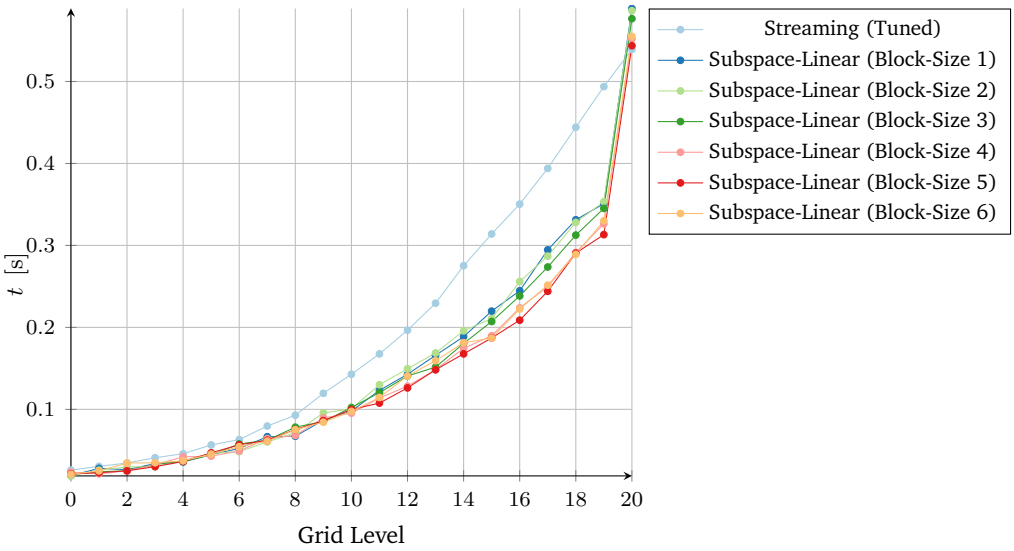

Figure 5.30: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^{\intercal}\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

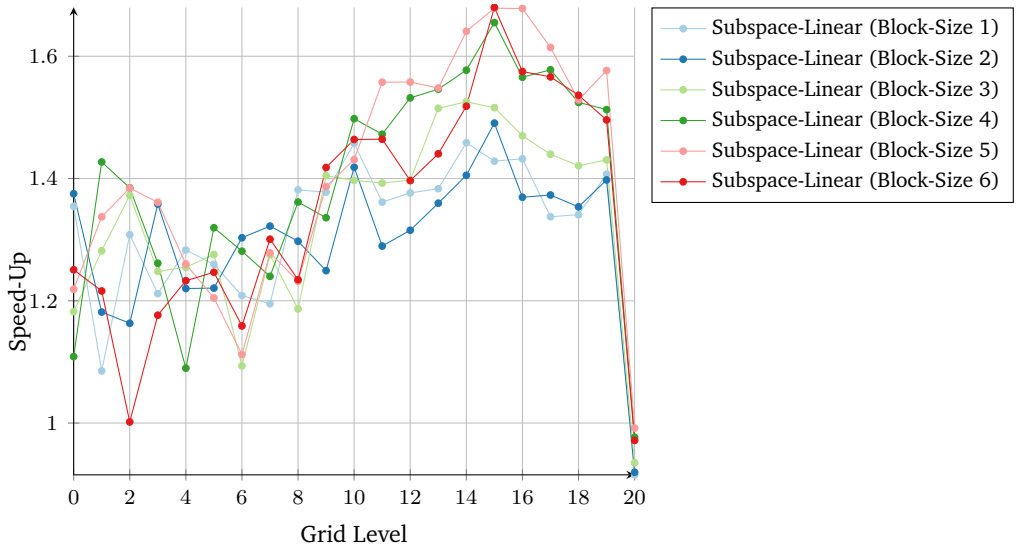

Figure 5.31: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

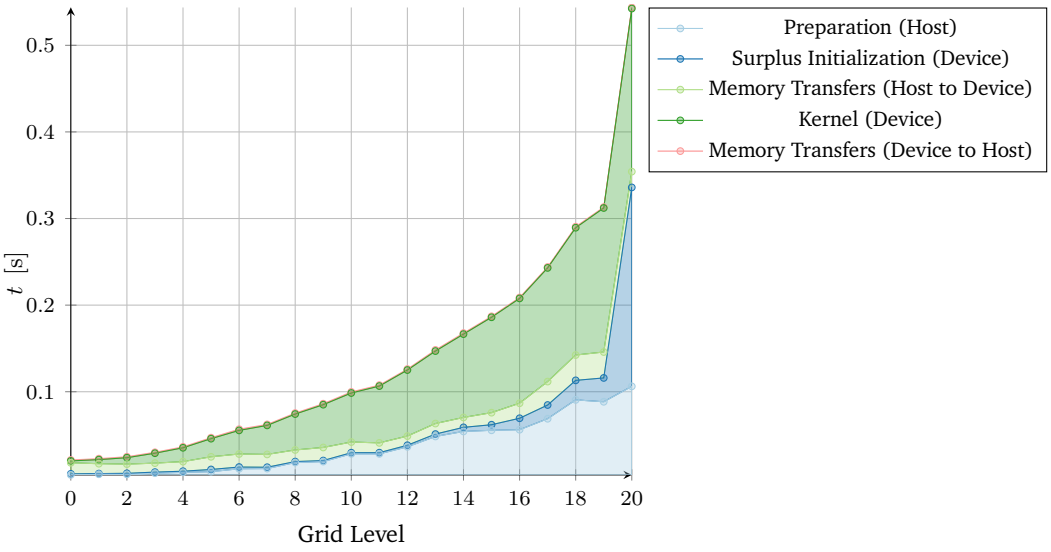

Figure 5.32: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of five) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

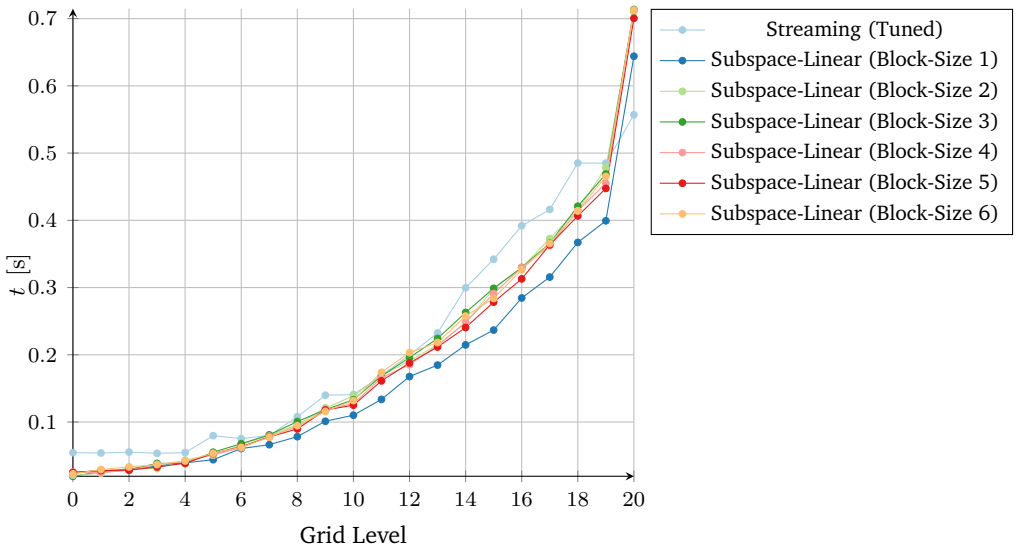

Figure 5.33: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

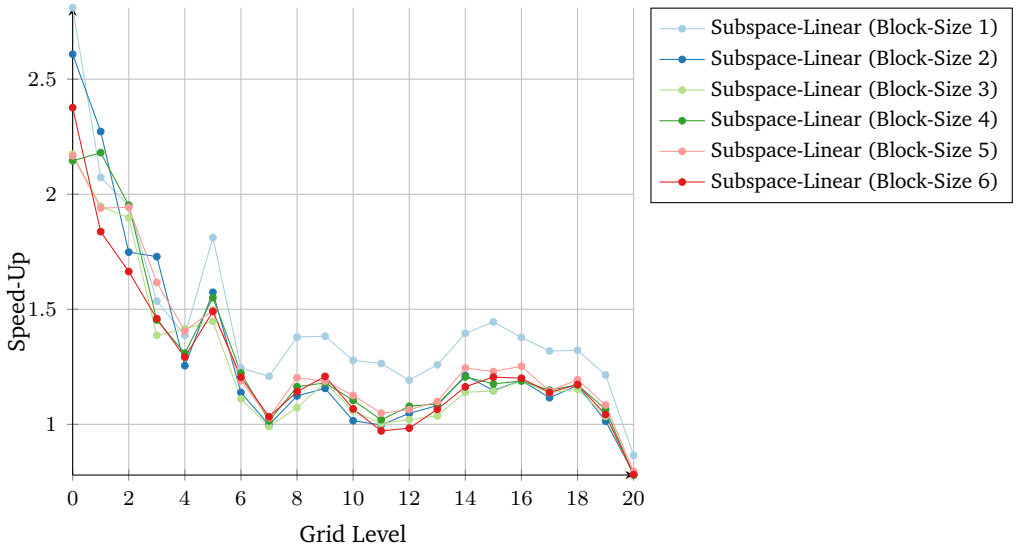

Figure 5.34: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

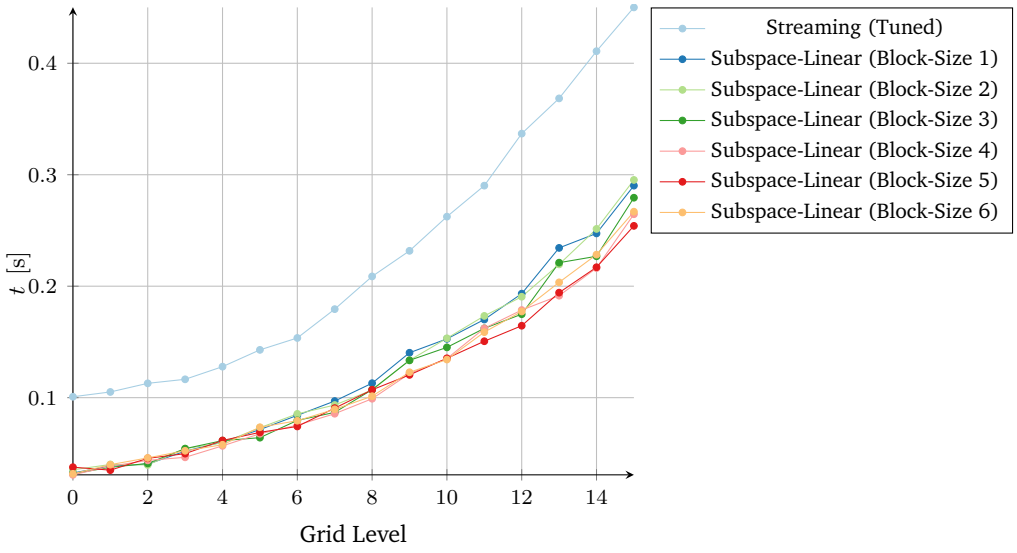

Figure 5.35: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

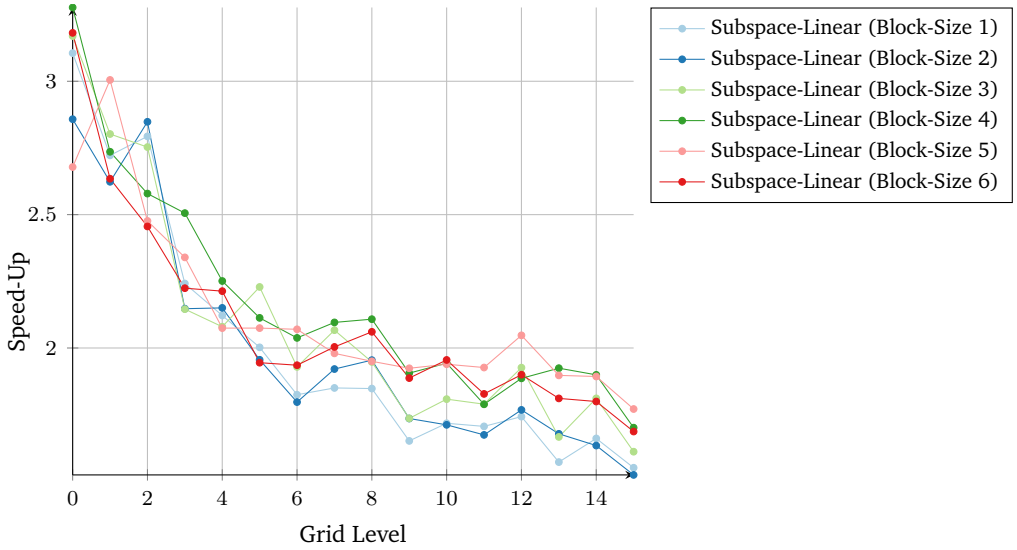

Figure 5.36: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

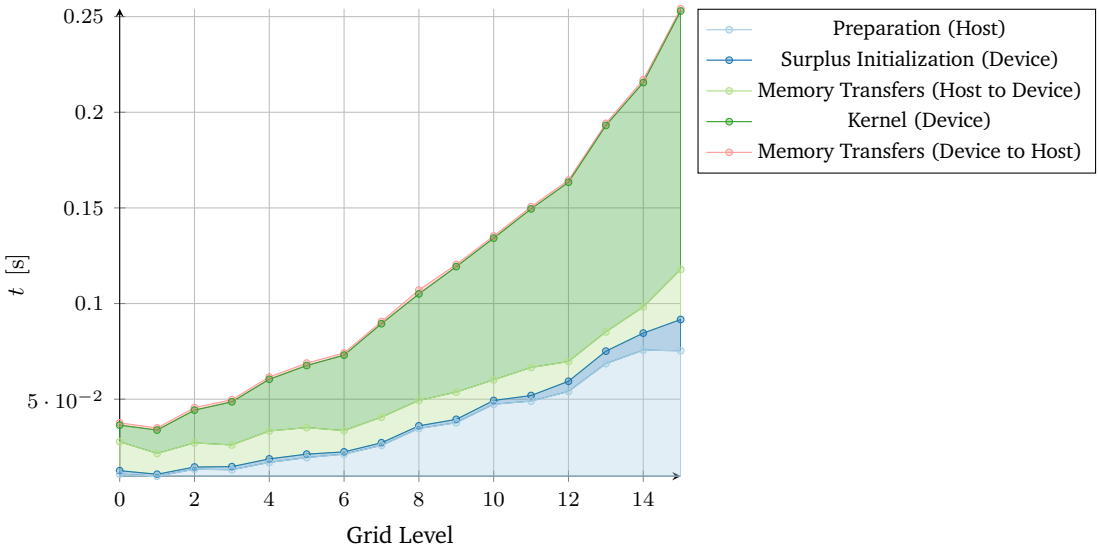

Figure 5.37: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of five) for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

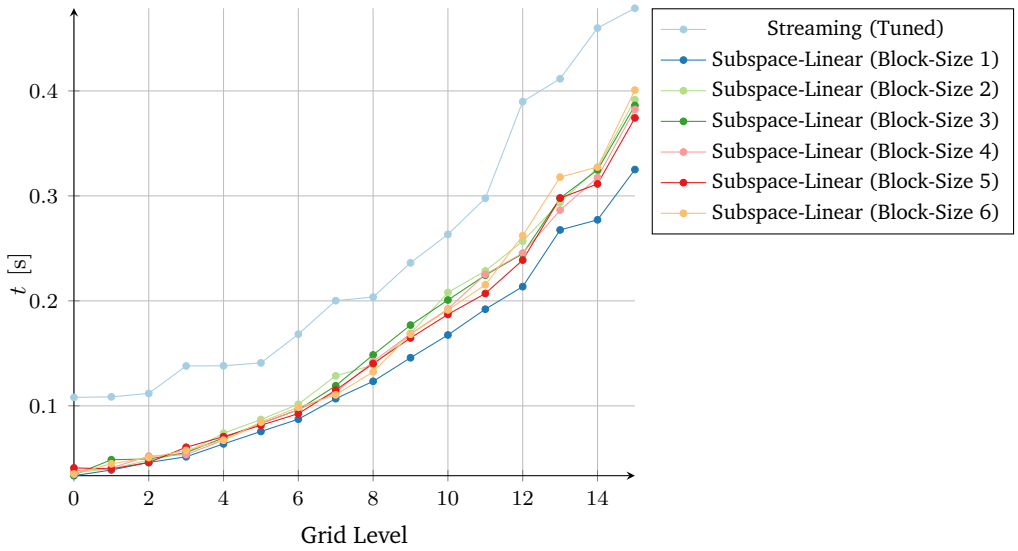

Figure 5.38: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

a negative one to a positive one for the operator $B\mathbf{v}$. Furthermore, we can observe that with larger block-sizes we are able to achieve a speed-up of more than a factor of three over the streaming algorithm, whereas with smaller block-sizes we only achieve a speed-up up to two. For all block sizes, the behavior and increase of the speed-up is similar, beginning after the first refinement steps it rises and stagnates after step 10 to 20, depending on the block-size, with smaller block-sizes stagnating earlier, which fits the earlier explanation as subspace-skipping is more important for higher refined grids.

On the Five Dimensional Gaussian Data Set with Level Five and Level Seven Base Grids

With an increased base-level to five (other parameters are again equivalent to the respective SDSS DR5 scenario), we can again see a behavior similar to grids with base-level two but clearly also an impact of the larger regular sub-structure of the grid (Figures 5.47 and 5.51) resulting in an increased speed-up (Figures 5.48 and 5.52). Note that, for this scenario, we did not encounter the earlier memory related issues, as we did not run enough refinement steps to generate the larger subgrids (see Table A.6 for details).

Increasing the base-level further to seven (other parameters still similar to the respective SDSS DR5 scenario), we can again observe similar changes (see Figures 5.54 and 5.58). A further change, however, is the reduced impact of the block-size on the operator $B^\intercal\boldsymbol{\alpha}$ (Figures 5.56 and 5.60), especially for a blocks-size larger than three, which may be caused by hightened warp divergence. Nevertheless we can still present a speed-up of almost 3.5 for $B^\intercal\boldsymbol{\alpha}$ and a speed-up ranging from 2.5 to 3 for $B\mathbf{v}$ (Figures 5.55 and 5.59).

On the Ten Dimensional Gaussian Data Set with Level Two and Level Five Base Grids

By increasing the dimensionality of the Gaussian data set to ten we can test if our observations still hold true for higher-dimensional data sets. Due to this increase, we expect a significantly higher number of subgrids to be generated by refinement and thus ultimately a decrease in performance of the subspace-linear algorithm.

Let us first begin with a base-level of two and 20 refinement-steps employing the parameters of previously used scenarios for this base-level. Following previous observations for the five-dimensional Gaussian data set, we can again see a considerable impact of the block-size, allowing us once more to beat the streaming algorithm (Figures 5.61 and 5.65). As expected, however, the speed-up on the ten-dimensional data set turns out to be less than the speed-up on the five-dimensional data set (Figures 5.62 and 5.66), for $B^\intercal\boldsymbol{\alpha}$ this speed-up increases over the refinement steps (for larger block-sizes), for $B\mathbf{v}$ it stagnates approximately after refinement step five, again most likely due to the influence of atomics. Nevertheless higher block-sizes once more tend to achieve better results for both operators and we consistently reach a speed-up higher one, specifically from approximately 1.2 to above 2.5 for the multi evaluation and approximately two for the transposed multi evaluation.

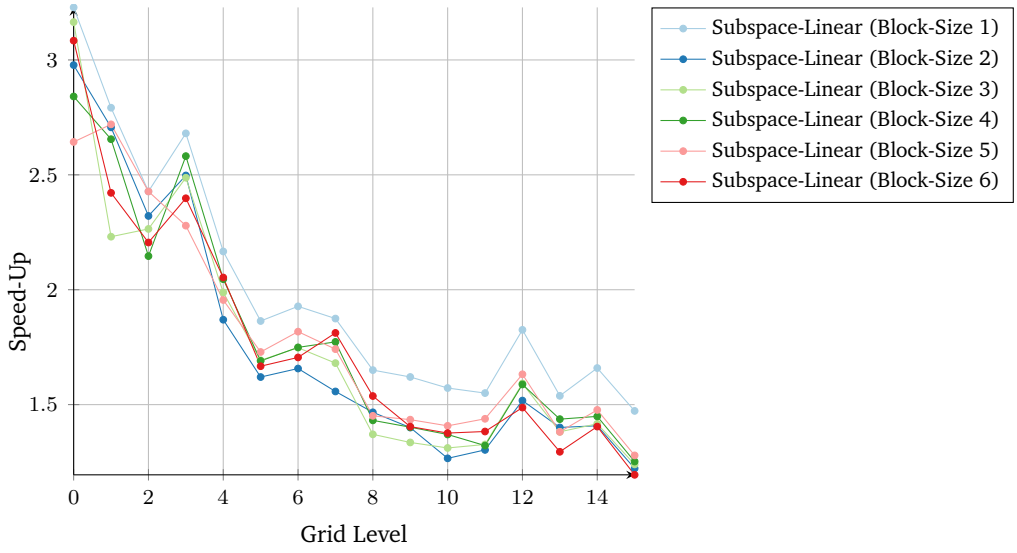

Figure 5.39: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Plotted over the refinement steps.

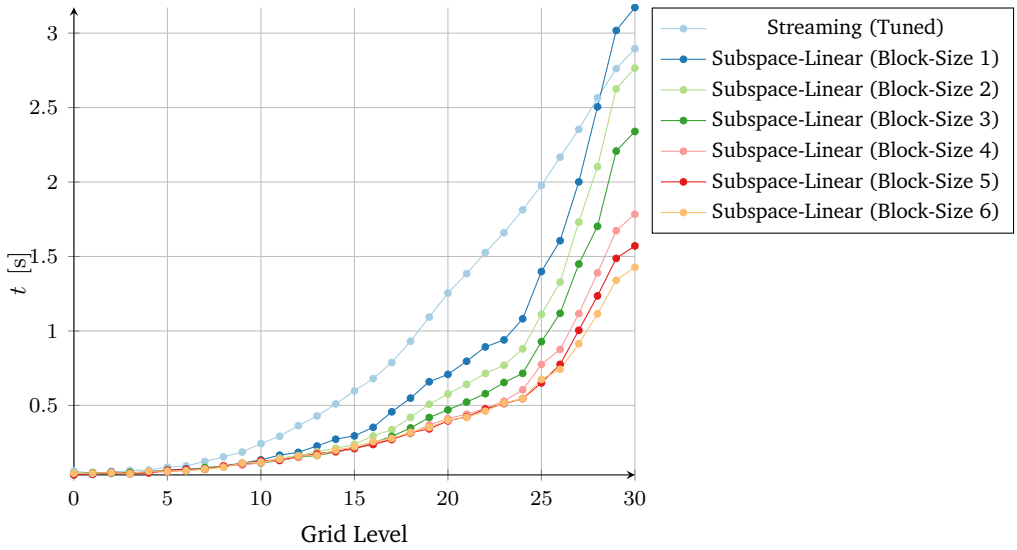

Figure 5.40: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

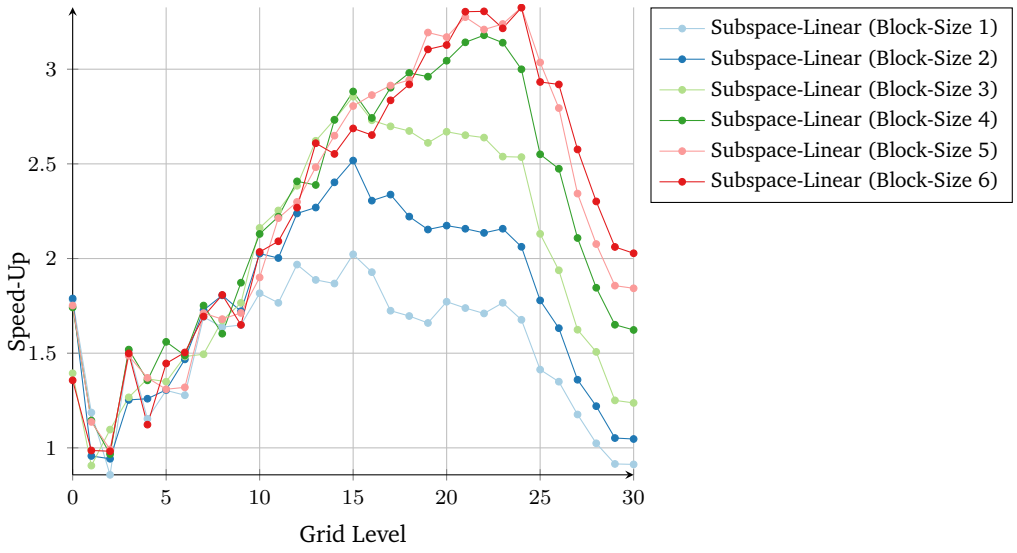

Figure 5.41: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

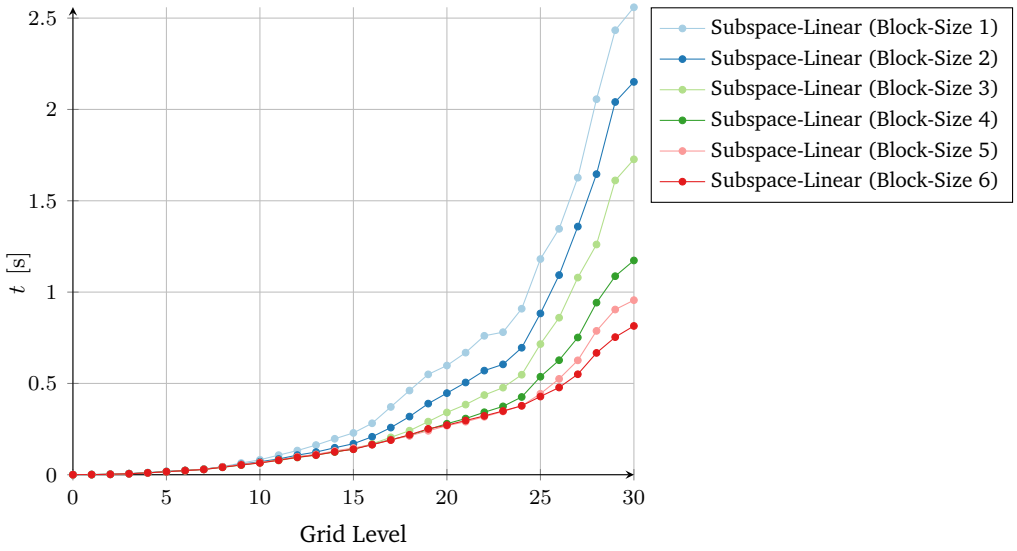

Figure 5.42: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

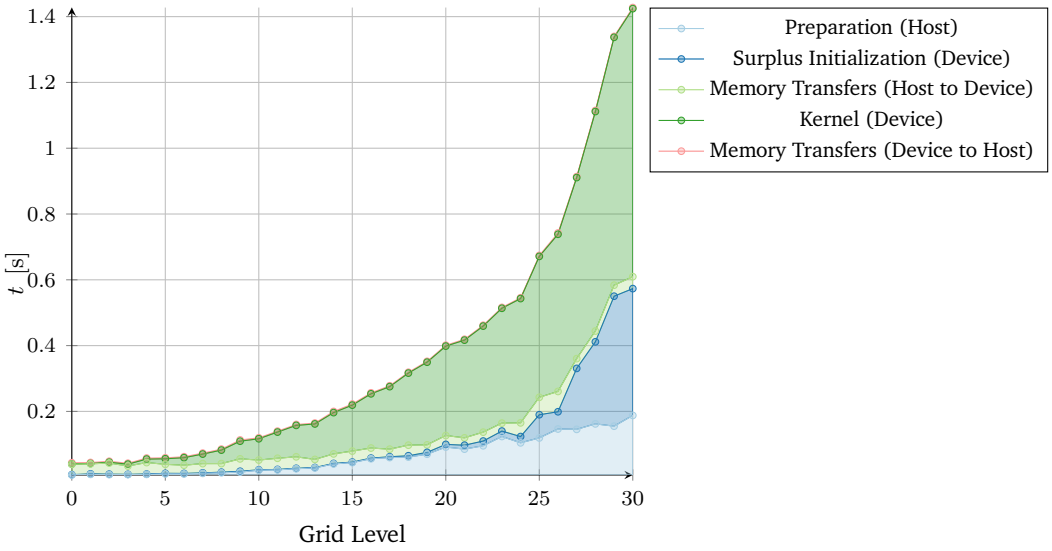

Figure 5.43: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B^\intercal \alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

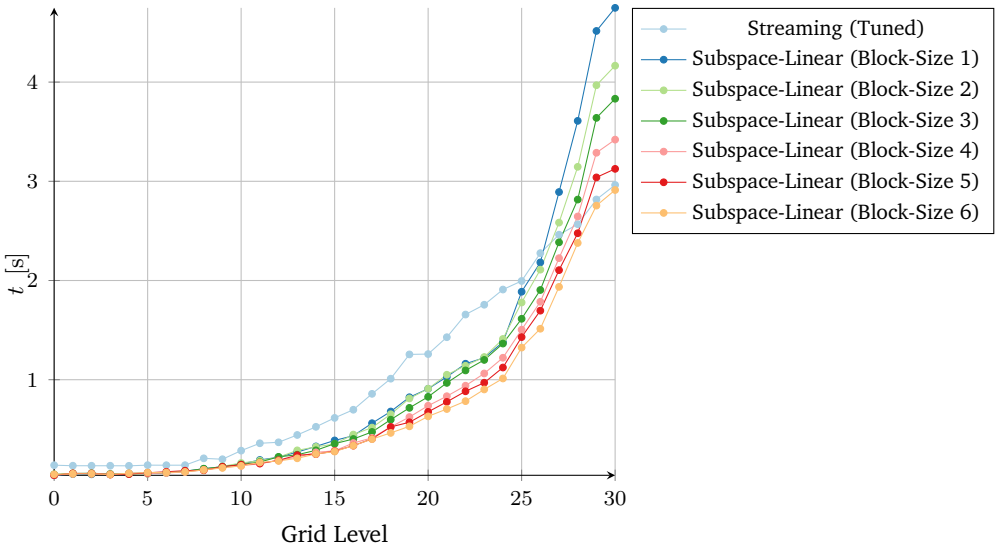

Figure 5.44: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

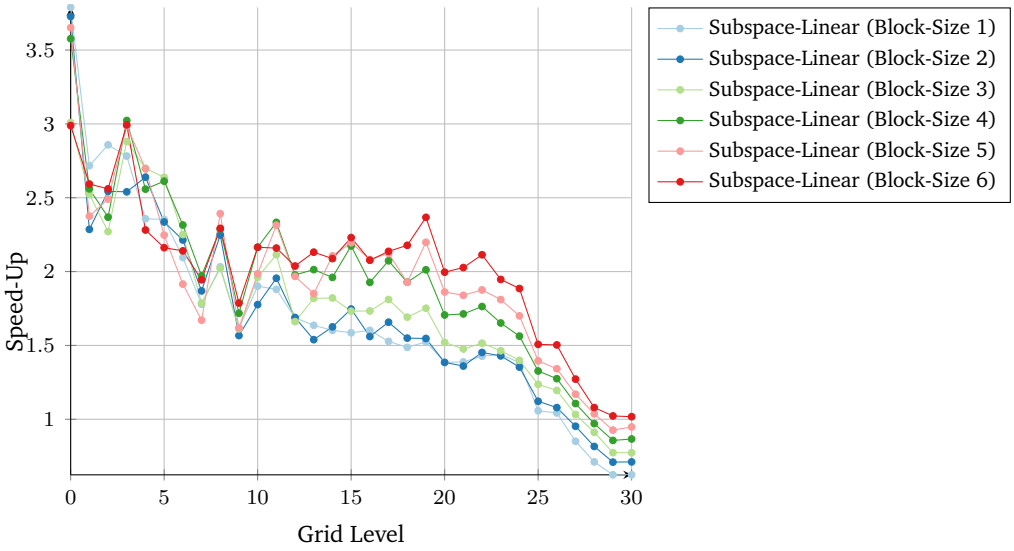

Figure 5.45: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

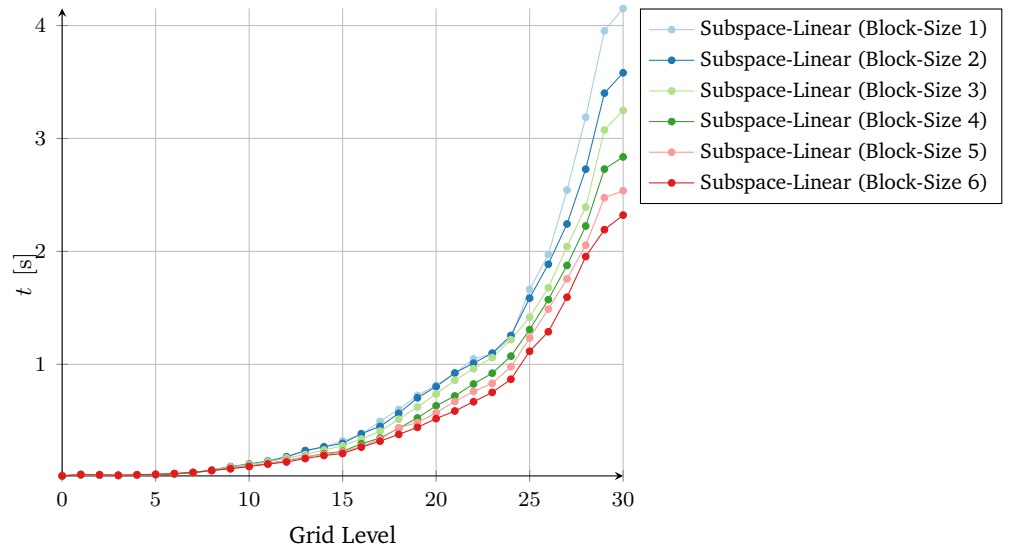

Figure 5.46: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

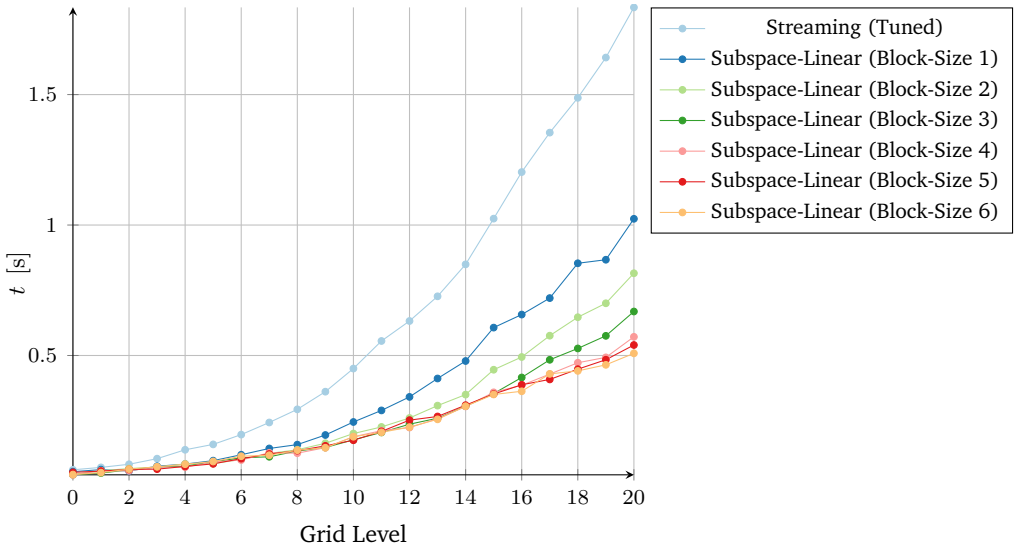

Figure 5.47: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

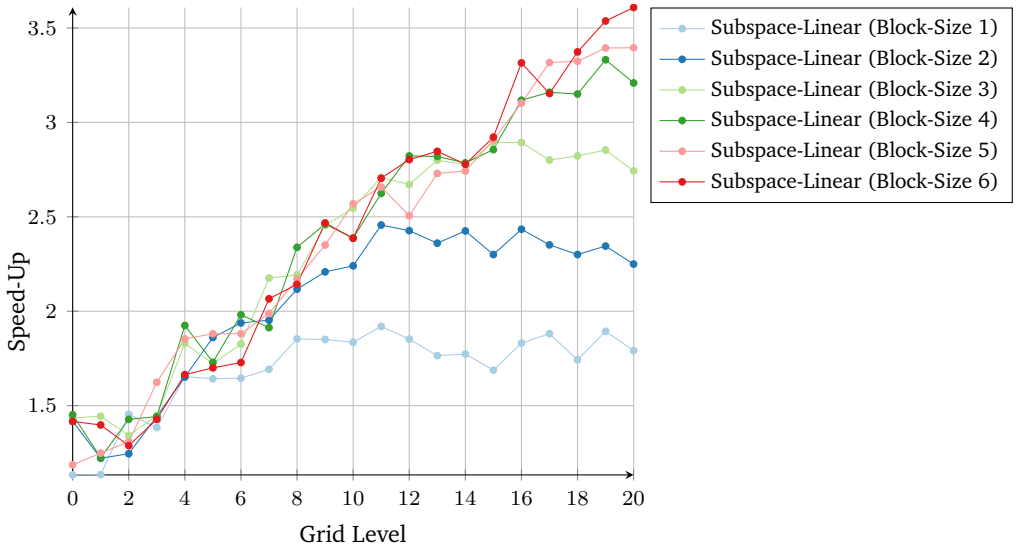

Figure 5.48: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

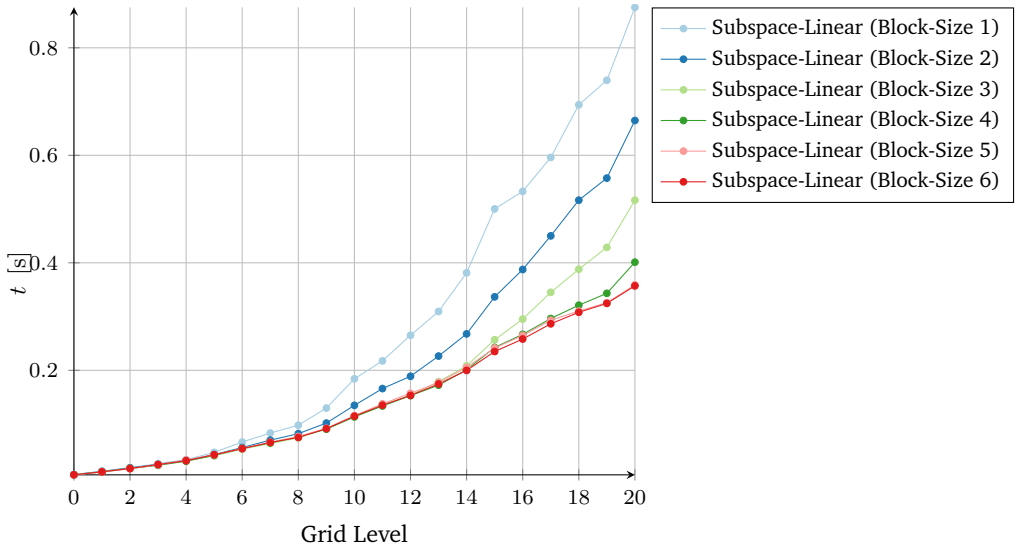

Figure 5.49: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

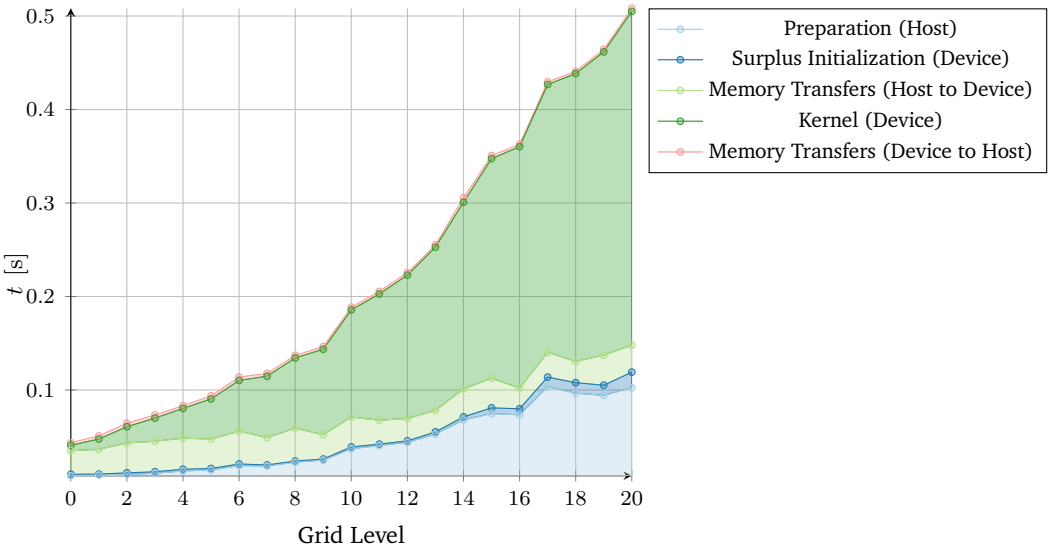

Figure 5.50: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

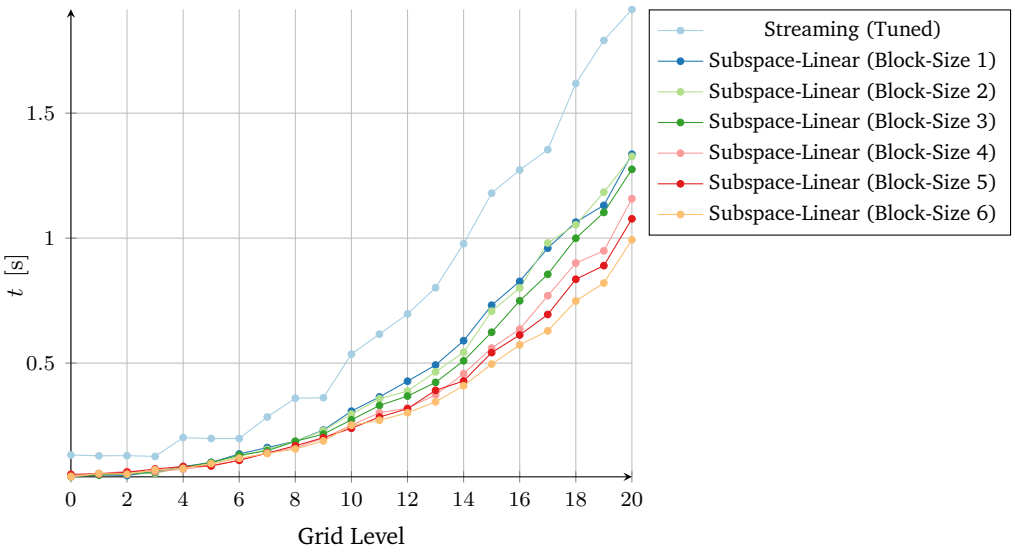

Figure 5.51: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

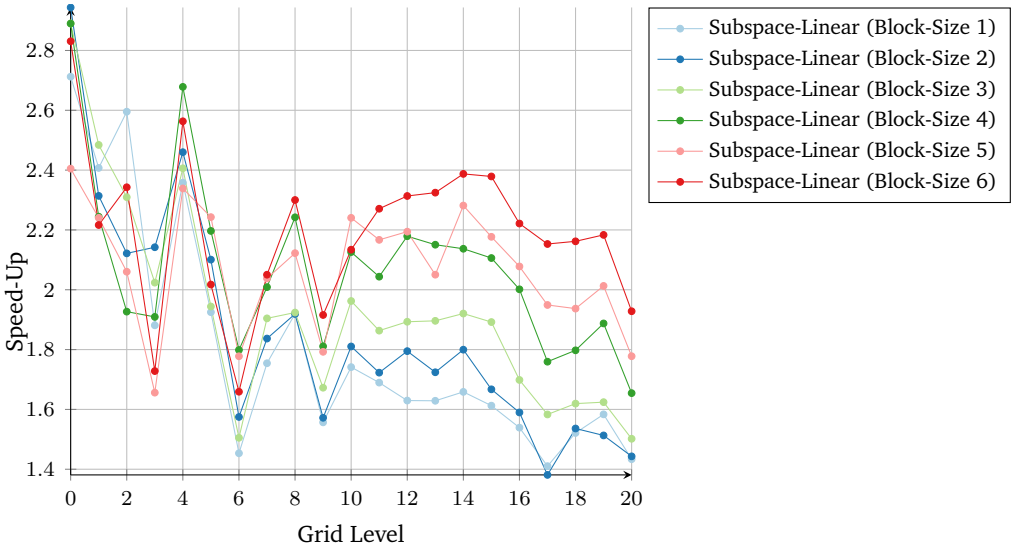

Figure 5.52: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

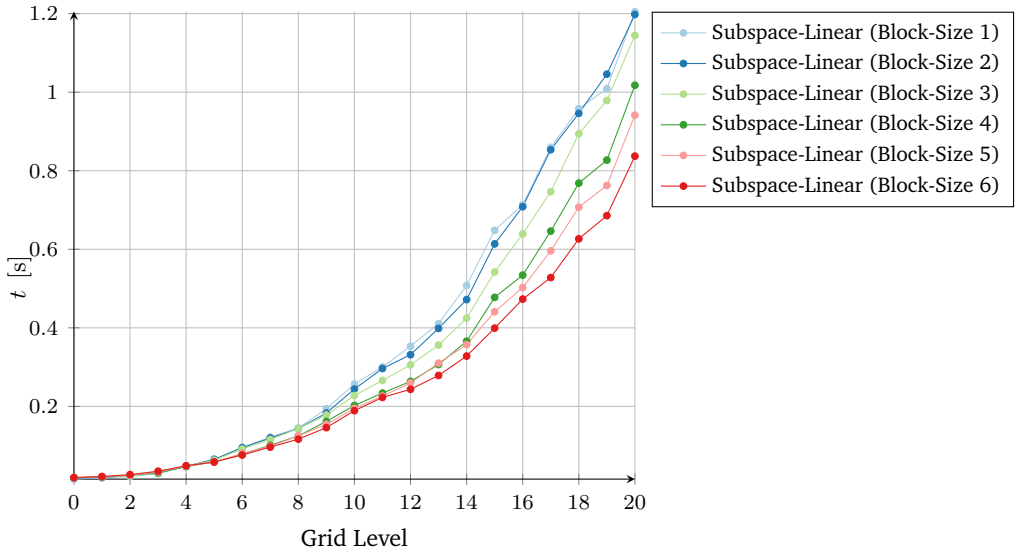

Figure 5.53: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

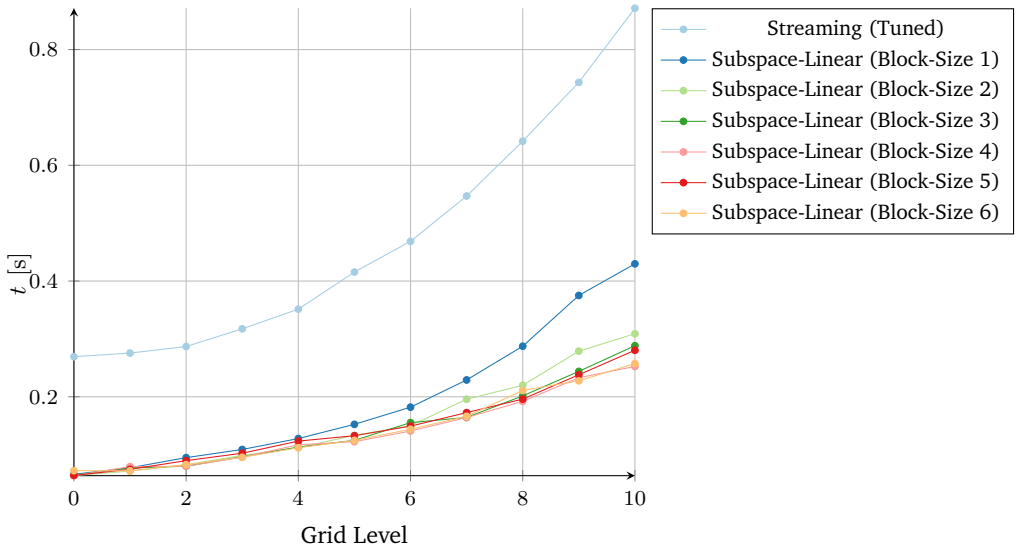

Figure 5.54: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

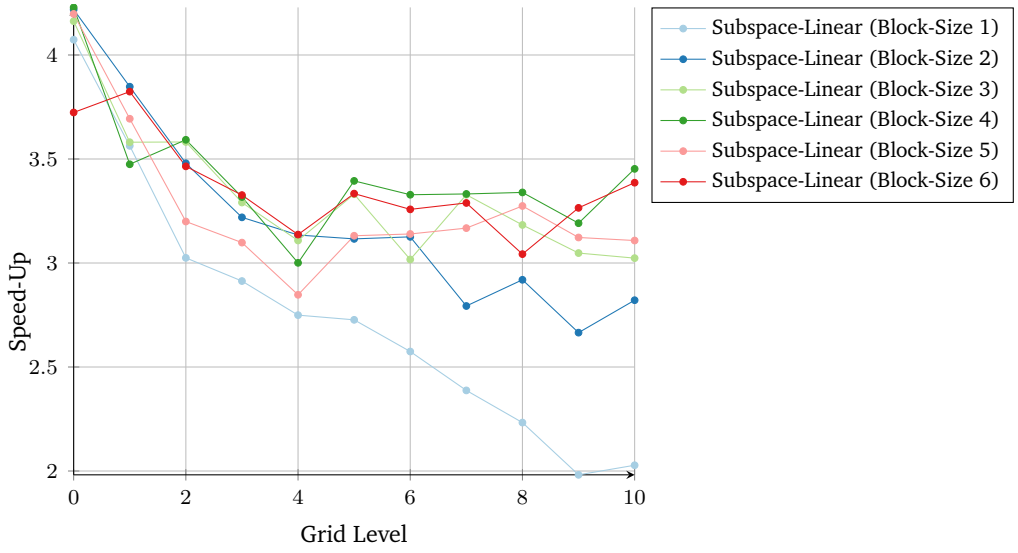

Figure 5.55: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

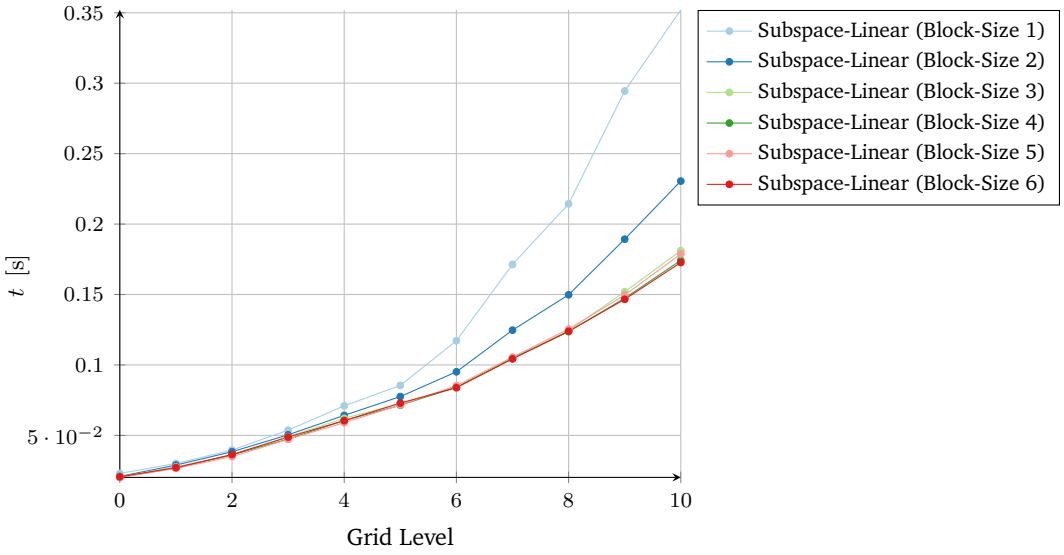

Figure 5.56: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

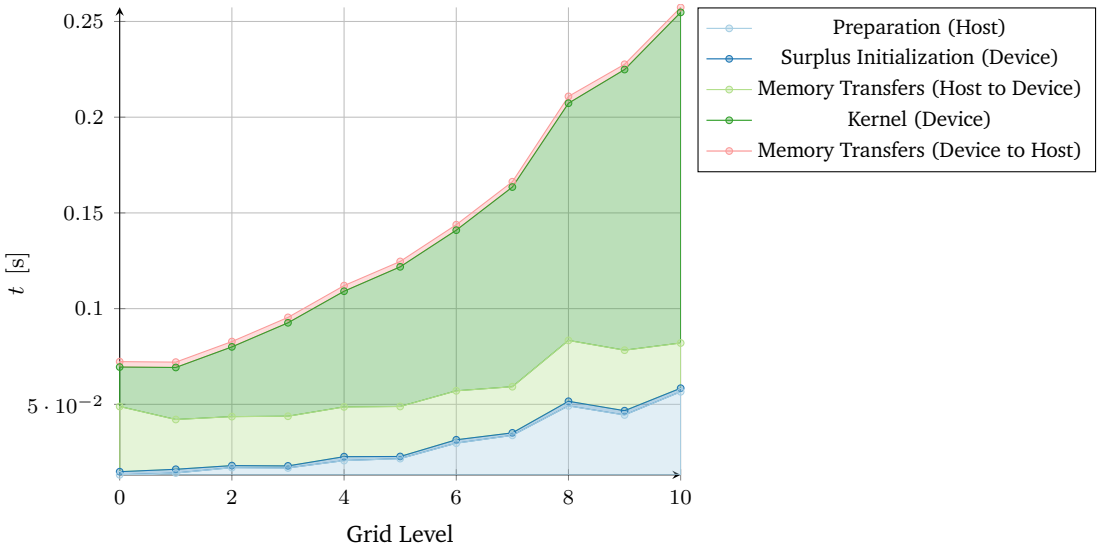

Figure 5.57: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

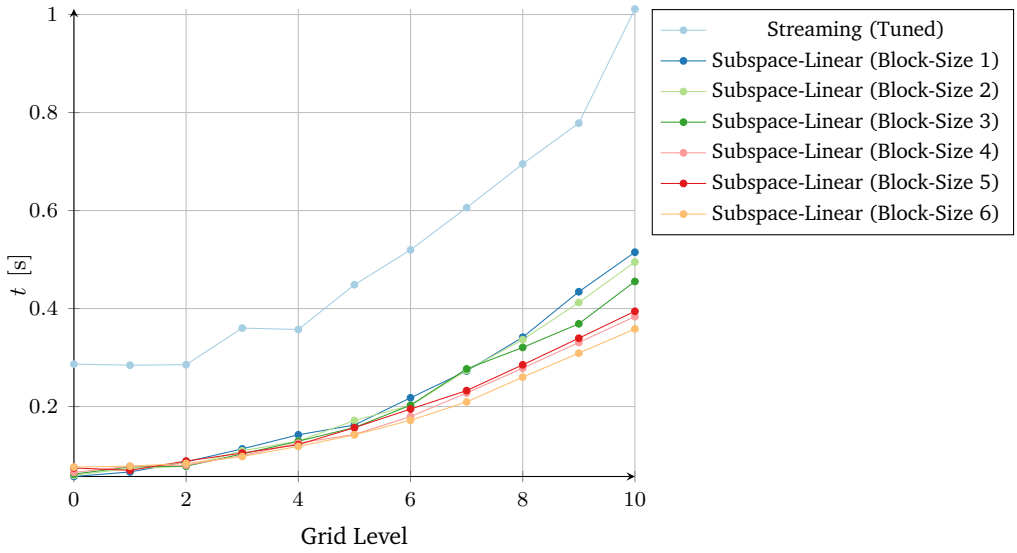

Figure 5.58: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

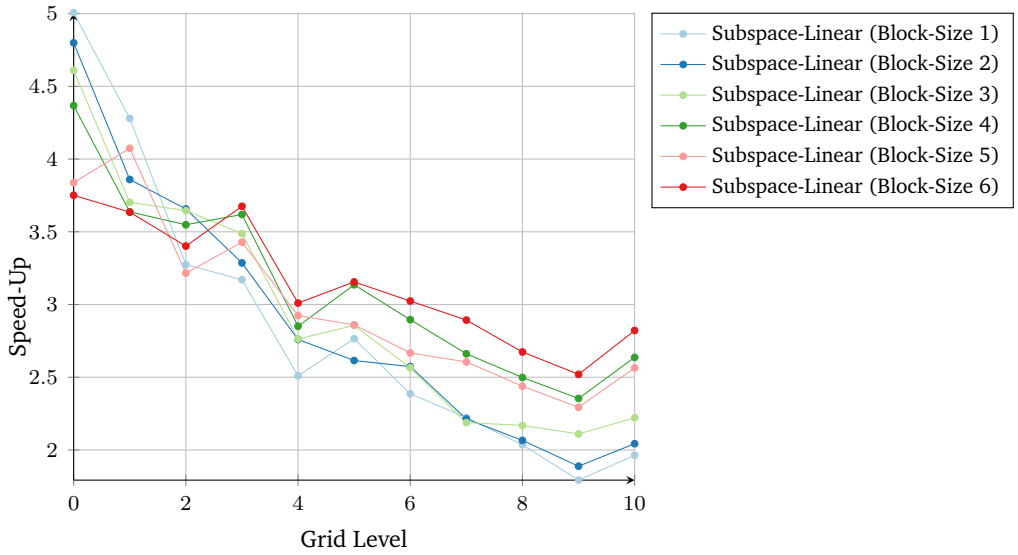

Figure 5.59: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

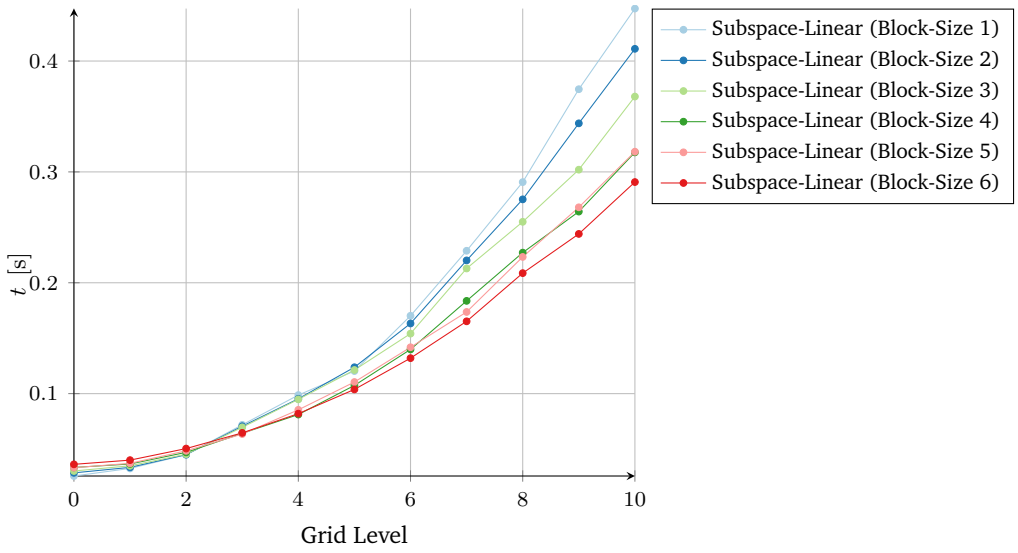

Figure 5.60: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

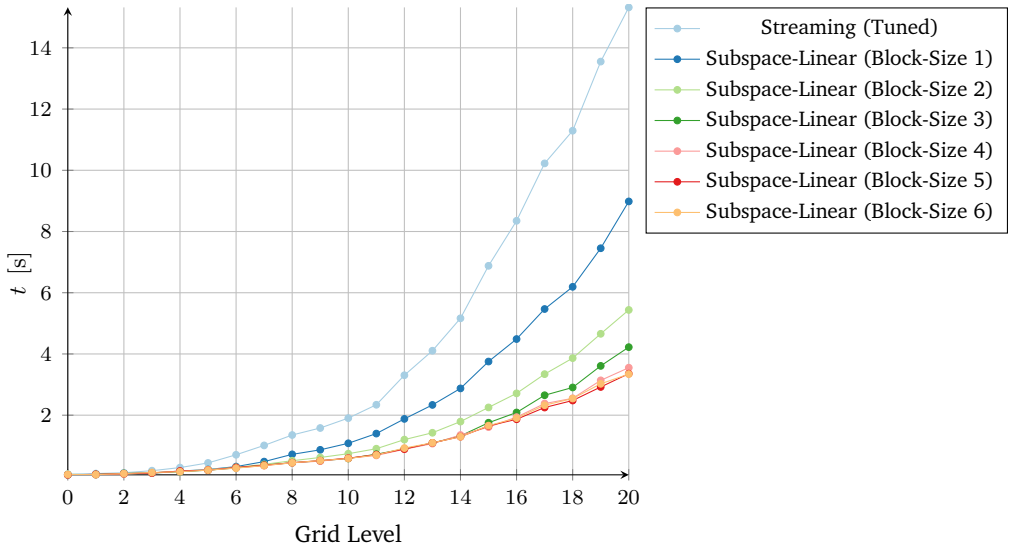

Figure 5.61: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

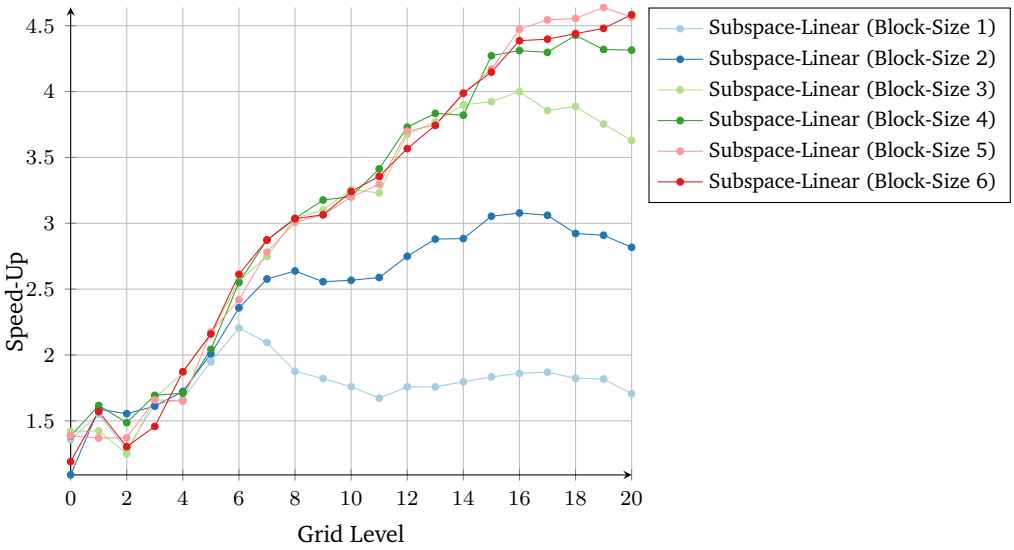

Figure 5.62: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

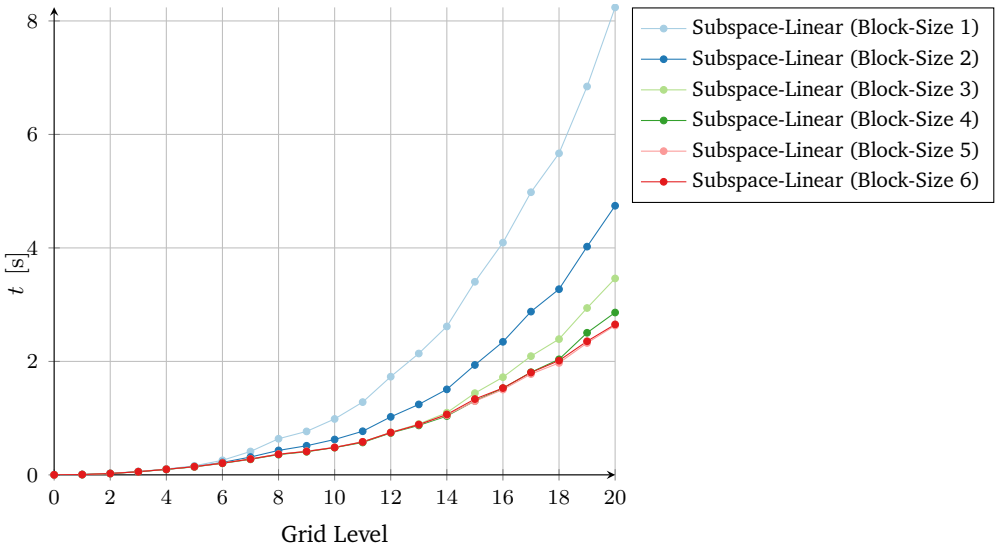

Figure 5.63: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

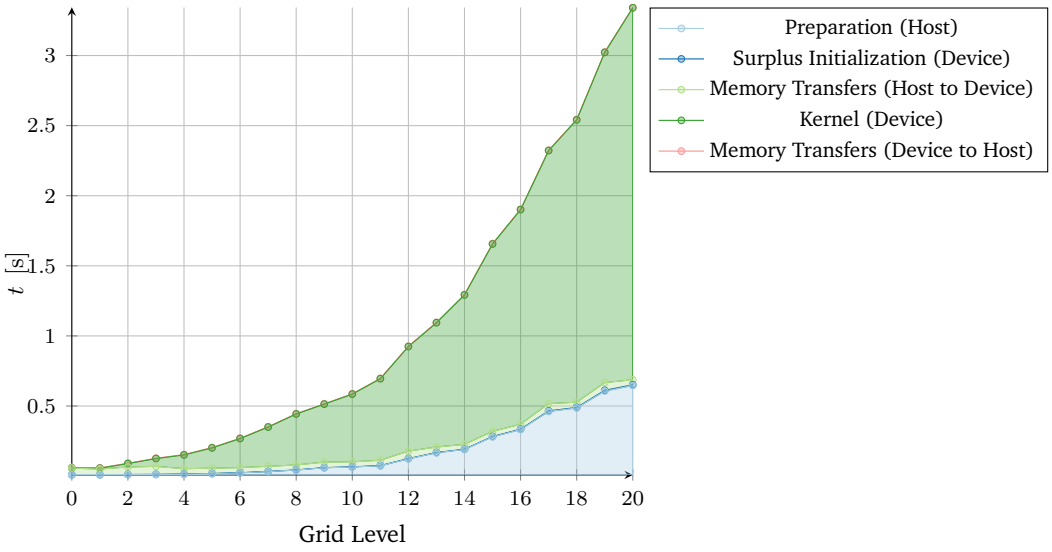

Figure 5.64: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

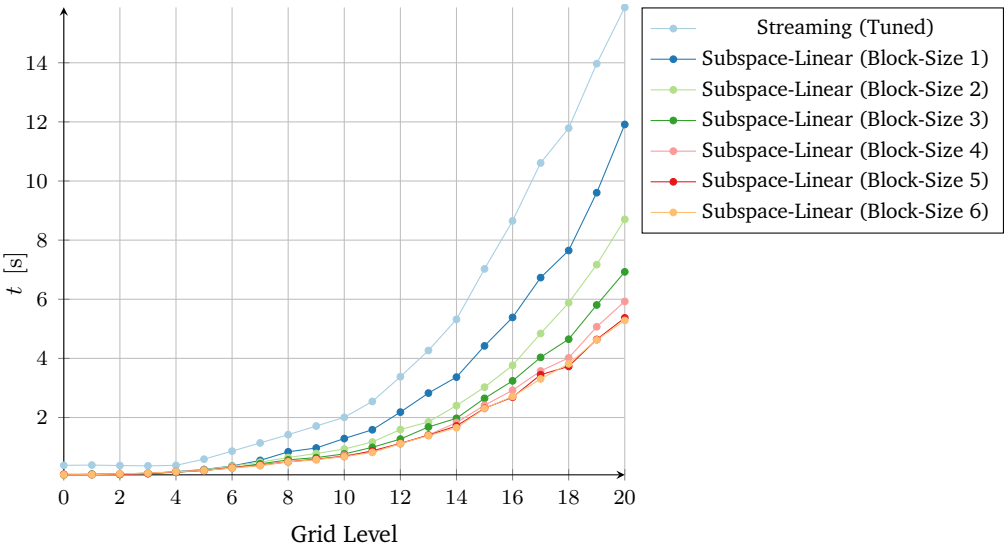

Figure 5.65: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

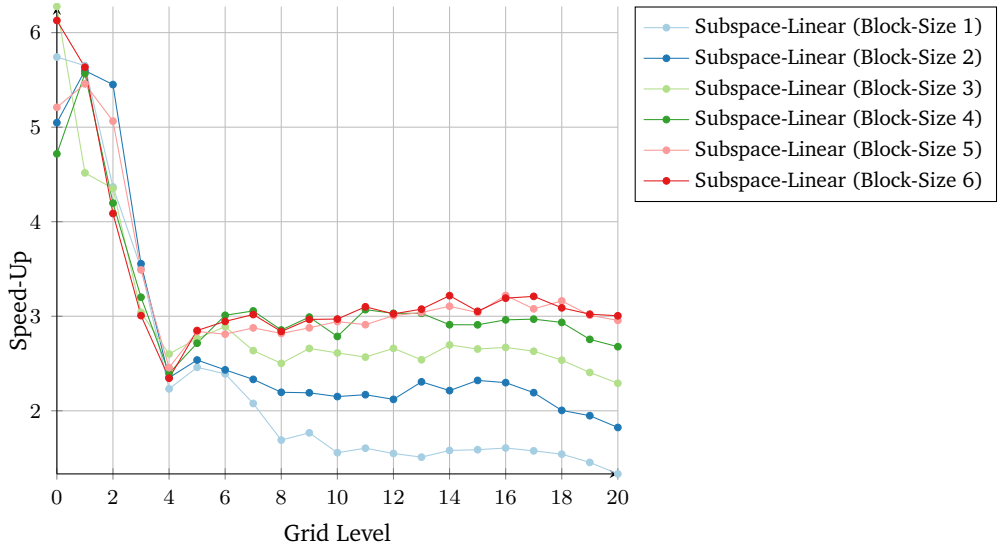

Figure 5.66: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

Increasing the base-level to five (and using the refinement-parameters from previous base-level five scenarios), we can again see an overall increase in the speed-up of the subspace-linear algorithm over the streaming algorithm, especially for smaller sized grids while it is somewhat similar on larger sized grids (Figures 5.69, 5.69, 5.73 and 5.73). This is most likely caused by the faster reduction in points per subspace, observable for the scenario with base-level seven (compare Table A.8 and Table A.9).

Due to the comparatively small maximum number of potential grid points, i.e. the number of grid points if all subgrids were full, we encounter no memory-related issues on both scenarios. Again, the speed-up can be further improved if the preparation step is executed only when the grid changes, e.g. when the algorithm is applied for refinement (Figure 5.71).

5.3.3 Conclusion and Outlook

With the results from the previous section, we can present speed-ups of the subspace-linear algorithm over the highly optimized and parameter-tuned streaming algorithm in all except the adaptive scenario for the HIGGS data set. This data set is particularly challenging for the subspace-linear algorithm due to its high dimensionality (28 dimensions), its high irregularity, and the resulting number of approximately 1.9 million subspaces. Aside from this scenario, the subspace-linear algorithm outperforms the parameter-tuned streaming algorithm by margins largely depending on the data set. By comparison of the SDSS DR5 data set with the five-dimensional Gaussian data set, we saw that the data set itself is an important factor to achieve higher speed-ups. While the structures both the data sets are vastly different, we assume that most of the changes in speed-up are related to the, at least partially, ordered structure of the Gaussian data sets, due to which points adjacent in the feature space are likely to be stored close to each other in the data set. A concrete proof of this assumption would require further investigation, such as testing against a shuffeled version of the Gaussian data set, however, we think that it should at least be considered to sort the data set beforehand according to a space-filling Z-order curve for moderate-dimensional problems (as this has only to be done once for the data set, assuming it does not change). For higher-dimensional problems, the impact of such a sorting step is likely to be reduced, as more dimensions inherently lead to more subspaces for the same number of data points.

All evaluations presented in this chapter include the preparation steps in their results. This means that the speed-ups presented could be, depending on the data set and scenario used, further improved, as this step is substantially more time consuming for the subspace-linear algorithm than for the streaming algorithm and it only has to be executed after the grid has changed. Note, however, that the run-time of these preparation steps may be significantly smaller on frameworks other than SG++, as they depend on the representation of the grid. For such frameworks, the streaming algorithm may have a longer preparation time.

A somewhat surprising observation we made regarding NVIDIA's Pascal architecture is, that the performance of 64 bit atomic operations improved significantly in comparison to previous architectures, due to NVIDIA's re-design of those [NVI16]. This allows for an

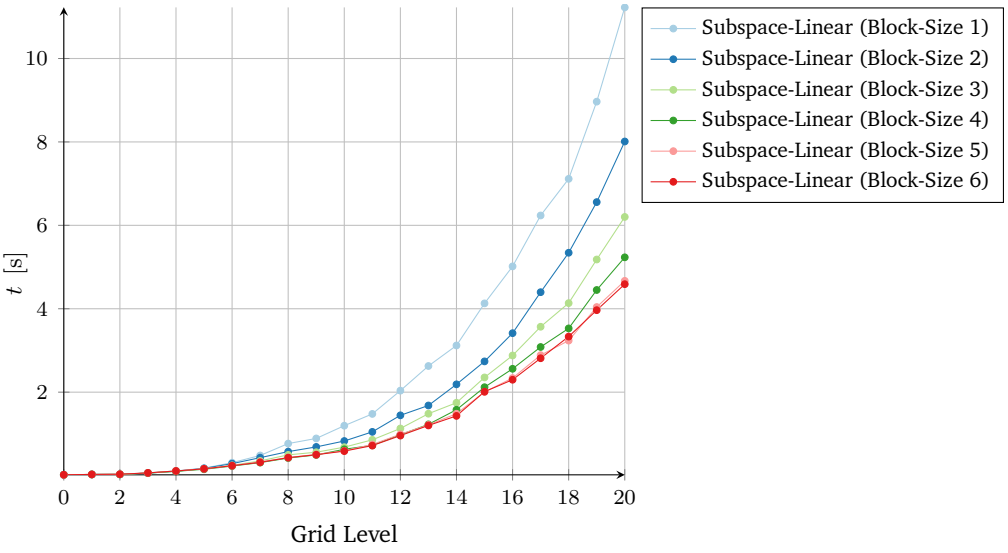

Figure 5.67: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

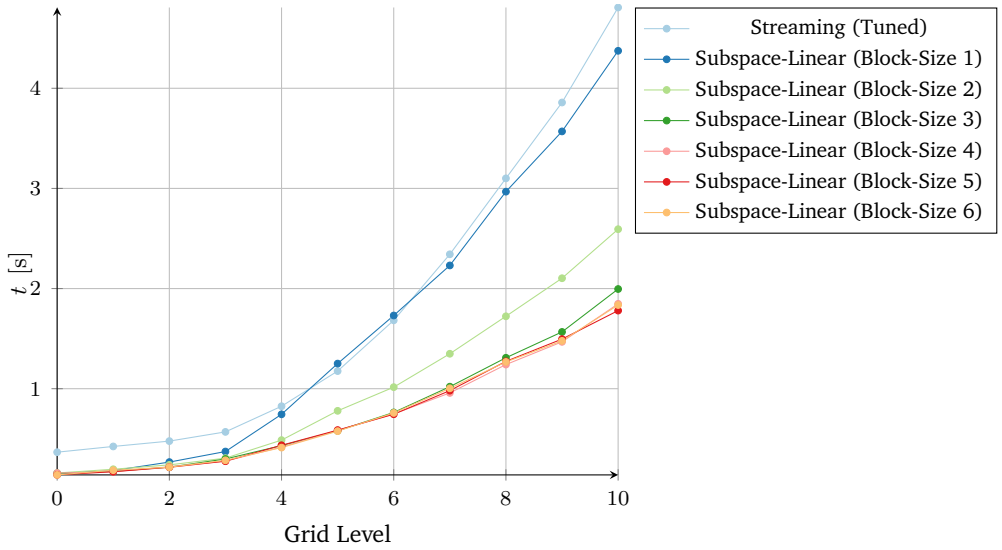

Figure 5.68: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

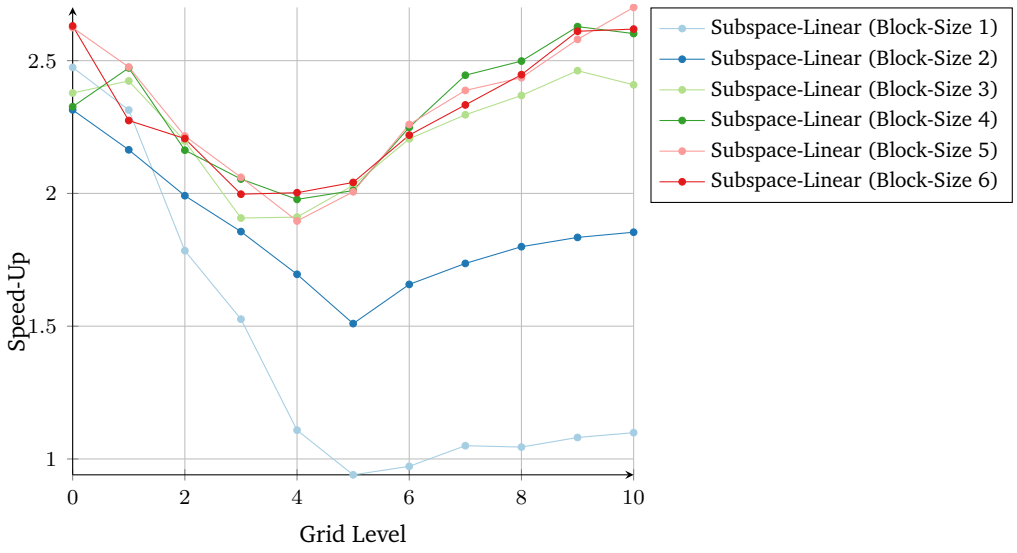

Figure 5.69: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

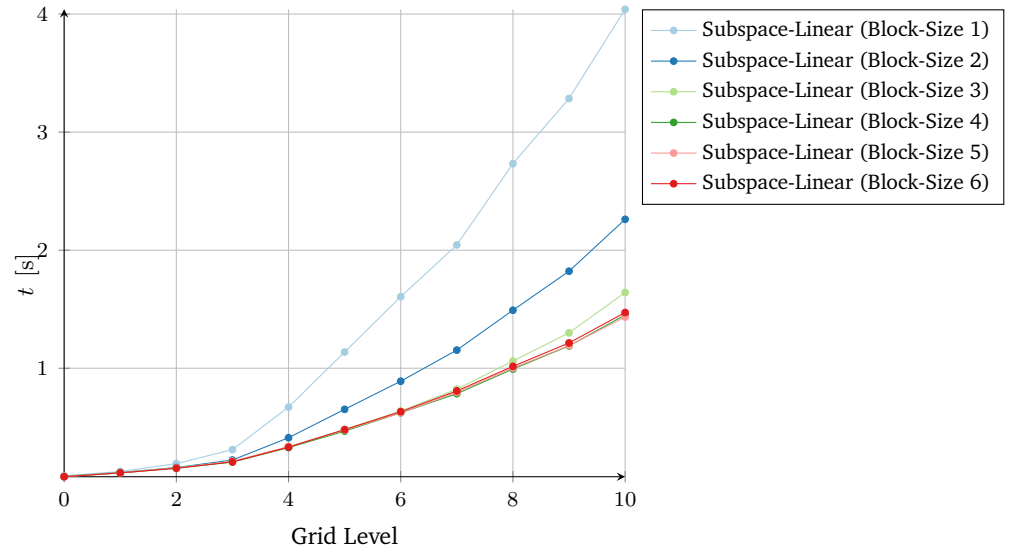

Figure 5.70: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

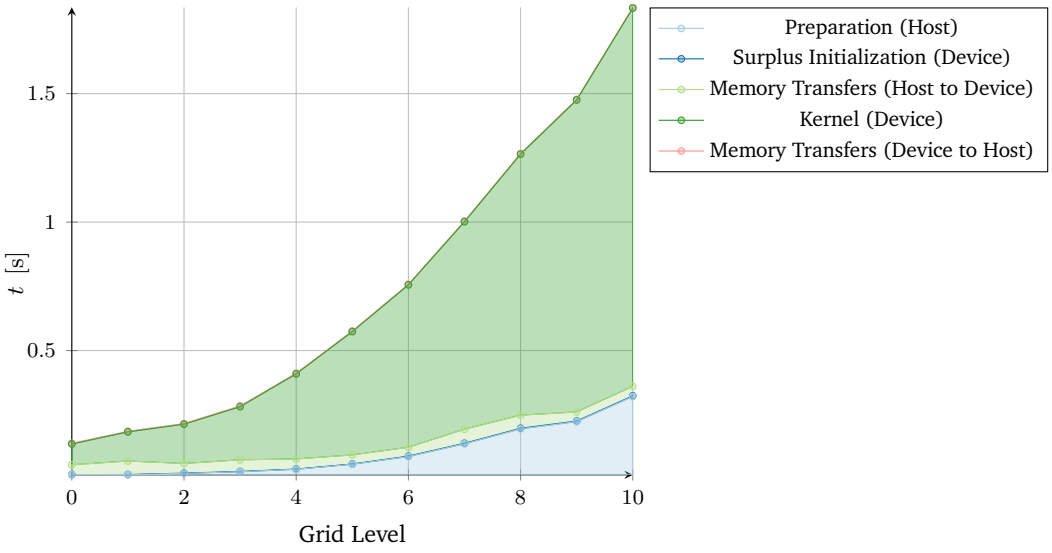

Figure 5.71: Run-time composition of one complete execution of the subspace-linear algorithm (with block-size of six) for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

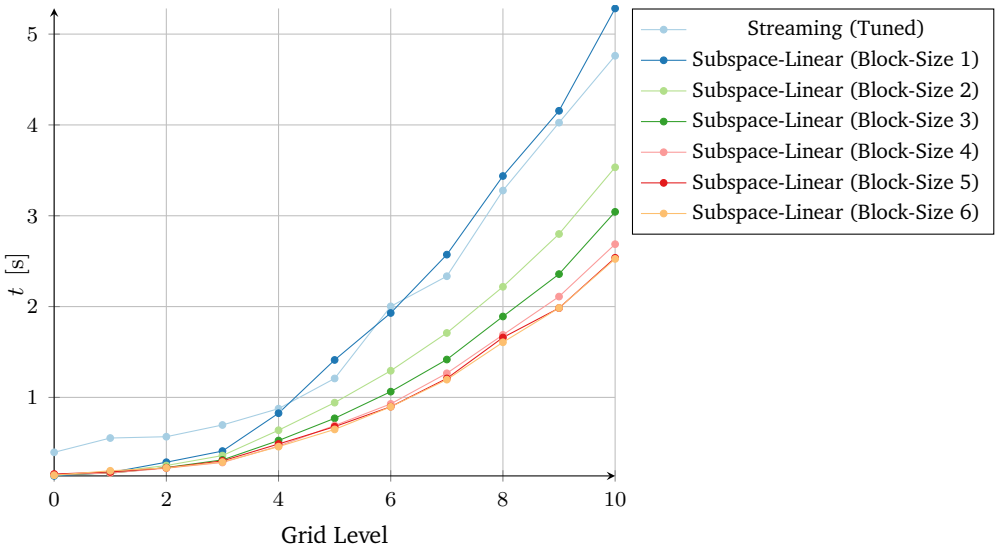

Figure 5.72: Absolute duration of streaming and subspace-linear algorithms (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

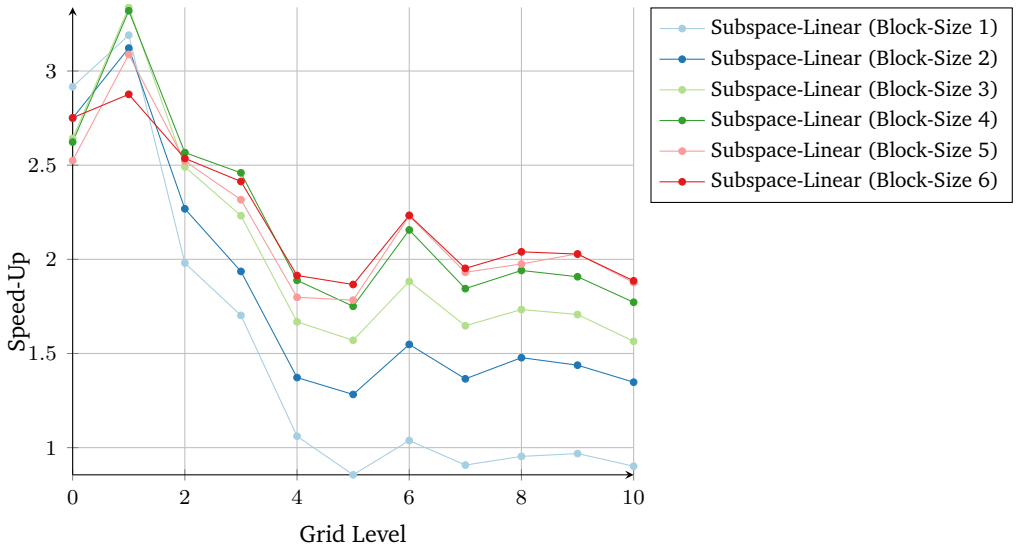

Figure 5.73: Speed-up of the subspace-linear algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

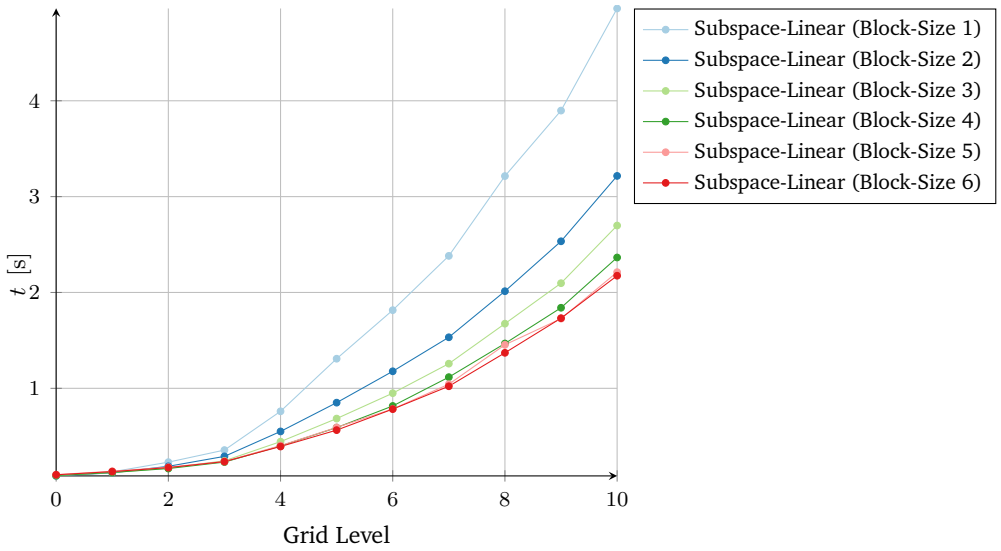

Figure 5.74: Kernel duration of the subspace-linear algorithm (with multiple block-size values) for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Plotted over the refinement steps.

efficient implementation of the operator $B\mathbf{v}$ with the subspace-linear algorithm on the Pascal architecture. While the use of atomics still has a significant impact, as we could see in the different behavior of the operators $B^\intercal\boldsymbol{\alpha}$ and $B\mathbf{v}$ even though their implementation is, apart from the atomic operation, largely identical, we were not required to make any algorithmic changes or specific optimizations to be competitive with the streaming algorithm. Regarding the subject of optimizations, we only employed basic loop unrolling via a recursive C++ template and introduced the processing of multiple data point evaluations in a single thread, thus, further optimization of our implementation may be possible. As we have seen, the number of evaluations processed in a single thread has a considerable impact on the run-time of the algorithm, especially for partially ordered data sets. For such data sets, we may be able to further improve the run-time by abandoning the concept of evaluating multiple data points per thread and instead evaluate a single subspace per thread-block, reducing the required surplus and subspace-node load operations to one per thread-block. Subspace-skipping could then be done, for example, with CUDA's warp vote functions, however, further evaluation of their performance would be required to give a definitive answer for this.

A severe limitation of the subspace-linear algorithm, introducing performance penalties for highly refined grids, is its memory requirement. We could observe this problem on several scenarios, even preventing us from executing the algorithm on some of the involved grids. Note that this limit would be especially severe if we would choose 32 bit integers instead of 64 bit integers to address the surplus values. It may be possible to circumvent the issue of execution by use of NVIDIA's unified memory capabilities [NVI16; NVI17], allowing the GPU to access CPU-memory by use of paging, however, we aim to provide a different solution for this problem in the next chapter.

5.4 The Subspace-Adaptive Algorithm

For comparisons, we use the parameter-tuned and highly optimized OpenCL streaming algorithm provided by SG++, as well as the CUDA implementation of the subspace-linear algorithm described in-depth in the previous section. Note that both of these algorithms also form the base of our implementation of the subspace-linear algorithm. In all following evaluations, we will only show results where both algorithms have been executed at least for some grids in the scenario, except for the subspace-adaptive algorithm with thresholds $p = 0$ and $u = 0$, which executes only the subspace-linear algorithm and is included as a reference for it.

5.4.1 Spatially Adaptive Scenarios

As it makes little sense to apply the streaming algorithm on completely regular sparse grids, see the results of the previous section, we exclusively evaluate the subspace-adaptive algorithm on spatially adaptive sparse grid scenarios.

On the SDSS DR5 Data Set with a Level Two Base Grid

We begin again with the SDSS DR5 data set, a base grid of level two, 30 surplus-based refinement steps each refining 80 points refined per step with a maximum of 120 CG solver iterations, and a regularization factor of 1×10^{-5}.

If we look at the absolute duration of the subspace-adaptive algorithm for the operator $B^{\intercal}\boldsymbol{\alpha}$ in comparison to its two competitors Figure 5.75, the subspace-linear and the streaming algorithm, we can see that it, depending on the chosen threshold values and refinement steps, performs similar or somewhat worse than the streaming algorithm and notably worse than the subspace-linear algorithm, with the exception of the higher refinement steps where the subspace-linear algorithm runs into memory-related problems (described in Section 5.3.2). This comparison, however, includes the preparation steps required for each algorithm. Due to our implementation of the subspace-adaptive algorithm as modular solution which allows us to switch out the implementation of the sub-algorithms (streaming and subspace-linear), an additional preparation step is required. This preparation step has again only to be executed once the grid changes and is the main reason for the different execution times of the subspace-linear algorithm and the subspace-adaptive algorithm with threshold-values of $p = 0$ and $u = 0$ (compare Figure 5.75). Note that the subspace-adaptive algorithm with this threshold will always select the subspace-linear algorithm for all subgrids.

Excluding all preparation steps (of the subspace-linear and subspace-adaptive algorithms) yields a significant change in run-time (Figure 5.76). Now the subspace-adaptive algorithm with $p = 0$ and $u = 0$ performs similar to the subspace-linear algorithm, as it should (with the notable exception of refinement steps 27 and 28 which still show some impact of the excessive memory use, however, inexplicably to us, this seems to be reduced). We are, depending on the selected threshold values, able to lessen or remove the impact of the memory-related issues encountered on the subspace-linear algorithm (discussed in Section 5.3.2), leading to speed-up over this algorithm for the refinement-steps 27 and higher. For the operator $B\mathbf{v}$, we can observe a similar behavior when taking into account the behavior of this operator for the streaming and subspace-linear algorithms (Figure 5.79).

While we tried to tune the threshold values according to the properties of the grids and subgrids involved in this scenario (see Appendix A Table A.1 for details) and evaluated different combinations, we can see that their selection is not perfect. Ideally, we would want a run-time better than both, the subspace-linear algorithm and the streaming algorithm, as well as combination of these that does not require as much memory as the subspace-linear algorithm alone, thus avoiding memory-related problems. Especially in Figure 5.77, we can observe that the grid point-based selector brings, at least in this scenario, no benefit at all. The overhead introduced by running the streaming kernel in succession to the subspace-linear one seems to be larger than the overhead of the subspace-linear algorithm over the streaming algorithm for the subgrids with only a few number of points. However, we can also observe that the point-base selector has a significant influence for the higher refinement steps, especially as the streaming-algorithm is required on these steps due to the high memory-consumption of the subspace-linear algorithm. We originally designed the utilization threshold to handle the problems encountered in these steps, however our

threshold-values for this step have proven to be either too aggressive (in case of $u = 2 \times 10^{-7}$ and $u = 5 \times 10^{-7}$), thus not selecting enough large subgrids to be executed with the streaming algorithm and therefore not resolving memory problems, or too lax (in case of $u = 1 \times 10^{-7}$), leading to performance-penalties by selecting too many subgrids to be executed with the streaming algorithm (Figure 5.76). Based on these observations, we can conclude that a relative measure, such as utilization may not be the best choice, as it is easily influenced by the number of used points on the subgrid. A probably better solution would be the separation of this threshold into two thresholds, checking the maximum possible size of the subgrid and if this above a chosen value schedule it to be executed with the streaming algorithm if it does not have at least a larger amount of points, determined by the second threshold value.

By switching our point-based threshold value from $p = 0$ to $p = 4$ after refinement step 26 (or using a new selector as proposed above), we can present a speed-up over the streaming algorithm on all grids, with execution times roughly equivalent to the subspace-linear algorithm, and even undercutting these in refinement steps 27 and 28 (Figures 5.78 and 5.80). Note that with the appropriate threshold values, we are able to continue execution on refinement steps 29 and 30, where the subspace-linear algorithm could not be executed at all due to its excessive memory requirements.

On the SDSS DR5 Data Set with a Level Five and Seven Base Grid

Using a regular sparse grid with level five as base for the refinement and slightly changing the parameters (see Table 5.4 for details) on the same data set leads to similar results, especially when taking our observations previously made on the subspace-linear algorithm into account (compare Figures 5.81 to 5.84 and Figures 5.75 to 5.80 as well as Section 5.3.2). We encounter the same problems with the utilization-based threshold selector, due to which we, in this case, are not able to out-perform the subspace-linear algorithm, even on the higher refinement steps.

Increasing the base-level further to seven follows this pattern (Figures 5.85 to 5.88). We are again unable to beat the subspace-linear algorithm, as it has no severe memory-related performance penalties on this scenario.

On the Five-Dimensional Gaussian Data Set

For the five-dimensional Gaussian data set we reach conclusions equivalent to the ones we already reached on the SDSS DR5 data set, with respect to the changes in performance of the subspace-linear algorithm already discussed in Section 5.3.2: We can overcome the memory related issues of the subspace-linear algorithm, are able to beat the streaming algorithm on all involved grids, but are unable to beat the subspace-linear algorithm when it does not require an excessive amount of memory (Figures 5.89 to 5.104). An interesting observation we can make for the scenario with base-level two, is the different impact of the point-threshold in higher refinement steps for the transposed multi-evaluation $B\mathbf{v}$.

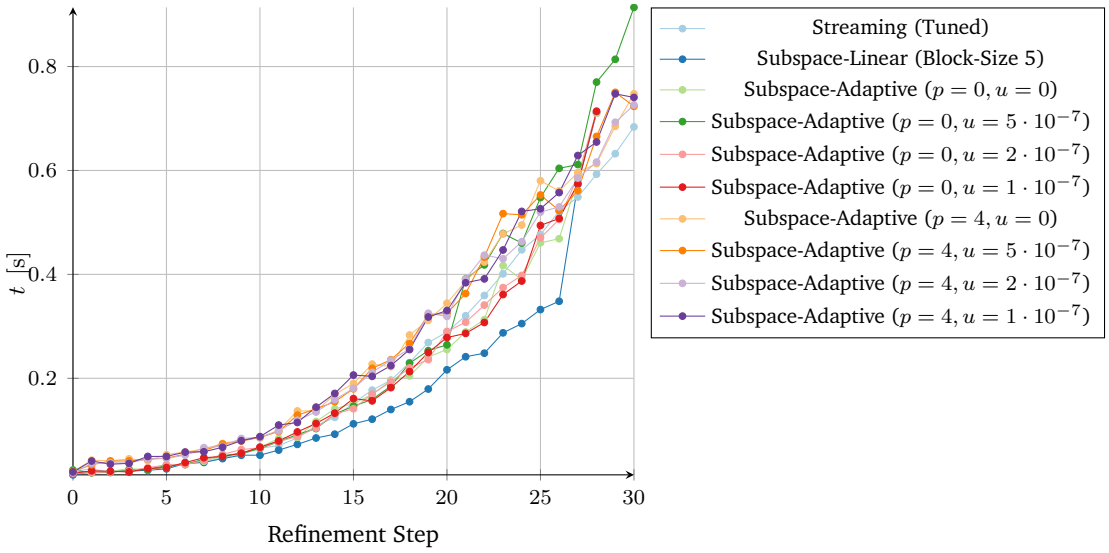

Figure 5.75: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of five.

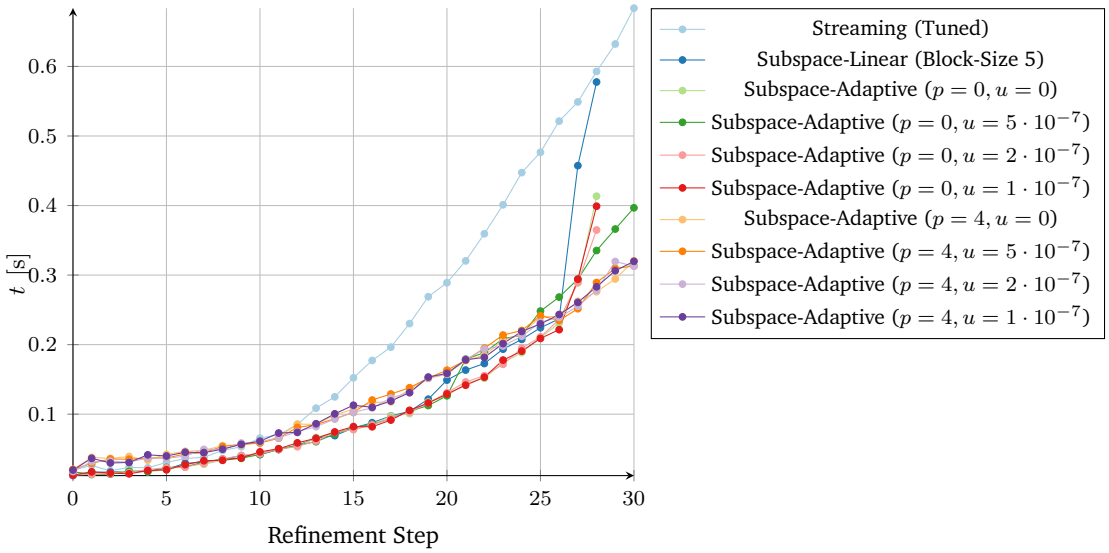

Figure 5.76: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of five.

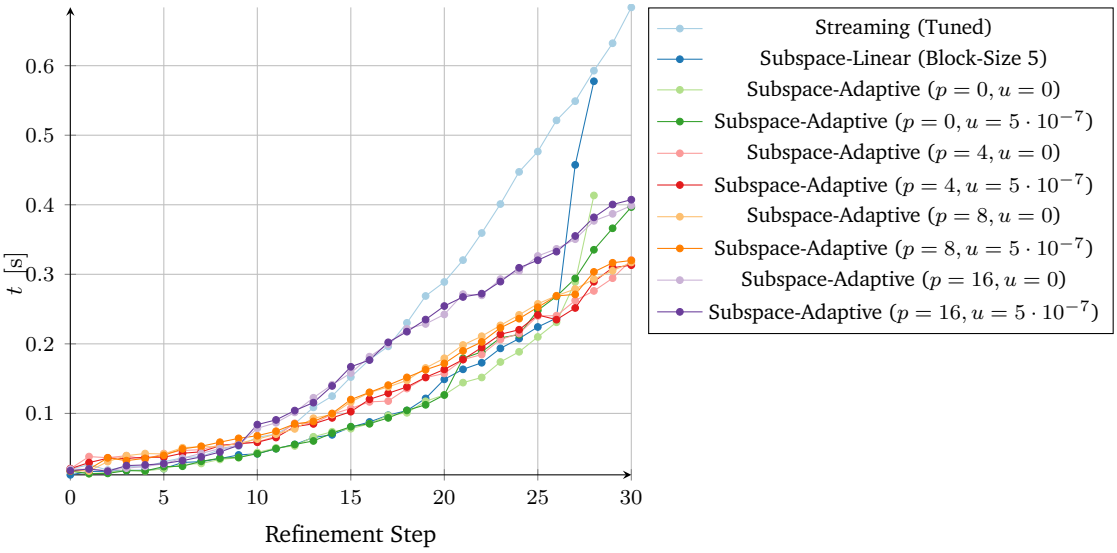

Figure 5.77: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of five.

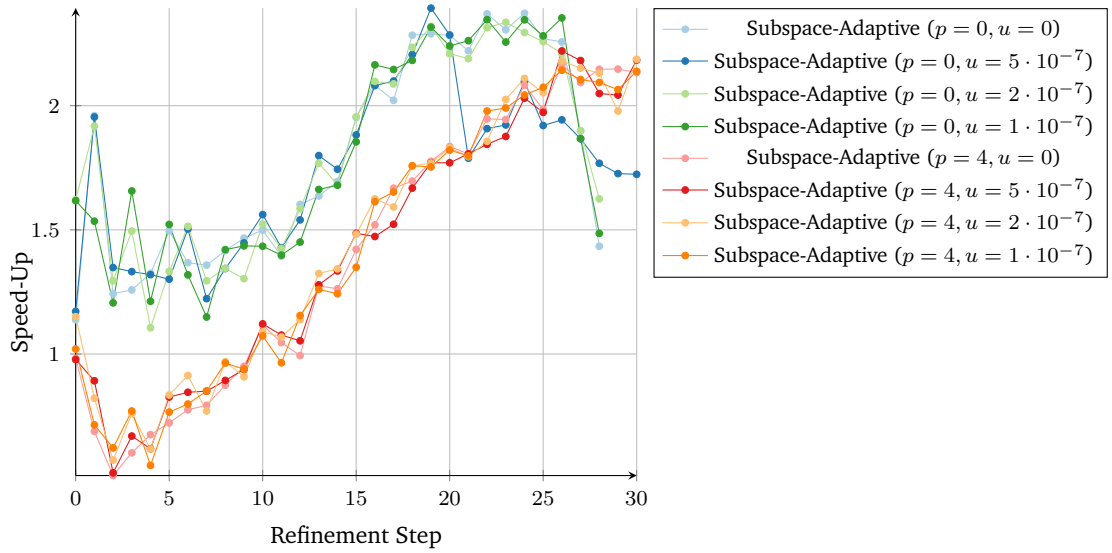

Figure 5.78: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of five has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

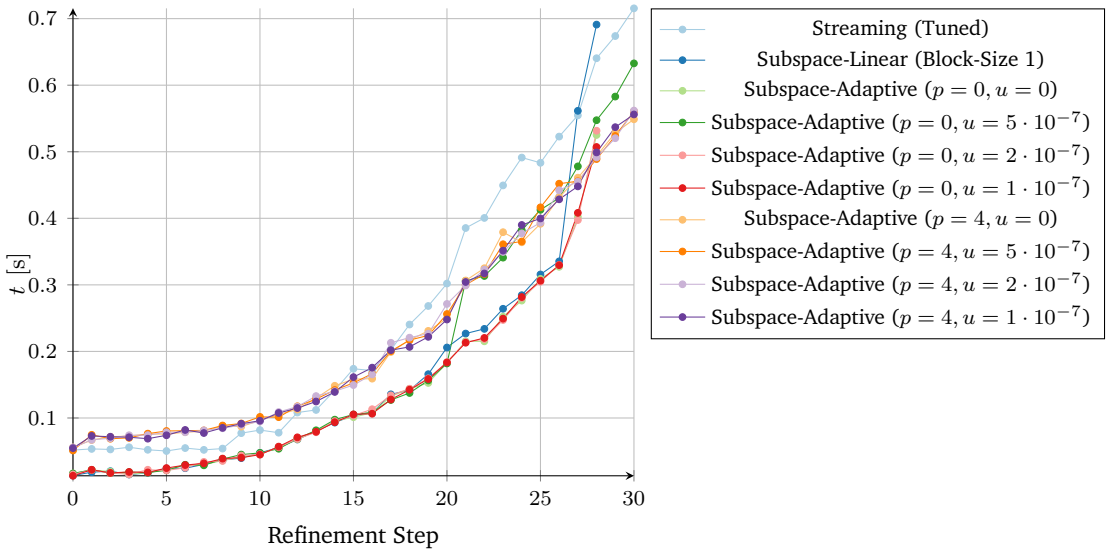

Figure 5.79: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of one.

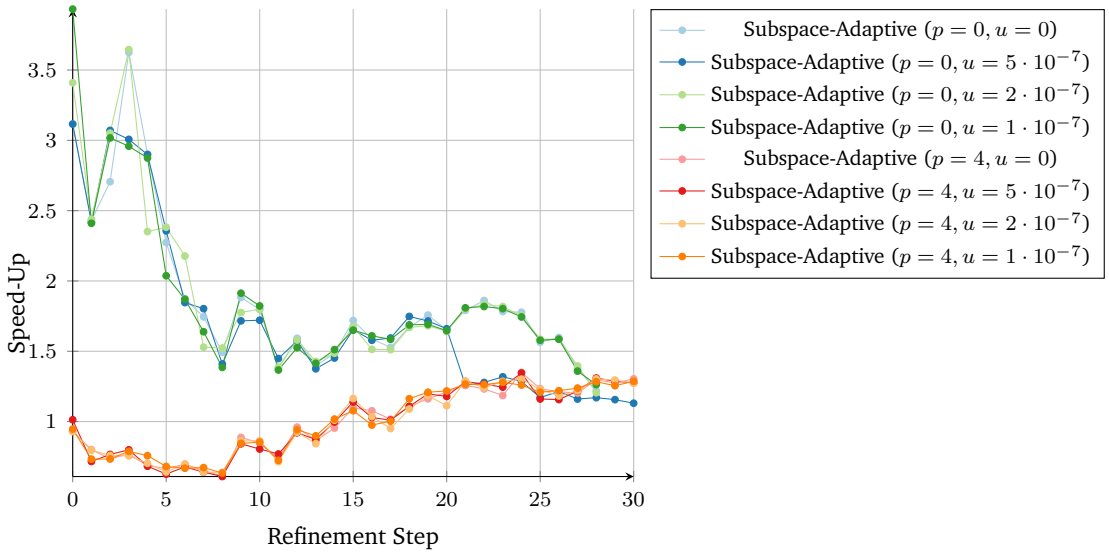

Figure 5.80: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of one has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

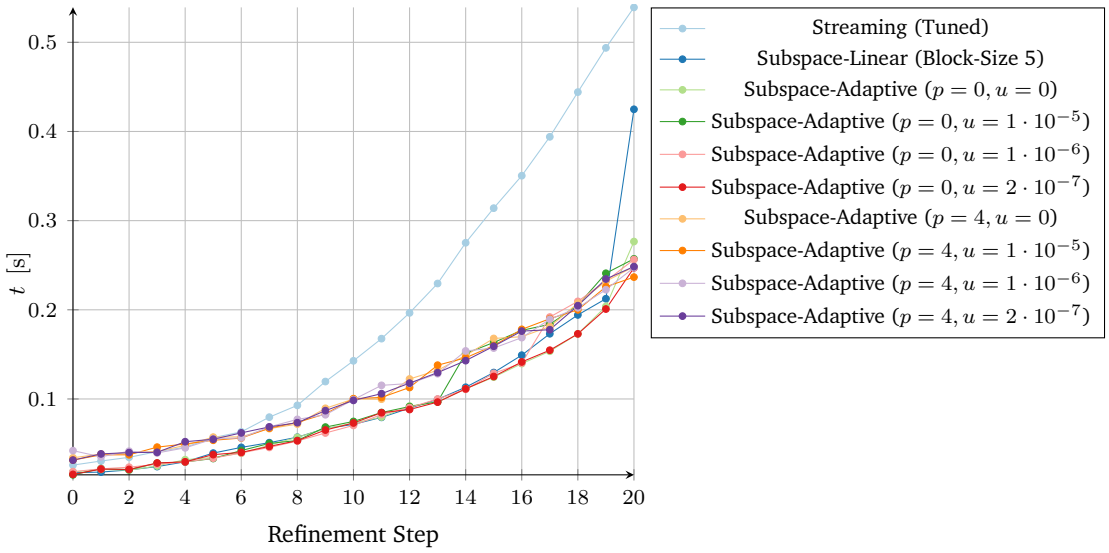

Figure 5.81: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of five.

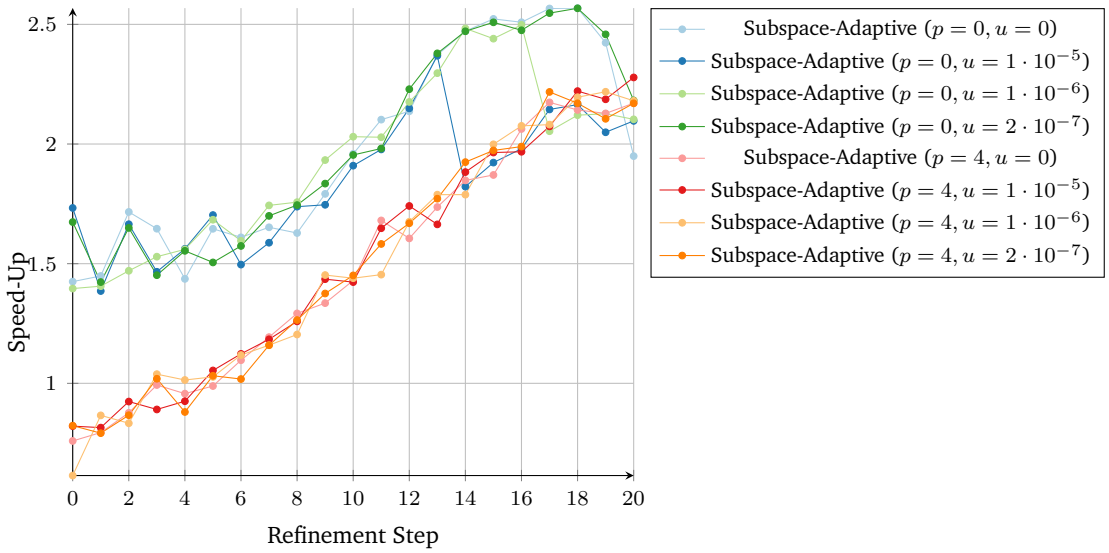

Figure 5.82: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of five has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

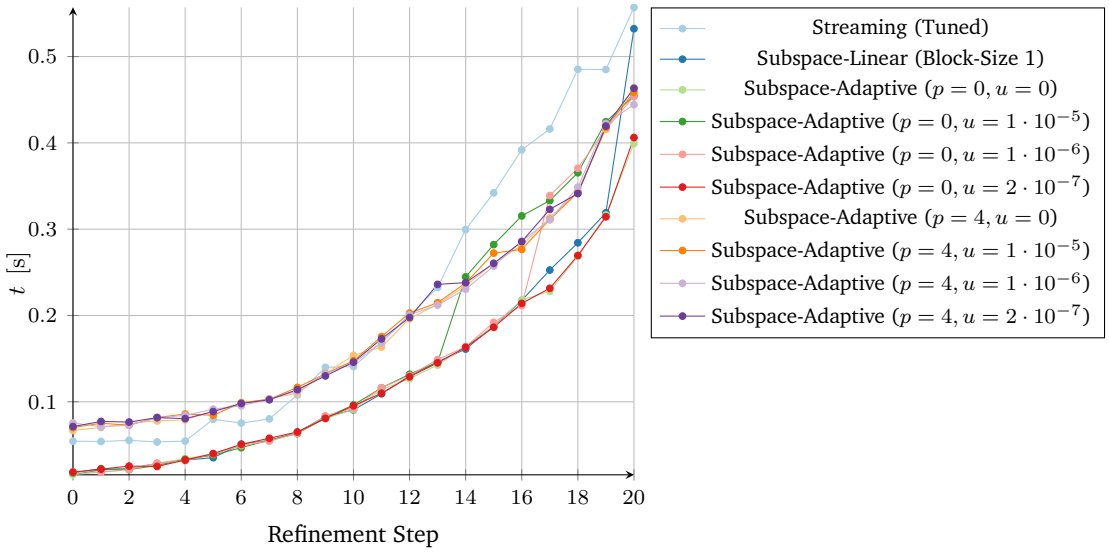

Figure 5.83: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of one.

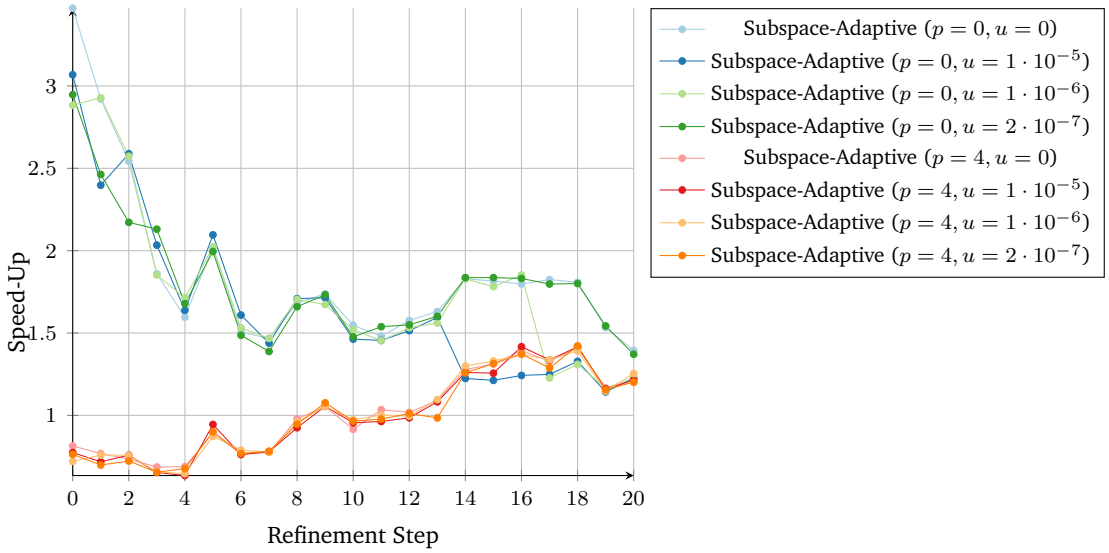

Figure 5.84: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of one has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

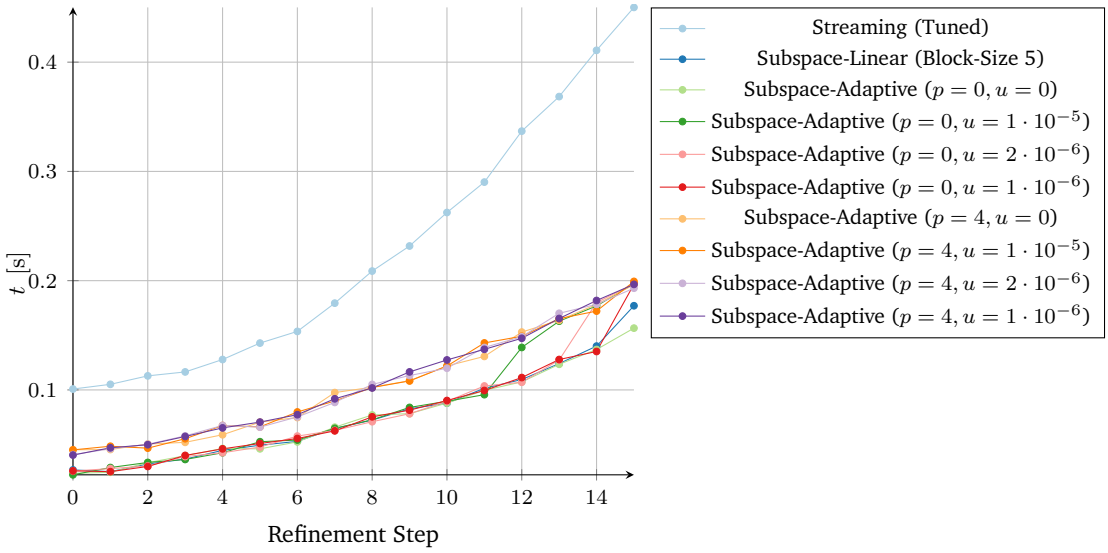

Figure 5.85: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of five.

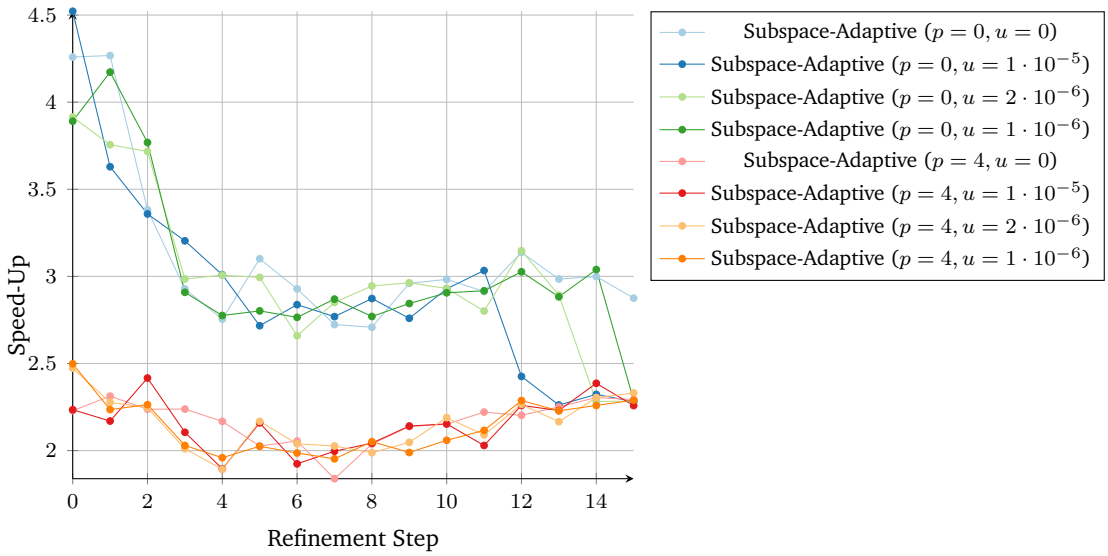

Figure 5.86: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of five has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

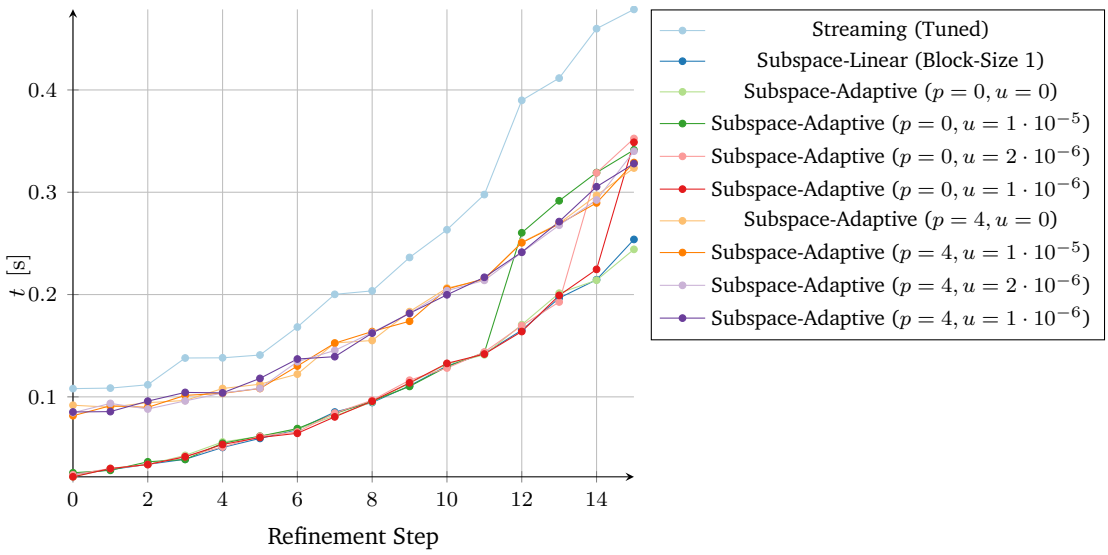

Figure 5.87: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm (including from the subspace-adaptive) use a block-size of one.

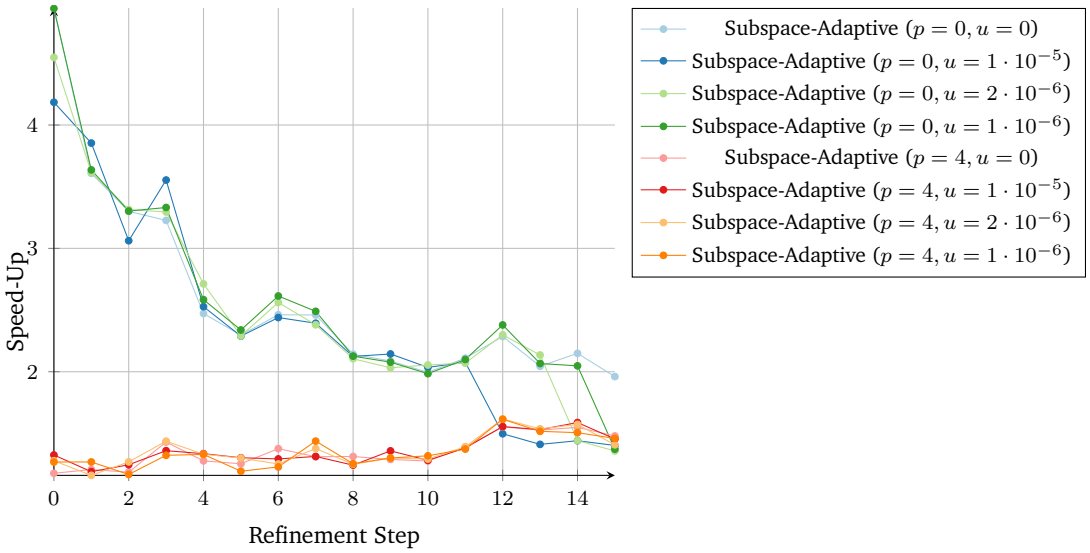

Figure 5.88: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the SDSS DR5 data set, and 64 bit precision. Preparation steps are not included. A block-size of one has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

Generally, a point-threshold of $p = 4$ has been proven to be the best overall selection (in our limited set of tested parameters) other than $p = 0$ for both operators, $B^\intercal\boldsymbol{\alpha}$ and $B\mathbf{v}$, see for example Figures 5.90 and 5.99. On this scenario, however, a point-threshold of $p = 8$ performs notably better than $p = 4$ for $B\mathbf{v}$ on the final six refinement steps (Figure 5.93). It make sense, that, for some workloads, the best point threshold value tends to be higher for $B\mathbf{v}$ than for $B^\intercal\boldsymbol{\alpha}$ due to the use of atomics in the subspace-linear algorithm, which introduces an additional overhead on a per grid-point basis, as the results of this operator are accumulated per grid-point (by use of said atomics).

On the Ten-Dimensional Gaussian Data Set

For the ten-dimensional Gaussian data set we can again observe similar results with respect to the previous evaluations of this data set for the subspace-linear agorithm and the evaluations of the five-dimensional Gaussian data set (Figures 5.105 to 5.113). Due to the higher kernel execution times, the preparation steps are relatively small in comparison to previous scenarios, however, their impact can still be observed (Figures 5.105 and 5.106). We are again unable to out-perform the subspace-linear algorithm.

On the HIGGS Data Set with a Level Two Base Grid

The HIGGS data set provides, in comparison to the previously discussed scenarios, different and interesting results. We again used the scenario given in Table 5.4, however, limited it to ten refinement steps due to the large execution encountered in the later steps.

Even with all preparation steps included, the speed-up of the subspace-linear algorithm over the streaming algorithm is largely one for operator $B^\intercal\boldsymbol{\alpha}$, with a trend to be slightly more than one on the last three refinement steps for a point threshold of $p = 4$ and utilization thresholds of $u = 1 \times 10^{-2}$ and $u = 1 \times 10^{-3}$ (Figures 5.114 and 5.115). This shows that we can indeed beat the subspace-linear algorithm (at least on some scenarios for the operator $B^\intercal\boldsymbol{\alpha}$), even if we encounter no performance-penalties due to its memory-requirement, as well as the streaming algorithm (ever so slightly) by a clever combination of both algorithms.

For the non-transposed multi evaluation $B\mathbf{v}$ we also converge towards a speed-up of one, however are not able to exceed it (Figures 5.116 and 5.117).

5.4.2 Conclusion and Outlook

Based on the results above, we can conclude that the subspace-adaptive algorithm is able to outperform the streaming algorithm by similar margins as the subspace-linear algorithm, but is on many scenarios not notably faster than the subspace-linear algorithm itself. However, we also could obviate the high memory usage of the subspace-linear algorithm to some extent, by combining it with the streaming algorithm and thus provide speed-ups similar

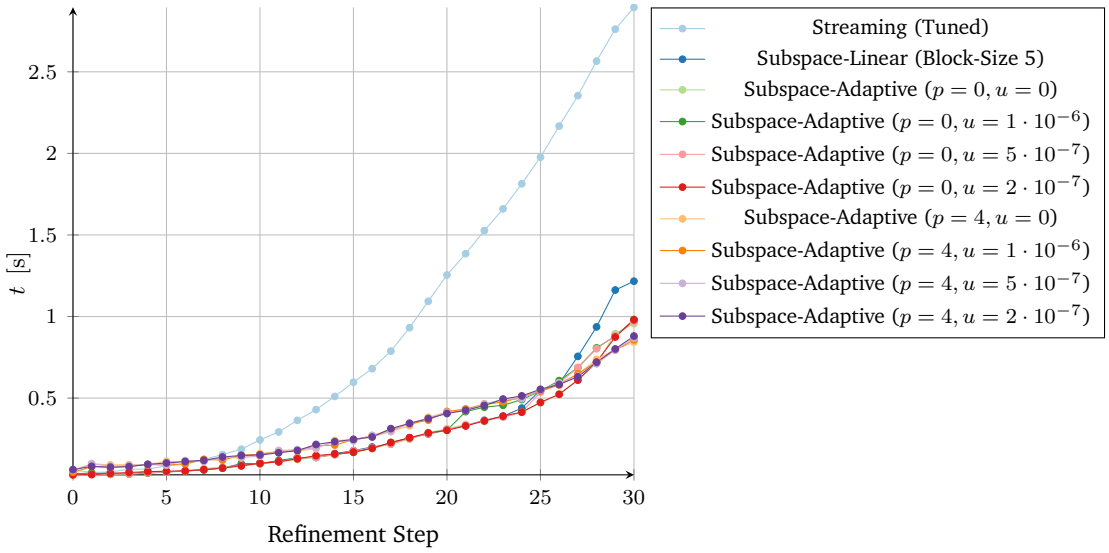

Figure 5.89: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

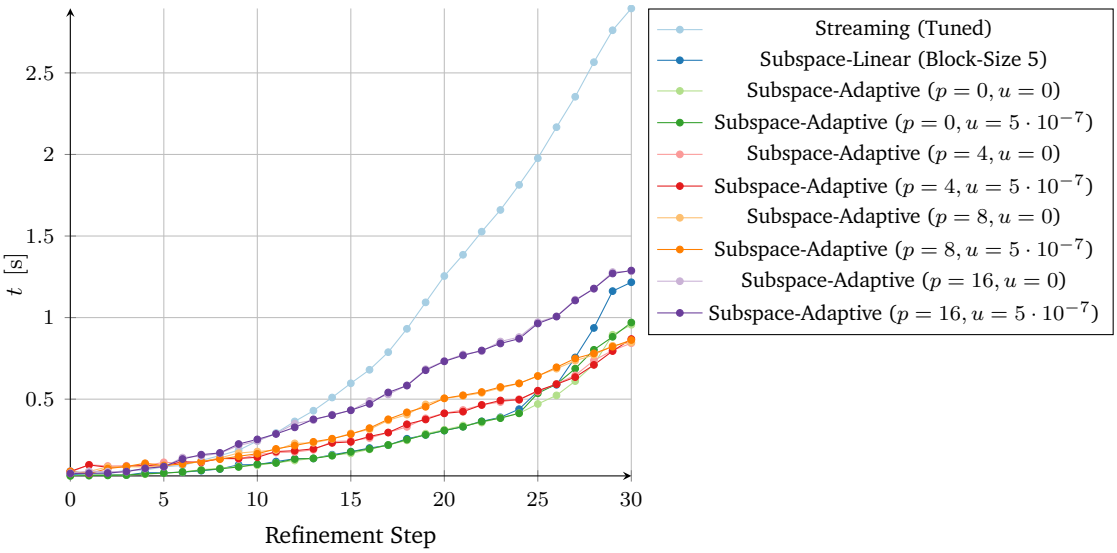

Figure 5.90: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

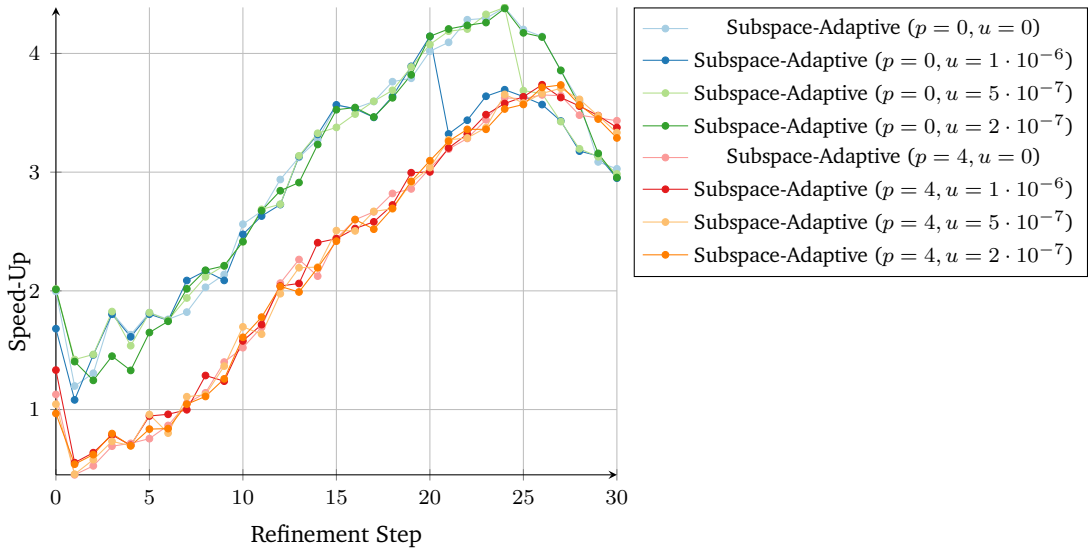

Figure 5.91: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

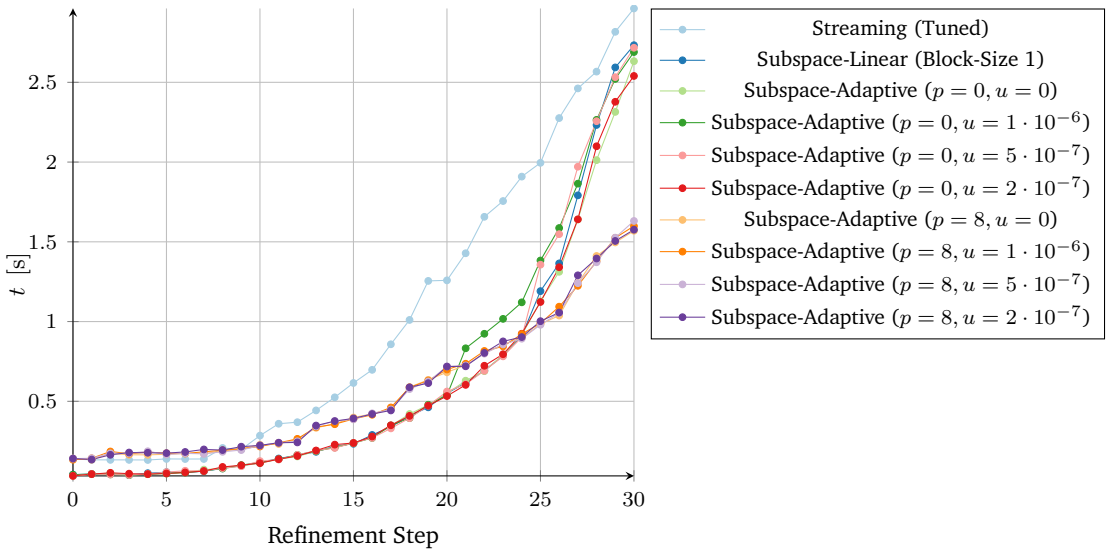

Figure 5.92: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

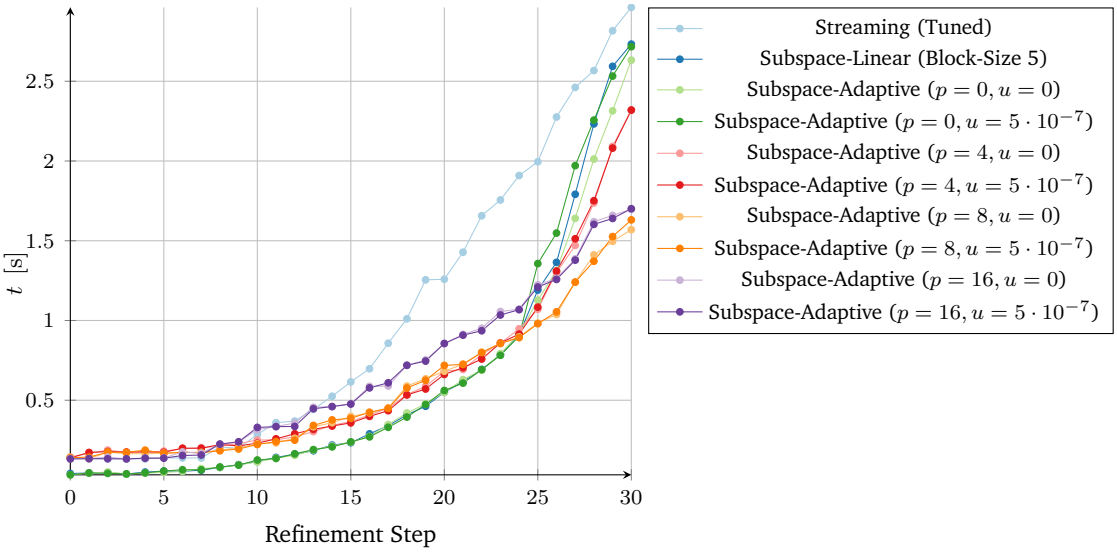

Figure 5.93: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

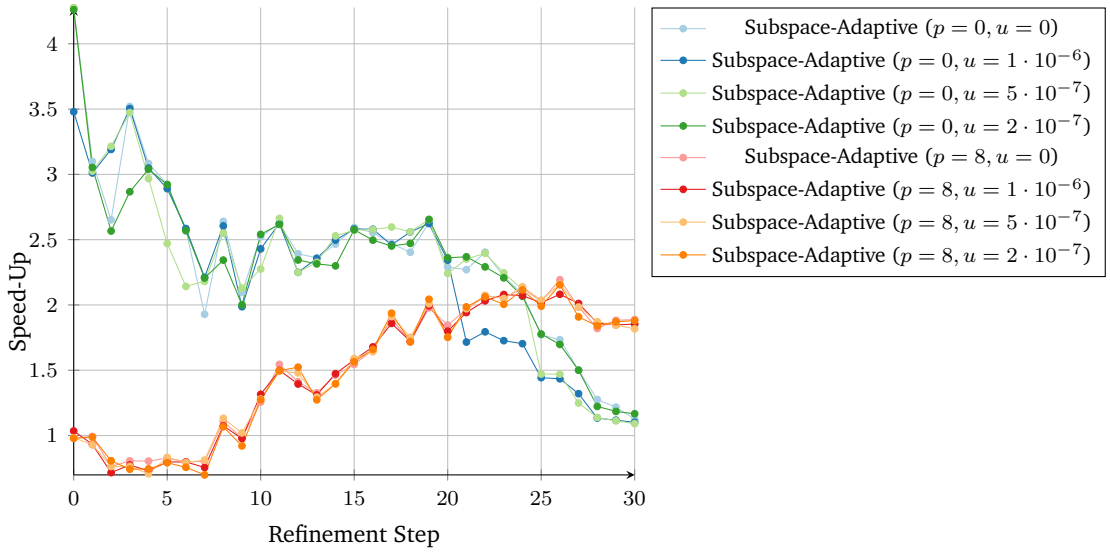

Figure 5.94: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

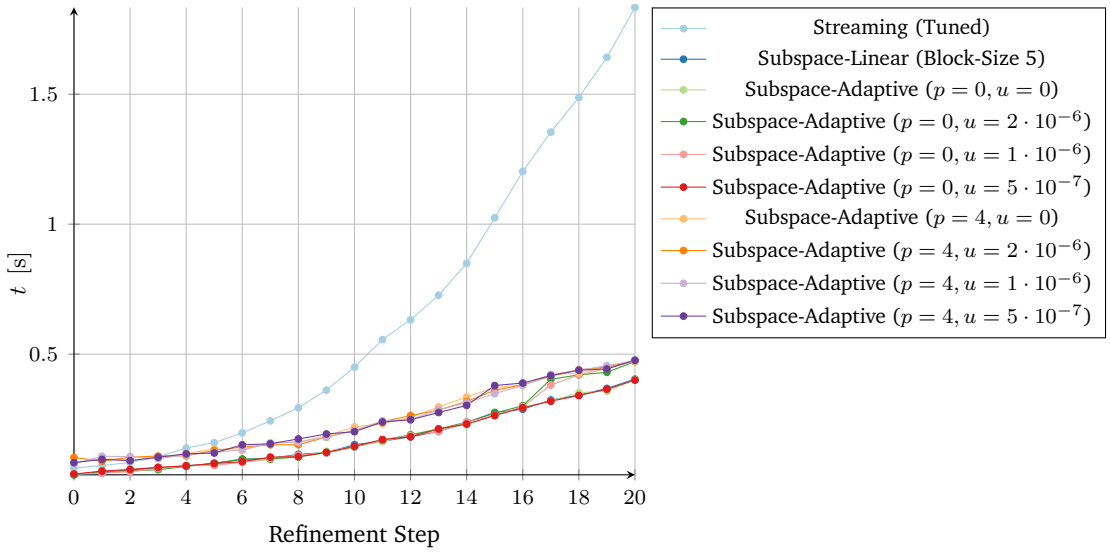

Figure 5.95: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

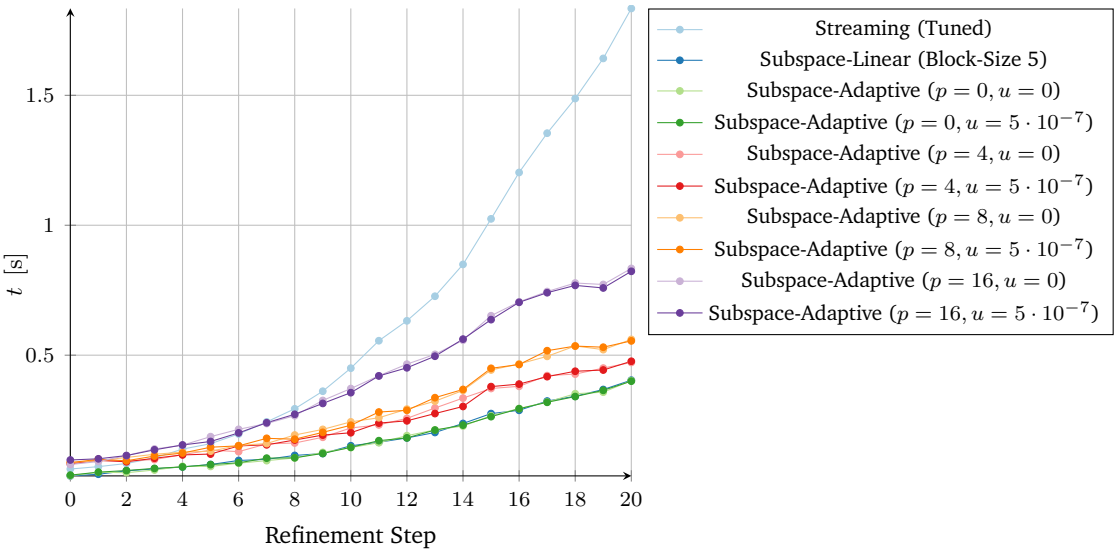

Figure 5.96: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

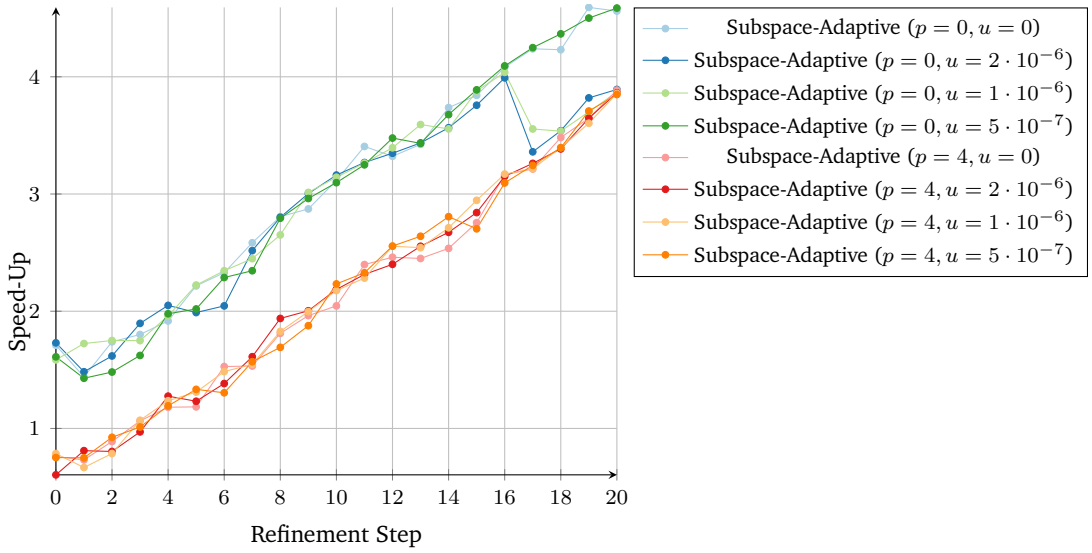

Figure 5.97: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

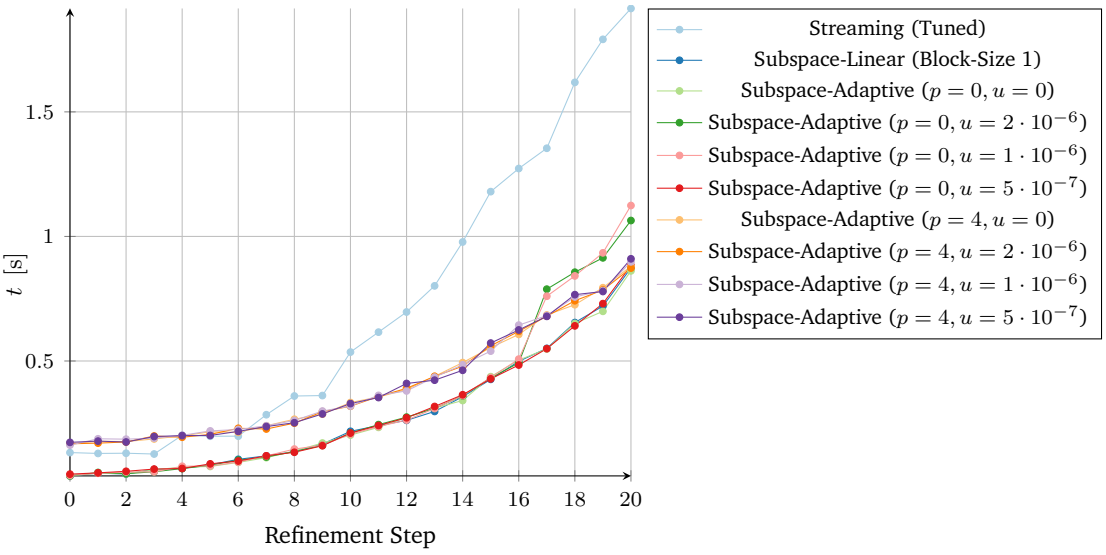

Figure 5.98: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

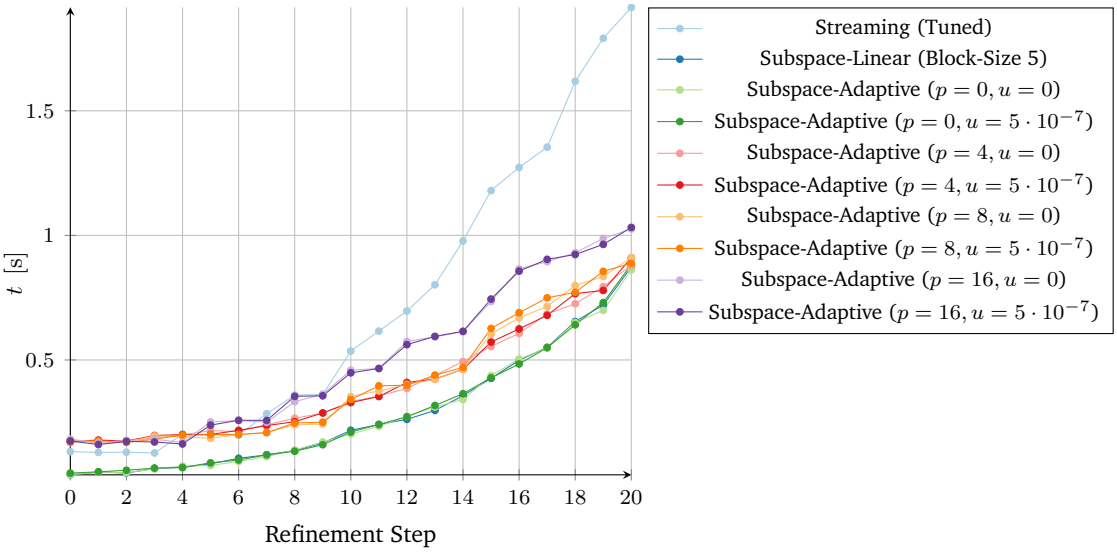

Figure 5.99: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

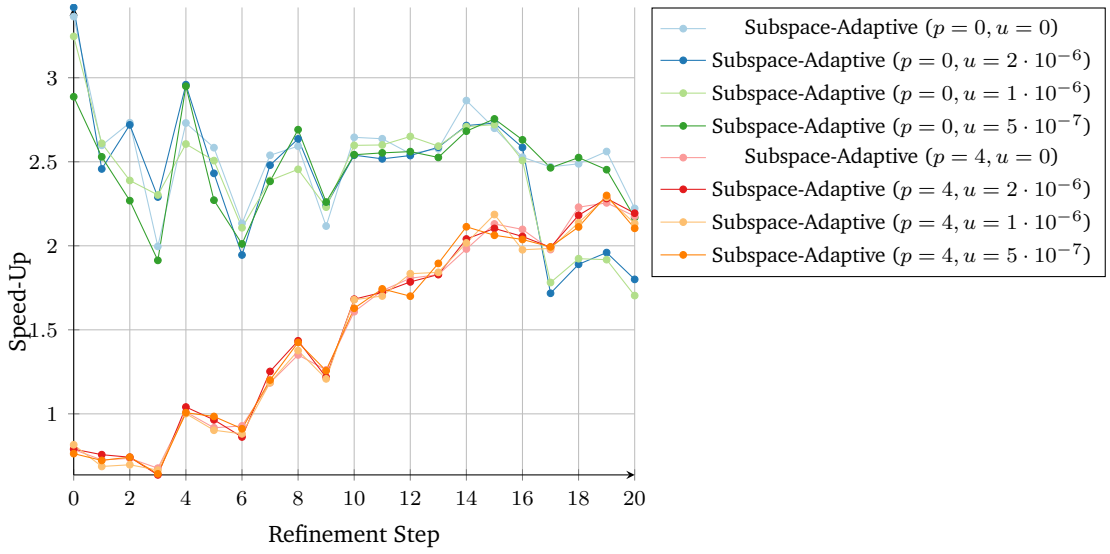

Figure 5.100: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

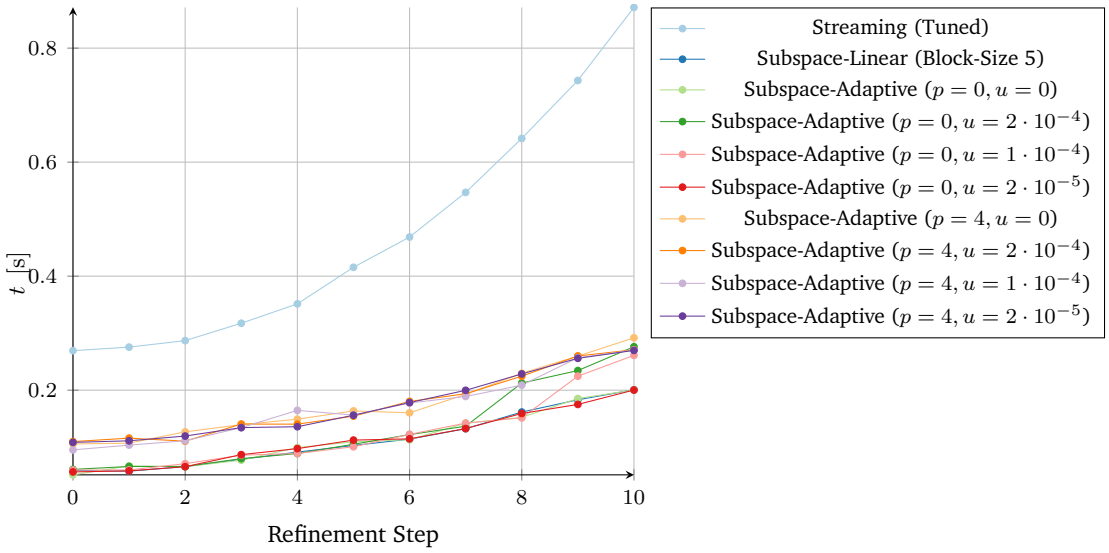

Figure 5.101: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

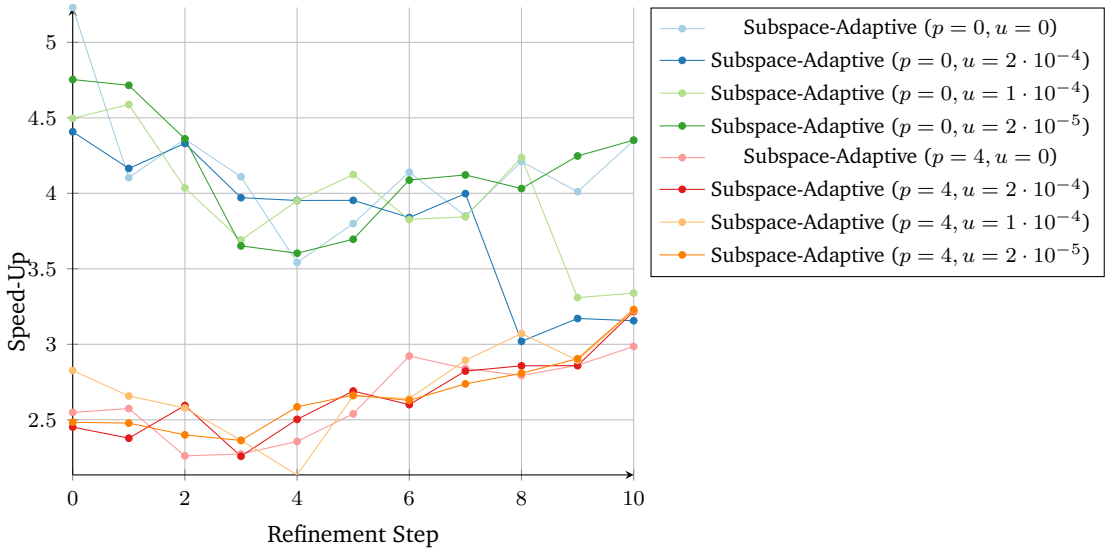

Figure 5.102: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

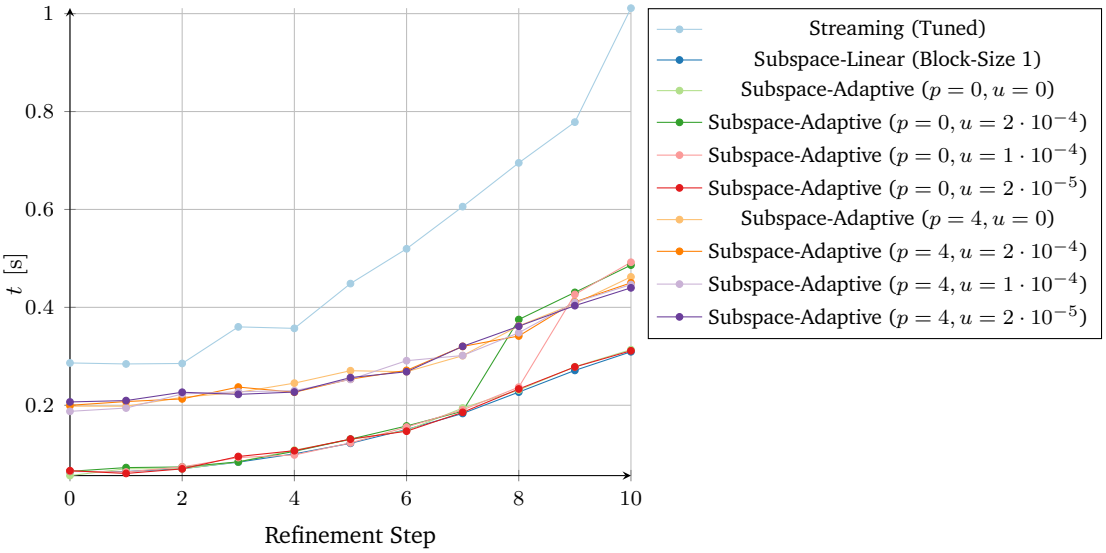

Figure 5.103: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

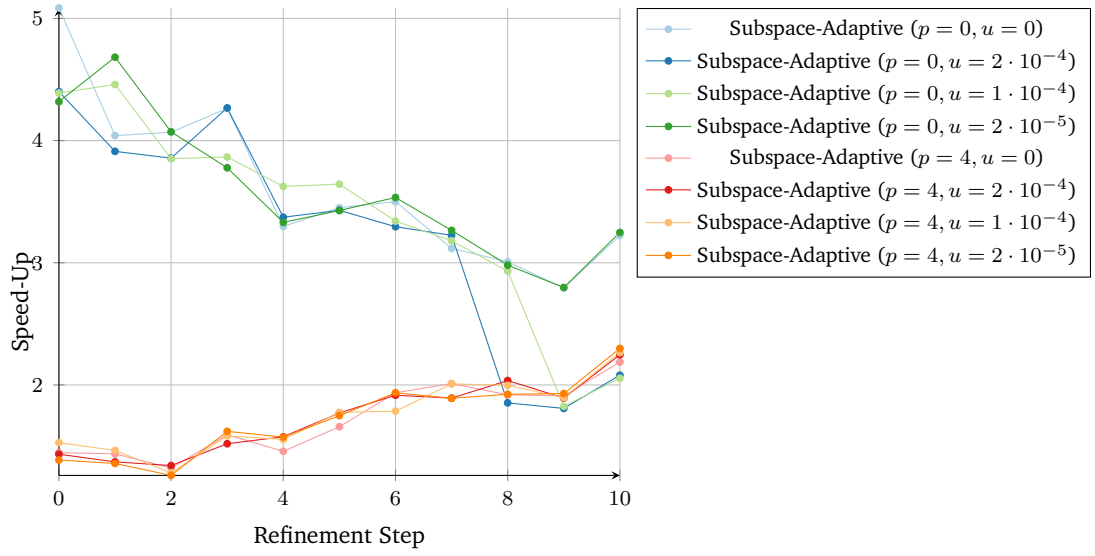

Figure 5.104: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the five-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. A block-size of six has been chosen for all executions of the subspace-linear algorithm (including those from the subspace-adaptive algorithm).

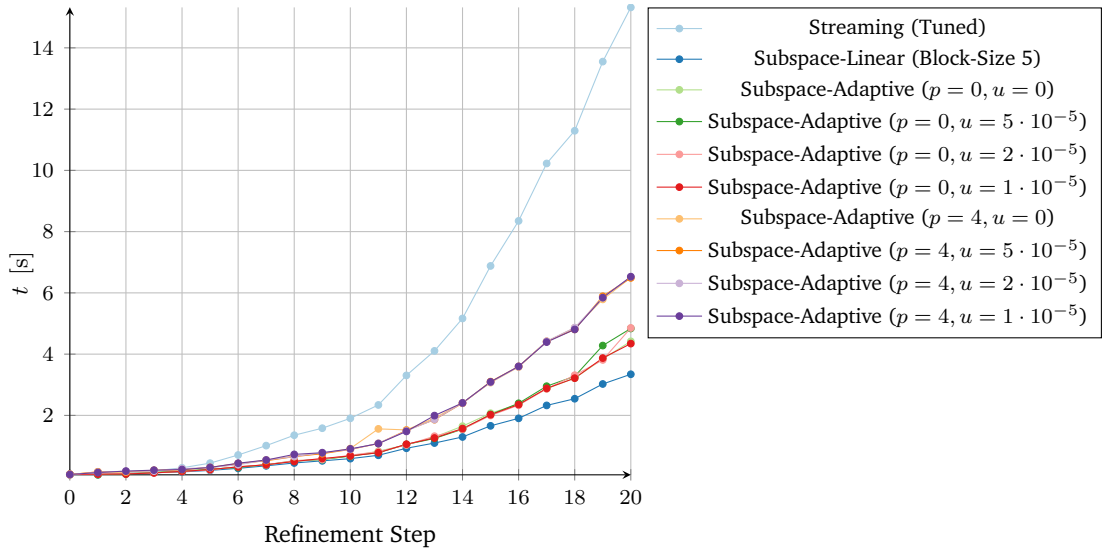

Figure 5.105: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

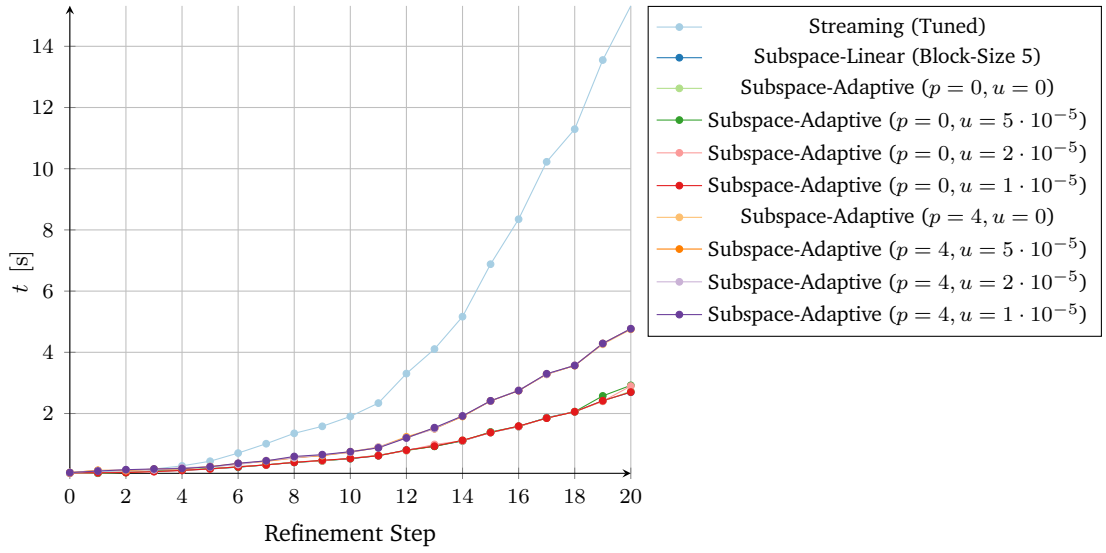

Figure 5.106: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\alpha$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

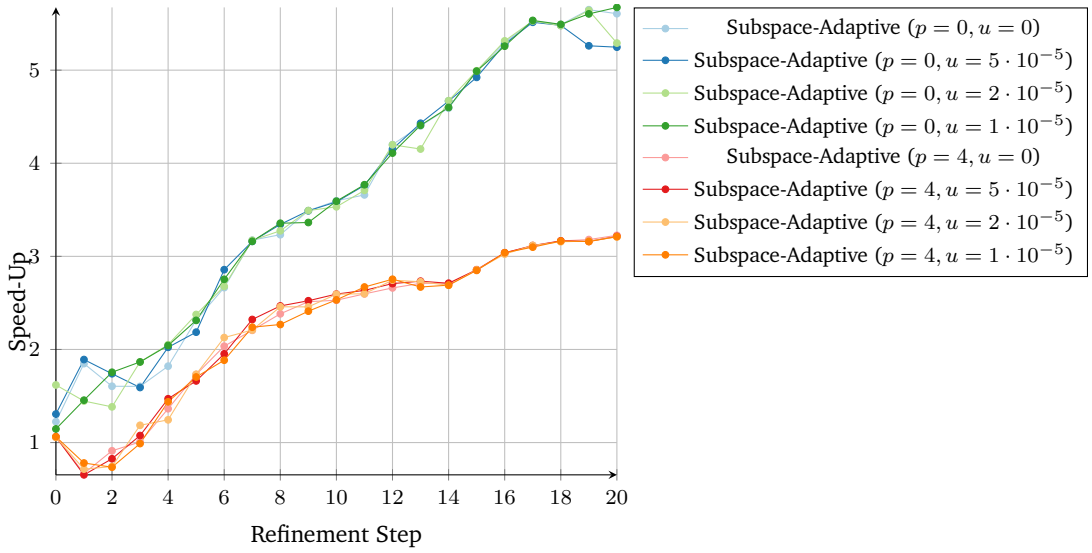

Figure 5.107: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. All executions of the subspace-linear algorithm use a block-size of six.

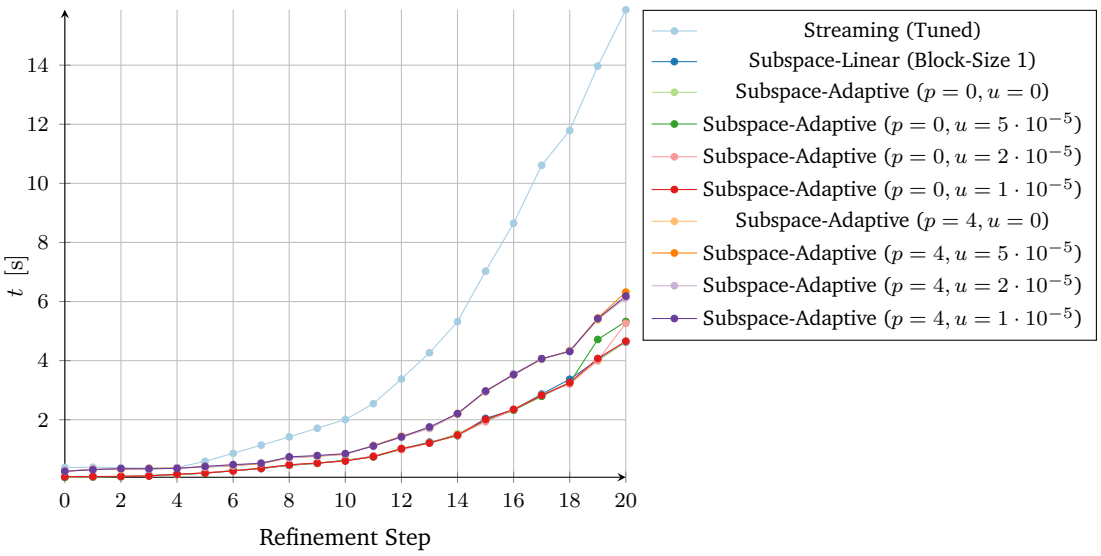

Figure 5.108: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

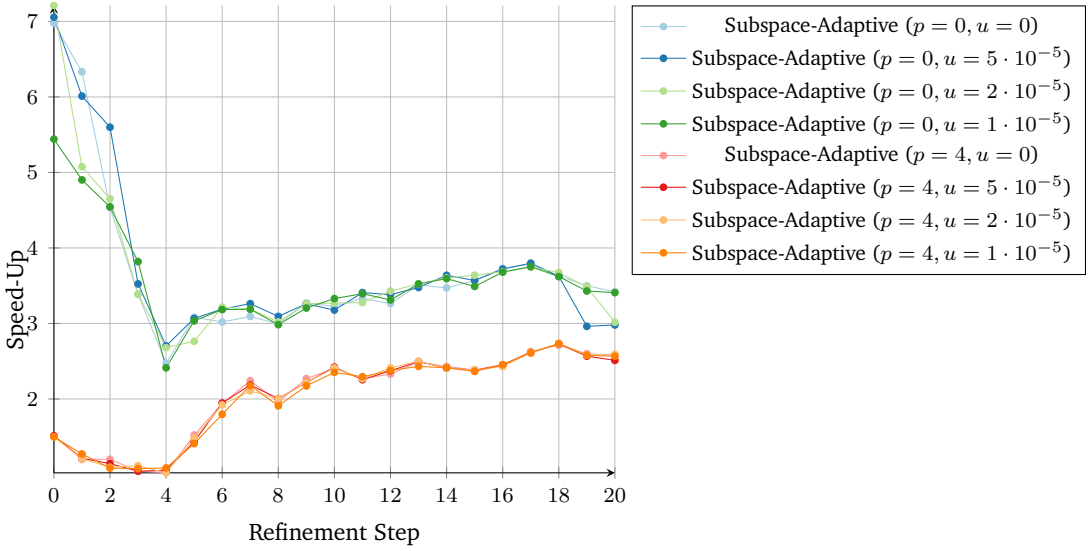

Figure 5.109: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the ten-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. All executions of the subspace-linear algorithm use a block-size of six.

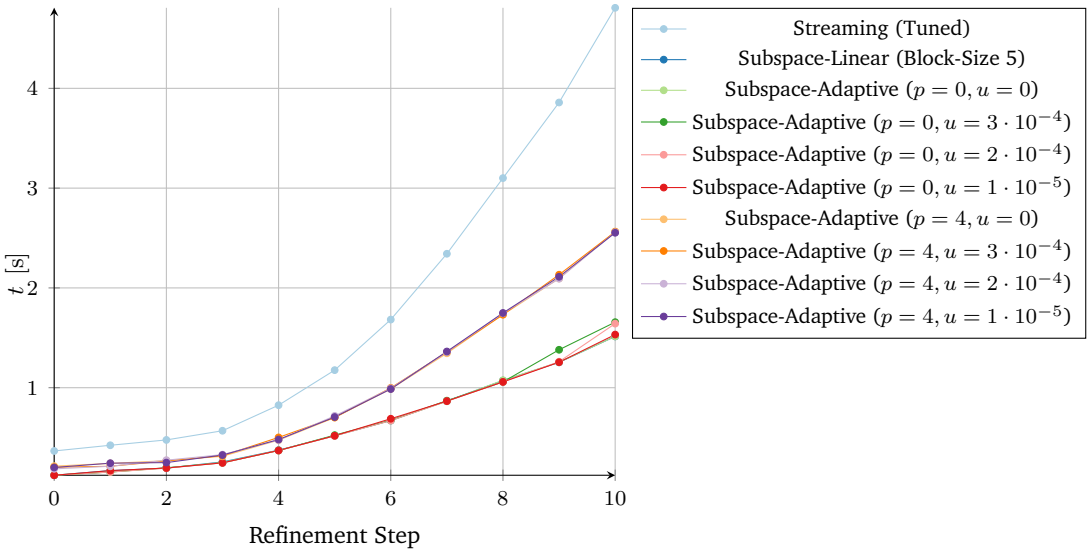

Figure 5.110: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

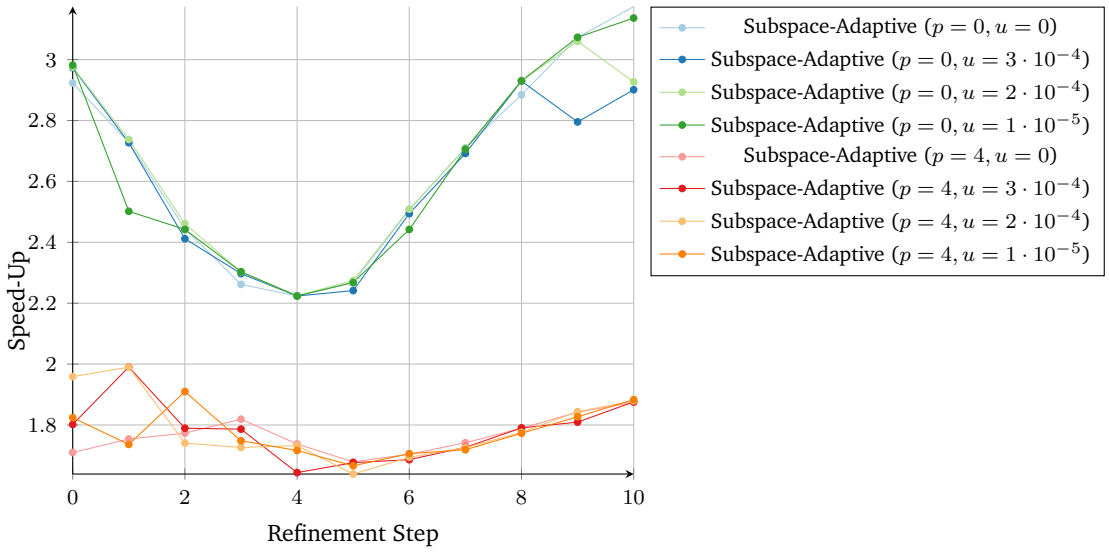

Figure 5.111: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^T\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. All executions of the subspace-linear algorithm use a block-size of six.

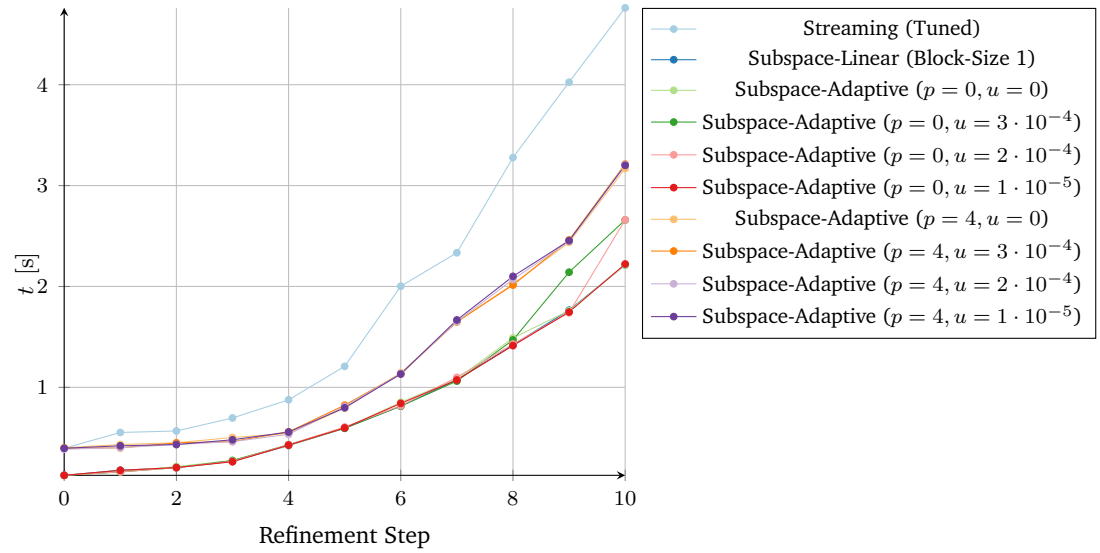

Figure 5.112: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Timings for the subspace-linear and subspace-adaptive algorithms do not include preparation steps. All executions of the subspace-linear algorithm use a block-size of six.

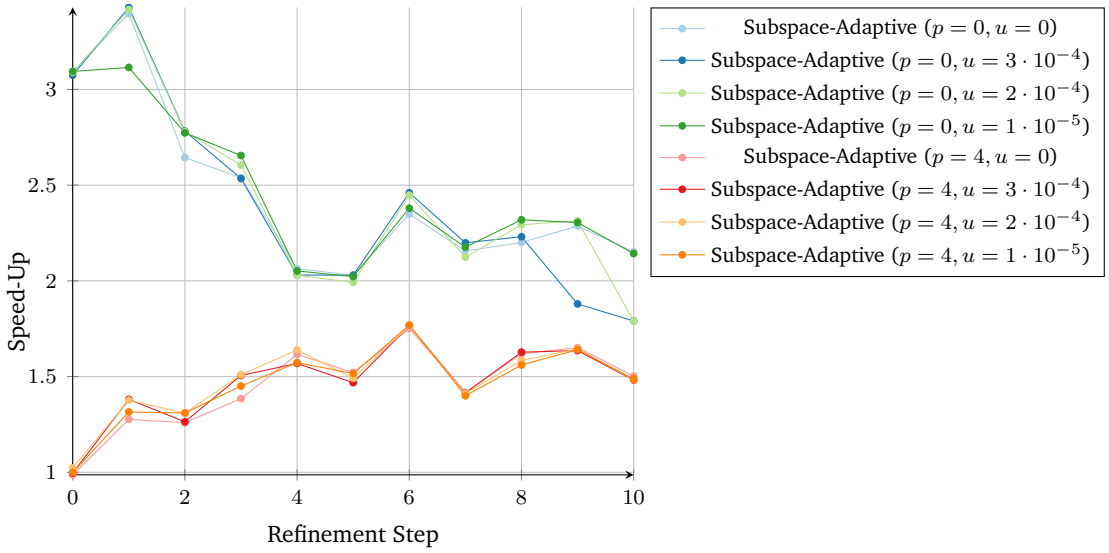

Figure 5.113: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level five, the ten-dimensional Gaussian data set, and 64 bit precision. Preparation steps are not included. All executions of the subspace-linear algorithm use a block-size of six.

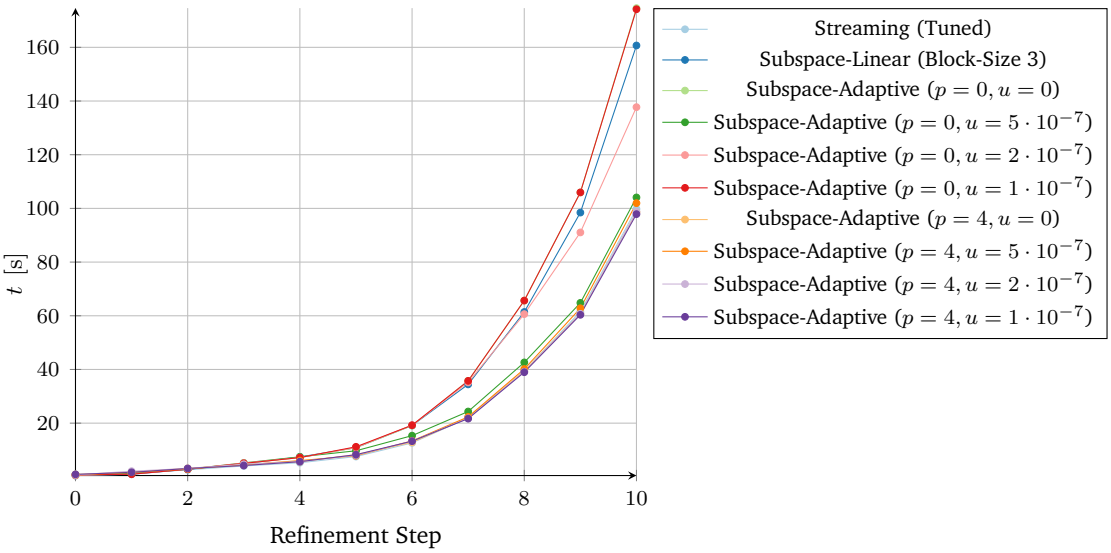

Figure 5.114: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B^\intercal\boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm use a block-size of three.

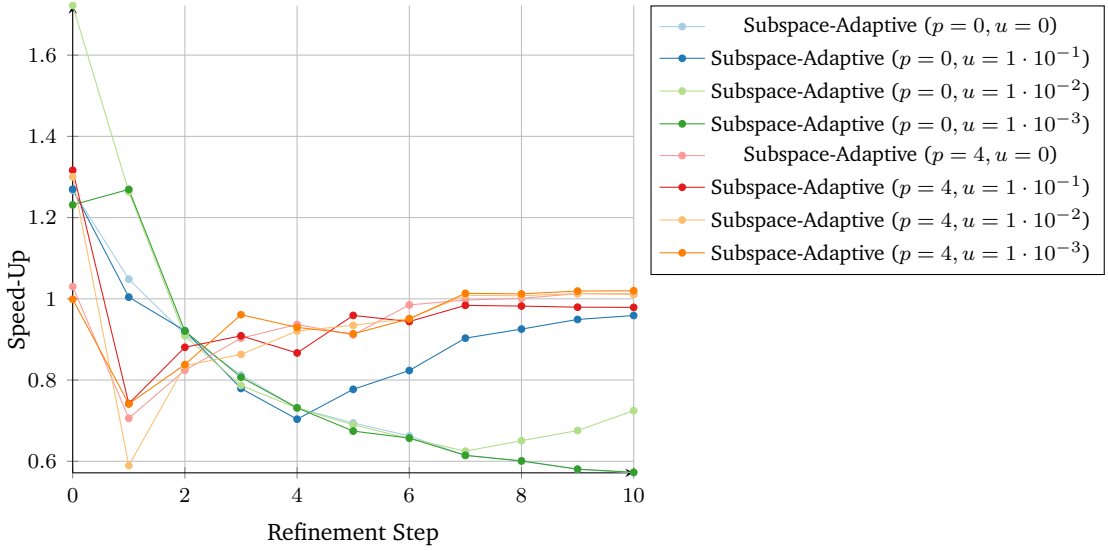

Figure 5.115: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B^\intercal \boldsymbol{\alpha}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the HIGGS data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm use a block-size of three.

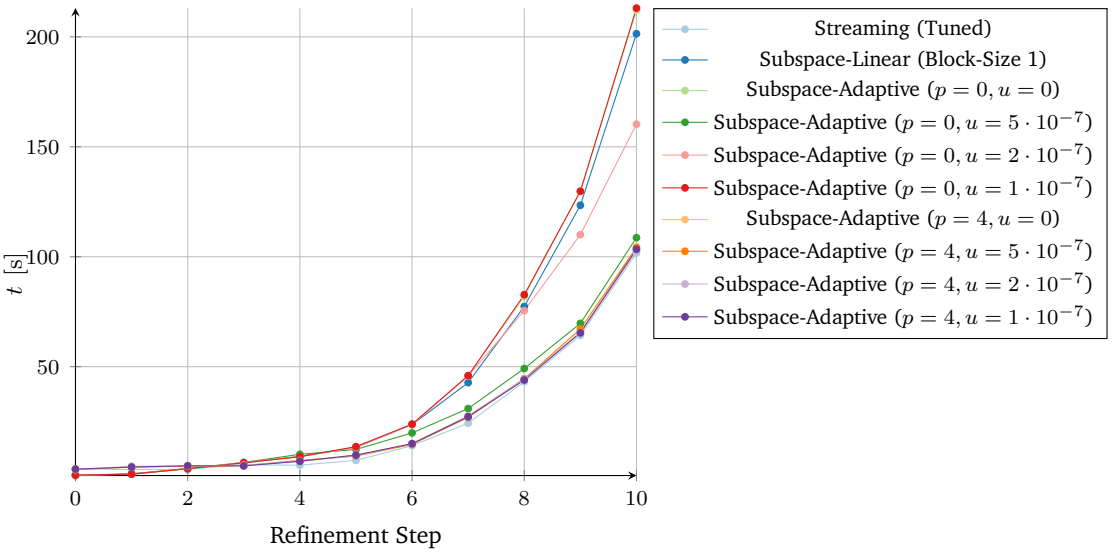

Figure 5.116: Absolute duration of the subspace-adaptive algorithm in comparison to the subspace-linear and streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level two, the HIGGS data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm use a block-size of three.

to the subspace-linear algorithm on grids where the subspace-linear algorithm could not be executed due to this issue. The only scenario where the subspace-adaptive algorithm outperformed the subspace-linear algorithm without such problems is based on the HIGGS data set, which has, as seen in the previous chapter, proven to be notably difficult for this algorithm.

We have also seen that the utilization-threshold is somewhat flawed, as it is influenced by the number of actual grid-points on the subgrid too easily. For example, the addition of a single grid-point to a subgrid of size one doubles its utilization, however, if the potential maximum size of the subgrid is large, it is, in many cases, still not advisable to include it, due to its impact on memory. To this end, we suggest the evaluation of the following replacement for the utilization-threshold:

$$\text{uth}_{\hat{u},u} = \begin{cases} \text{streaming} & \text{if } |\hat{S}| > \hat{u} \text{ and } |S| < u \\ \text{subspace-linear} & \text{otherwise} \end{cases} \tag{5.1}$$

where

S	is the subspace to be evaluated,
$\lvert S\rvert$	is the number of basis functions, i.e. grid points, in S,
$\lvert\hat{S}\rvert$	is the maximum number of grid points that could potentially be in S, and
u and $\hat{u}$	define the selector.

This selector could then be combined with the point-based threshold as described in Section 4.2.

While the implementation of the subspace-adaptive algorithm is largely a preparation step, thus outside the critical code path, and rather insignificant for larger scenarios (see the HIGGS scenario for an example), it could still be improved. A limitation of our implementation is the successive execution of the different sub-algorithms. Using, for example, CUDA streams and two kernels implemented in CUDA, we could schedule both in parallel, which could lead to a better utilization of the GPU and reduced overhead.

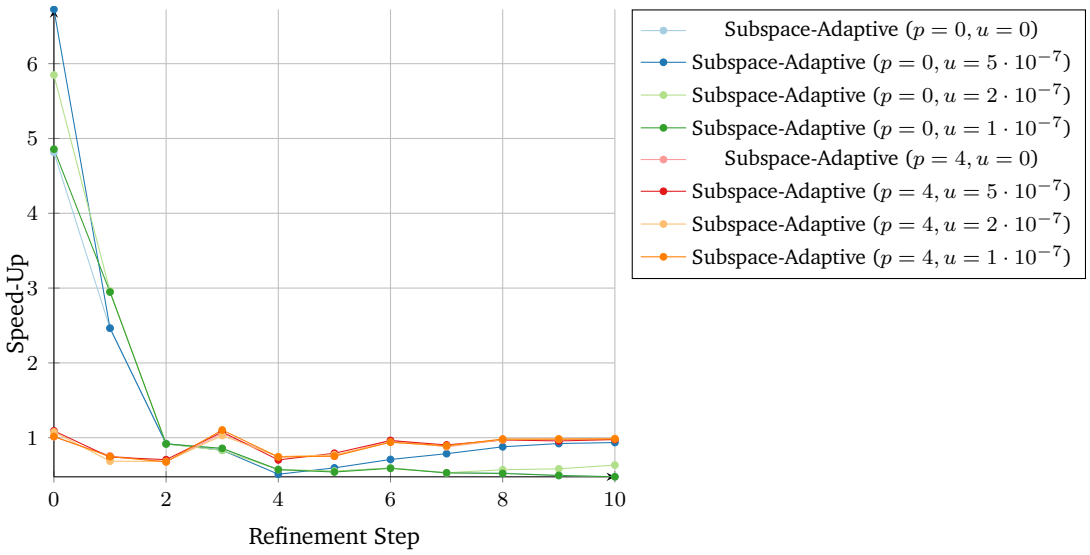

Figure 5.117: Speed-up of the subspace-adaptive algorithm over the streaming algorithm for $B\mathbf{v}$ on the NVIDIA Tesla P100, using spatially adaptive sparse grids with base-level seven, the HIGGS data set, and 64 bit precision. All timings include preparation steps. All executions of the subspace-linear algorithm use a block-size of three.

6 Discussion and Outlook

In this thesis, we have presented a short introduction to the GPU-architecture, on which we implemented our algorithms, and an introduction to regression on spatially adaptive sparse grids. We have then discussed already existing regression algorithms for such grids, after which we have presented an implementation of the subspace-linear algorithm on GPUs. To mitigate some of its problems, we introduced the subspace-adaptive algorithm, switching between subspace-linear and streaming algorithm.

Both algorithms have been validated and tested against the highly optimized streaming algorithm provided by the SG++ framework [HP13] on multiple data sets, ranging from 5 to 28 dimensions, using multiple surplus-based refinement scenarios. Both algorithms have shown speed-ups over the streaming algorithm on most scenarios, however, the subspace-adaptive algorithm has not been able to out-perform the subspace-linear algorithm, except in scenarios where the specific problems of the subspace-linear algorithm have become evident. For such scenarios, the subspace-adaptive algorithm presents a viable alternative to the streaming algorithm. As already discussed in their individual sections (Sections 5.3.3 and 5.4.2), we believe that there is still potential for improvements to both of our implementations.

A Statistics for Adaptive Sparse Grid Evaluation Scenarios

In this appendix, we provide further details for the spatially adaptive sparse grid evaluation scenarios presented in Section 5.2 (specifically Table 5.4).

Tables A.1 to A.9 contain information about the spatially adaptive sparse grids generated after each refinement step for the respective scenarios, a refinement step of zero indicates the entry for the regular base sparse grid. The number of grid points used states how many grid points the grid actually has, while the maximum number of grid points is the number of grid points the grid would have if all subgrids were full grids, latter of which can be used to calculate the memory required to store all surpluses for the subpace-linear algorithm. The average subgrid utilization is the utilization, i.e. the ratio of used to potential maximum grid points, of all individual subgrids averaged (and therefore not the utilization of the grid). It can be used to estimate the regularity of these grids, as it, in contrast to the average number of points per subgrid, incorporates their maximum size. The last four columns are given to provide some insight into the absolute size of the largest subgrids, which are determined by both, the numbers of actually used and maximum potential grid points on that subgrid.

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	11	11	6	100.00 %	1.83	2	2	2	2
1	71	71	21	100.00 %	3.38	4	4	4	4
2	351	351	56	100.00 %	6.27	8	8	8	8
3	900	1471	126	71.68 %	7.14	11	16	11	16
4	1514	5375	248	44.43 %	6.10	12	16	9	32
5	2275	14 751	396	31.57 %	5.74	12	16	8	64
6	3307	38 847	603	22.85 %	5.48	13	32	10	128
7	4442	83 839	801	18.49 %	5.55	15	32	8	256
8	5924	196 735	1103	14.30 %	5.37	15	32	7	512
9	7778	422 527	1418	11.75 %	5.49	15	32	9	1024
10	9691	849 919	1783	9.85 %	5.44	16	32	7	2048
11	12 003	1 603 711	2132	8.69 %	5.63	16	32	6	4096
12	15 371	3 361 343	2677	7.28 %	5.74	17	128	6	8192
13	19 669	6 707 903	3293	6.28 %	5.97	18	64	6	16 384
14	24 065	13 082 047	4028	5.38 %	5.97	20	512	6	32 768
15	30 121	25 782 399	4858	4.69 %	6.20	21	128	6	65 536
16	35 333	44 684 415	5349	4.46 %	6.61	25	1024	6	131 072
17	41 714	89 747 455	6223	3.98 %	6.70	27	1024	6	262 144
18	47 965	159 725 311	7006	3.65 %	6.85	29	1024	6	524 288
19	54 849	244 054 271	7705	3.43 %	7.12	30	256	4	1 048 576
20	62 262	375 044 863	8408	3.24 %	7.41	31	256	3	2 097 152
21	70 307	539 774 847	9257	3.02 %	7.60	33	1024	2	4 194 304
22	79 419	571 278 463	9851	2.94 %	8.06	38	2048	2	4 194 304
23	87 998	686 487 679	10 576	2.83 %	8.32	39	2048	3	4 194 304
24	97 113	751 159 935	11 383	2.68 %	8.53	40	4096	3	4 194 304
25	106 489	937 764 479	12 229	2.55 %	8.71	40	2048	3	4 194 304
26	113 780	1 015 177 471	12 631	2.52 %	9.01	40	2048	3	4 194 304
27	121 658	1 164 712 959	13 301	2.44 %	9.15	41	2048	3	4 194 304
28	131 261	1 529 741 567	14 126	2.33 %	9.29	41	2048	2	8 388 608
29	139 602	2 622 677 247	14 844	2.25 %	9.40	43	4096	3	16 777 216
30	148 431	2 738 560 255	15 549	2.18 %	9.55	45	4096	3	16 777 216

Table A.1: Statistics of the grids created for the SDSS DR5 data set (371 908 data points, 5 dimensions) using surplus-based refinement, starting with a level two regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	1471	1471	126	100.00 %	11.67	16	16	16	16
1	2193	5439	250	59.42 %	8.77	16	16	9	32
2	3119	15 743	411	40.37 %	7.59	16	16	10	64
3	4276	40 479	617	29.05 %	6.93	16	16	12	128
4	5709	96 415	871	21.82 %	6.55	16	16	11	256
5	7437	222 175	1182	16.89 %	6.29	17	64	9	512
6	9890	500 671	1586	13.31 %	6.24	18	64	7	1024
7	12 875	1 076 927	2045	10.91 %	6.30	18	64	7	2048
8	16 977	2 226 047	2557	9.19 %	6.64	19	512	6	4096
9	22 412	4 555 583	3219	7.72 %	6.96	20	128	6	8192
10	27 561	8 665 279	3821	6.78 %	7.21	25	1024	6	16 384
11	34 419	15 852 927	4593	5.88 %	7.49	25	1024	6	32 768
12	41 442	30 805 887	5452	5.12 %	7.60	27	1024	6	65 536
13	49 040	55 904 639	6217	4.66 %	7.89	28	256	6	131 072
14	58 251	108 523 391	7282	4.13 %	8.00	34	512	6	262 144
15	66 772	189 390 079	8308	3.74 %	8.04	36	512	6	524 288
16	75 276	278 453 759	8871	3.61 %	8.49	36	512	6	1 048 576
17	84 444	398 373 119	9484	3.48 %	8.90	39	2048	3	2 097 152
18	95 060	667 786 239	10 312	3.28 %	9.22	43	1024	2	4 194 304
19	106 856	891 597 055	11 204	3.10 %	9.54	46	4096	3	4 194 304
20	119 063	1 111 972 095	12 115	2.93 %	9.83	47	4096	2	8 388 608

Table A.2: Statistics of the grids created for the SDSS DR5 data set (371 908 data points, 5 dimensions) using surplus-based refinement, starting with a level five regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	18 943	18 943	462	100.00 %	41.00	64	64	64	64
1	19 741	43 647	655	71.49 %	30.14	64	64	11	128
2	20 990	106 751	931	51.01 %	22.55	64	64	11	256
3	22 641	233 855	1239	38.87 %	18.27	64	64	11	512
4	25 020	512 127	1612	30.37 %	15.52	64	64	11	1024
5	27 791	1 043 711	2005	24.78 %	13.86	64	64	9	2048
6	31 470	2 022 911	2398	21.08 %	13.12	64	64	8	4096
7	36 717	4 288 383	3101	16.63 %	11.84	64	64	6	8192
8	41 901	8 947 071	3832	13.64 %	10.93	64	64	6	16 384
9	48 592	17 354 495	4670	11.39 %	10.41	64	64	6	32 768
10	55 674	32 556 799	5501	9.83 %	10.12	64	64	6	65 536
11	62 977	60 370 687	6426	8.54 %	9.80	64	64	6	131 072
12	71 932	105 743 615	7316	7.62 %	9.83	64	64	6	262 144
13	81 219	186 369 535	8270	6.83 %	9.82	64	64	6	524 288
14	89 815	278 642 687	8861	6.46 %	10.14	64	64	5	1 048 576
15	99 423	415 405 823	9701	5.97 %	10.25	64	64	3	2 097 152

Table A.3: Statistics of the grids created for the SDSS DR5 data set (371 908 data points, 5 dimensions) using surplus-based refinement, starting with a level seven regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	57	57	29	100.00 %	1.97	2	2	2	2
1	1681	1681	435	100.00 %	3.86	4	4	4	4
2	5551	14 721	2065	44.49 %	2.69	4	4	4	8
3	9632	44 177	3906	30.05 %	2.47	5	16	5	16
4	13 853	105 217	5833	22.43 %	2.37	6	32	6	32
5	20 910	262 289	9011	16.52 %	2.32	7	64	7	64
6	36 432	734 625	16 026	11.55 %	2.27	8	128	8	128
7	63 127	2 030 769	28 129	8.11 %	2.24	9	256	9	256
8	107 746	5 525 649	48 542	5.77 %	2.22	9	256	9	512
9	163 923	13 421 297	74 235	4.31 %	2.21	9	256	9	1024
10	261 332	34 453 393	119 075	3.13 %	2.19	10	2048	10	2048
11	387 830	83 008 817	177 336	2.34 %	2.19	10	2048	10	4096
12	576 580	196 611 105	263 670	1.79 %	2.19	11	8192	11	8192
13	1 066 560	503 831 113	487 401	1.44 %	2.19	12	16 384	12	16 384
14	2 241 597	1 562 636 505	1 016 509	1.03 %	2.21	12	16 384	12	32 768
15	4 176 860	4 595 140 841	1 901 485	0.74 %	2.20	12	16 384	12	65 536

Table A.4: Statistics of the grids created for the truncated HIGGS data set (5×10^6 data points, 28 dimensions) using surplus-based refinement, starting with a level two regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	11	11	6	100.00 %	1.83	2	2	2	2
1	71	71	21	100.00 %	3.38	4	4	4	4
2	351	351	56	100.00 %	6.27	8	8	8	8
3	939	1471	126	73.61 %	7.45	12	16	12	16
4	1730	5119	240	50.77 %	7.21	15	32	15	32
5	2804	15 295	405	36.06 %	6.92	15	32	10	64
6	4205	37 631	599	28.01 %	7.02	16	32	11	128
7	6204	94 911	886	21.33 %	7.00	18	32	8	256
8	9075	239 359	1291	16.40 %	7.03	20	64	7	512
9	12 742	565 503	1812	12.75 %	7.03	23	64	8	1024
10	17 350	1 146 111	2316	11.04 %	7.49	26	64	7	2048
11	22 103	2 134 783	2810	9.70 %	7.87	26	64	5	4096
12	27 589	4 261 503	3486	8.23 %	7.91	26	64	5	8192
13	34 186	6 435 199	3928	7.87 %	8.70	28	128	4	16 384
14	41 227	11 387 263	4652	6.98 %	8.86	28	128	2	32 768
15	47 775	17 650 047	5196	6.52 %	9.19	31	128	2	65 536
16	55 547	31 730 047	6175	5.65 %	9.00	31	128	4	65 536
17	64 728	54 548 991	7267	4.98 %	8.91	33	128	3	131 072
18	78 059	88 397 311	8178	4.58 %	9.54	34	128	2	262 144
19	92 531	162 740 991	9389	4.11 %	9.86	35	256	4	524 288
20	106 048	243 066 879	10 470	3.81 %	10.13	38	128	4	1 048 576
21	117 527	337 545 215	11 106	3.69 %	10.58	39	128	2	2 097 152
22	129 632	397 405 183	11 473	3.62 %	11.30	39	128	3	2 097 152
23	141 414	481 090 047	12 119	3.46 %	11.67	40	512	3	2 097 152
24	153 758	541 669 119	12 659	3.36 %	12.15	40	512	3	2 097 152
25	169 286	793 880 831	14 036	3.07 %	12.06	41	256	2	4 194 304
26	185 579	892 506 879	14 681	2.97 %	12.64	41	256	2	4 194 304
27	202 774	1 105 010 431	15 740	2.81 %	12.88	41	256	2	4 194 304
28	221 817	1 372 762 879	17 047	2.64 %	13.01	43	512	4	4 194 304
29	238 482	1 840 448 255	17 828	2.56 %	13.38	44	512	2	8 388 608
30	250 169	1 915 067 135	18 160	2.56 %	13.78	44	512	4	8 388 608

Table A.5: Statistics of the grids created for the five-dimensional Gaussian data set (1×10^6 data points, 5 dimensions) using surplus-based refinement, starting with a level two regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	1471	1471	126	100.00 %	11.67	16	16	16	16
1	2293	5375	248	61.16 %	9.25	16	16	16	32
2	3451	16 127	418	42.17 %	8.26	18	32	10	64
3	5216	42 047	640	31.61 %	8.15	18	32	13	128
4	7548	105 343	941	23.90 %	8.02	18	32	11	256
5	10 152	263 551	1370	17.77 %	7.41	19	32	6	512
6	13 581	577 919	1854	14.12 %	7.33	22	64	5	1024
7	17 787	1 054 591	2290	12.32 %	7.77	23	64	6	2048
8	22 209	1 855 487	2711	11.07 %	8.19	27	64	6	4096
9	28 247	3 790 847	3378	9.39 %	8.36	29	128	5	8192
10	35 999	7 545 599	4162	8.11 %	8.65	31	128	3	16 384
11	43 868	14 054 143	4955	7.10 %	8.85	31	128	4	32 768
12	50 925	24 980 735	5684	6.44 %	8.96	34	128	3	65 536
13	58 820	44 558 335	6609	5.75 %	8.90	36	128	3	131 072
14	71 099	76 308 479	7639	5.12 %	9.31	37	128	2	262 144
15	86 512	147 477 759	8911	4.54 %	9.71	38	128	4	524 288
16	100 577	195 752 703	9619	4.32 %	10.46	39	128	4	1 048 576
17	114 411	305 725 183	10 534	4.05 %	10.86	39	128	2	2 097 152
18	126 943	355 229 439	10 995	3.97 %	11.55	40	128	2	2 097 152
19	139 465	361 625 343	11 197	3.97 %	12.46	41	128	2	2 097 152
20	155 516	461 233 151	12 206	3.70 %	12.74	41	128	3	2 097 152

Table A.6: Statistics of the grids created for the five-dimensional Gaussian data set (1×10^6 data points, 5 dimensions) using surplus-based refinement, starting with a level five regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	18 943	18 943	462	100.00 %	41.00	64	64	64	64
1	19 825	41 599	639	73.38 %	31.03	64	64	14	128
2	21 501	103 551	931	51.44 %	23.09	64	64	12	256
3	23 957	268 159	1382	35.51 %	17.34	64	64	7	512
4	27 506	578 175	1850	27.28 %	14.87	64	64	7	1024
5	31 988	1 084 671	2278	22.80 %	14.04	64	64	8	2048
6	37 308	2 055 167	2800	19.03 %	13.32	64	64	5	4096
7	44 440	4 344 319	3566	15.34 %	12.46	64	64	6	8192
8	51 956	7 996 159	4290	13.07 %	12.11	64	64	4	16 384
9	62 001	15 101 695	5186	11.06 %	11.96	64	64	4	32 768
10	73 031	29 947 647	6253	9.35 %	11.68	64	64	4	65 536

Table A.7: Statistics of the grids created for the five-dimensional Gaussian data set (1×10^6 data points, 5 dimensions) using surplus-based refinement, starting with a level seven regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	21	21	11	100.00 %	1.91	2	2	2	2
1	241	241	66	100.00 %	3.65	4	4	4	4
2	1400	1961	281	75.04 %	4.98	8	8	8	8
3	3144	8961	720	46.88 %	4.37	8	8	8	16
4	5901	28 233	1414	33.50 %	4.17	11	16	10	32
5	9955	73 665	2385	26.03 %	4.17	12	16	5	64
6	16 653	174 337	3915	21.09 %	4.25	13	16	6	128
7	24 332	323 681	5556	18.63 %	4.38	14	16	3	256
8	33 607	625 601	7895	15.54 %	4.26	16	32	3	512
9	39 695	898 593	9283	14.43 %	4.28	16	32	2	1024
10	48 247	1 467 809	11 402	12.93 %	4.23	18	32	2	2048
11	59 973	2 026 657	13 897	11.85 %	4.32	18	32	2	2048
12	84 619	4 907 745	20 693	9.26 %	4.09	18	32	3	4096
13	106 876	8 229 569	26 989	7.88 %	3.96	20	64	2	8192
14	134 276	15 454 977	35 207	6.57 %	3.81	21	32	2	16 384
15	178 488	28 432 193	46 609	5.55 %	3.83	22	32	2	32 768
16	218 257	36 616 065	54 766	5.21 %	3.99	23	32	2	32 768
17	267 937	50 833 537	66 118	4.74 %	4.05	23	32	2	32 768
18	295 293	61 014 465	72 439	4.54 %	4.08	25	32	2	32 768
19	354 077	92 111 553	87 670	4.03 %	4.04	25	32	2	65 536
20	401 790	130 813 505	97 811	3.85 %	4.11	25	32	2	131 072

Table A.8: Statistics of the grids created for the ten-dimensional Gaussian data set (1×10^6 data points, 5 dimensions) using surplus-based refinement, starting with a level two regular sparse grid (refinement step zero).

	# Grid Points		Subgrids			Largest Subgrid by Used		Largest Subgrid by Max.	
Ref.	Used	Max.	# of	Avg. Utilization	Avg. # Pt. per	# Pt. Used	# Pt. Max.	# Pt. Used	# Pt. Max.
0	13 441	13 441	1001	100.00 %	13.43	16	16	16	16
1	15 342	26 561	1411	75.15 %	10.87	16	16	16	32
2	18 377	59 873	2000	56.48 %	9.19	18	32	12	64
3	22 504	129 441	2957	41.10 %	7.61	18	32	6	128
4	32 691	416 449	5910	23.64 %	5.53	18	32	4	256
5	48 139	1 128 801	10 432	15.56 %	4.61	19	32	4	512
6	70 491	2 615 777	16 142	11.64 %	4.37	22	64	4	1024
7	99 655	5 306 657	23 241	9.39 %	4.29	23	64	4	2048
8	133 776	10 238 145	31 866	7.75 %	4.20	24	32	3	4096
9	166 287	15 918 881	39 577	6.85 %	4.20	25	64	2	8192
10	207 515	25 666 401	49 116	6.08 %	4.22	26	64	2	16 384

Table A.9: Statistics of the grids created for the ten-dimensional Gaussian data set (1×10^6 data points, 5 dimensions) using surplus-based refinement, starting with a level five regular sparse grid (refinement step zero).

Bibliography

[AAA+07] J. K. Adelman-McCarthy et al. “The Fifth Data Release of the Sloan Digital Sky Survey.” In: *The Astrophysical Journal Supplement Series* 172.2 (2007), p. 634 (cit. on p. 52).

[AMD17] AMD. *”Vega” Instruction Set Architecture*. Tech. rep. Advanced Micro Devices, Inc., July 28, 2017 (cit. on p. 25).

[Bel61] R. E. Bellman. *Adaptive Control Processes - A Guided Tour*. Princeton, New Jersey, U.S.A.: Princeton University Press, 1961, p. 255 (cit. on p. 29).

[BG04] H.-J. Bungartz, M. Griebel. “Sparse grids.” In: *Acta Numerica* 13 (2004), pp. 147–269 (cit. on pp. 24, 29, 31, 32).

[BPZ08] H.-J. Bungartz, D. Pflüger, S. Zimmer. “Adaptive Sparse Grid Techniques for Data Mining.” In: *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the Third International Conference on High Performance Scientific Computing, March 6–10, 2006, Hanoi, Vietnam*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 121–130 (cit. on p. 29).

[BSW14] P. Baldi, P. Sadowski, D. Whiteson. “Searching for Exotic Particles in High-Energy Physics with Deep Learning.” In: *Nature Communications* 5 (2014), p. 4308 (cit. on p. 53).

[Bus15] G. Buse. “Exploiting Many-Core Architectures for Dimensionally Adaptive Sparse Grids.” Dissertation. München: Institut für Informatik, Technische Universität München, May 2015 (cit. on pp. 29, 34).

[Hei14] A. Heinecke. “Boosting Scientific Computing Applications through Leveraging Data Parallel Architectures.” PhD thesis. Technical University Munich, 2014 (cit. on pp. 29, 42).

[HP13] A. Heinecke, D. Pflüger. “Emerging Architectures Enable to Boost Massively Parallel Data Mining using Adaptive Sparse Grids.” In: *International Journal of Parallel Programming* 41.3 (June 2013), pp. 357–399 (cit. on pp. 23, 40, 42, 52, 54, 129).

[HS52] M. R. Hestenes, E. Stiefel. “Methods of Conjugate Gradients for Solving Linear Systems.” In: *Journal of Research of the National Bureau of Standards* 49.6 (Dec. 1952), pp. 409–436 (cit. on p. 36).

[NBGS08] J. Nickolls, I. Buck, M. Garland, K. Skadron. “Scalable Parallel Programming with CUDA.” In: *Queue* 6.2 (Mar. 2008), pp. 40–53 (cit. on pp. 25, 27).

[NVI16] NVIDIA. *NVIDIA Tesla P100*. Tech. rep. WP-08019-001_v01.1. NVIDIA, May 9, 2016 (cit. on pp. 25, 28, 56, 96, 101).

[NVI17] NVIDIA. *CUDA C Programming Guide*. Tech. rep. PG-02829-001_v9.0. NVIDIA, Sept. 22, 2017 (cit. on pp. 26, 27, 101).

[Pfl10] D. Pflüger. “Spatially Adaptive Sparse Grids for High-Dimensional Problems.” Dissertation. München: Institut für Informatik, Technische Universität München, Feb. 2010 (cit. on pp. 23, 24, 29, 31, 35, 36, 53).

[PHP16] D. Pfander, A. Heinecke, D. Pflüger. “A New Subspace-Based Algorithm for Efficient Spatially Adaptive Sparse Grid Regression, Classification and Multi-evaluation.” In: *Sparse Grids and Applications - Stuttgart 2014*. Cham: Springer International Publishing, 2016, pp. 221–246 (cit. on pp. 23, 36, 39, 42, 43, 45, 52, 54, 65).

[SGS10] J. E. Stone, D. Gohara, G. Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.” In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73 (cit. on p. 25).

[She94] J. R. Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. Pittsburgh, PA, USA: Department of Computer Science, Carnegie-Mellon University, 1994 (cit. on p. 36).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature