Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Vergleich von Parallel STL Implementierungen für GPUs mit nativen Frameworks

Felix Heidrich

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dirk Pflüger
Betreuer/in:	Marcel Breyer
Beginn am:	5. April 2022
Beendet am:	5. Oktober 2022

Kurzfassung

Heutzutage werden neue Rechner und vor allem auch Server (Cluster) mit *Graphics Processing Units* (GPUs) ausgerüstet, um Laufzeiten so gering wie möglich zu halten. Probleme, die besonders viel Leistung von einem Rechner oder Cluster beanspruchen, können meistens auf einer GPU parallelisiert werden und dadurch die Dauer der Ausführung verringert werden. Um die hohe Rechenleistung und die damit verknüpfte Laufzeitverbesserung der GPUs jedoch nutzen zu können, bedarf es einer der vielen verschiedenen Frameworks, wie *Compute Unified Device Architecture* (CUDA), OpenCL oder SYCL, was einen Anfänger leicht überfordern kann. Verglichen mit dem Programmieren für eine *Central Processing Unit* (CPU) unter C++, ist bei diesen Frameworks die Hürde für Anfänger sehr viel höher, da man hier sehr viel mehr beachten muss. Programmierer, die sich bereits gut mit der Sprache C++ auskennen, kennen meist auch die Algorithmen der *Standard Library* (STL). Diese Bibliothek unterstützt seit dem C++17 Standard die Parallele Ausführung von Algorithmen durch Exekutoren. Der Grafikkartenhersteller NVIDIA wollte die *Parallel Standard Library* (Parallel STL) auch auf seinen GPUs ausführen können und führte Mitte 2020 den `nvc++` Compiler in seinem *NVIDIA High Performance Computing Software Development Kit* (NVHPC SDK) ein. Dieses *Software Development Kit* (SDK) kann Code durch den `nvc++` Compiler, der die Parallel STL verwendet, für die Nutzung auf GPUs kompilieren. Des Weiteren kann `nvc++` Code auch für CPUs kompilieren, wodurch dieser ebenfalls parallel auf CPUs ausgeführt werden kann. Intel folgte am Ende desselben Jahres mit dem *Intel oneAPI DPC++ Compiler*, der in Version 2021.1.2 des *Intel C++ Compiler Classic* Compiler-Pakets zum ersten Mal enthalten war. Dieser kann GPUs mittels SYCL ansprechen.
Inwiefern dieser neue Ansatz, der parallelen Programmierung für Grafikkarten, besser ist als das native Framework CUDA für NVIDIA GPUs, soll in dieser Arbeit herausgefunden werden. Für den Vergleich der beiden Ansätze werden ausgewählte Algorithmen und Probleme sowohl in CUDA als auch in der Parallel STL implementiert und anschließend deren Laufzeiten analysiert. Ferner wird auch auf die Codelänge der Beispiele eingegangen, da diese sich drastisch zwischen den beiden Frameworks unterscheidet und kürzerer Quellcode meist leichter zu verstehen ist. Die Laufzeitmessungen auf zwei Verbraucher- und zwei Datencenter-GPUs haben dabei ergeben, dass die Parallel STL durchschnittlich fast 50 % schneller als CUDA ist. Die Implementierungen wurden sehr einfach gehalten und kaum optimiert, damit der Vergleich nicht beeinflusst wird. Wenn man den CUDA Code mittels *Unified Shared Memory* (USM) optimiert, ist CUDA 7 % schneller als die Parallel STL. Es wurden auch Beispiele gefunden, bei denen die Parallel STL Implementierung 3 bis mehr als 1 000 mal langsamer als CUDA ist. Dies tritt vor allem dann auf, wenn sich der CUDA Code leicht optimieren lässt oder der CUDA Code nicht komplett durch Funktionen der Parallel STL abbilden lässt. Der Quellcode der Parallel STL Implementierungen war im Schnitt 43,23 % kürzer als der entsprechende CUDA Code.
Sämtliche Implementierungen sind unter folgendem Link im öffentlichen GitHub Repository dieser Bachelorarbeit zu finden: https://github.com/heidrifx/cuda-stdpar-samples. Dabei zeigt der Tag v0.1 den Stand, mit dem der Vergleich durchgeführt wurde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele

Abkürzungsverzeichnis

BLAS Basic Linear Algebra Subprograms 31

CPU Central Processing Unit 3, 15, 19

CUDA Compute Unified Device Architecture 3, 15, 19

DPC++ Data Parallel C++ 66

FPGA Field-Programmable Gate Array 15, 26

GPU Graphics Processing Unit 3, 15, 19

HPC High Performance Computing 20

Intel TBB Intel Thread Building Blocks 26

JIT Just-In-Time 20

LLVM Low Level Virtual Machine 20

MSVC Microsoft Visual C++ 26

Parallel STL Parallel Standard Library 3, 15, 19

PTX Parallel Thread Execution 20

SDK Software Development Kit 3

SLOC Source Lines Of Code 59

SM Streaming Multiprocessor 19

STL Standard Library 3, 15, 25

USM Unified Shared Memory 3, 15, 27

1. Einleitung

In der heutigen Zeit werden neue Rechner und vor allem auch Server (Cluster) mit Hardware-Beschleunigern, wie *Graphics Processing Units* (GPUs) oder *Field-Programmable Gate Arrays* (FPGAs) ausgerüstet. GPUs bieten einen deutlich höheren Durchsatz an Berechnungen, da ihre Kerne zwar weniger komplexe Aufgaben, als die einer *Central Processing Unit* (CPU) ausführen können, dafür aber weit mehr Kerne als die meisten CPUs haben. Dadurch steigt vor allem bei sehr rechenintensiven Problemen, die auf einer GPU mit ihren vielen Kernen besser parallelisiert werden können als auf CPUs, die Rechenleistung. Dies begünstigt die Laufzeit, die man möglichst gering halten will.

Ins Besondere die Nutzung der hohen Rechenleistung von GPUs sollte möglichst einfach gehalten werden, damit auch Laien Zugriff auf die immense Performanceverbesserung im Vergleich mit einer CPU erhalten und keine große Einstiegshürde diese an der Verwendung hindert. Heutzutage machen es die vielen, aktuell verwendeten Frameworks, wie CUDA, OpenCL oder SYCL, jedoch sehr schwer einen Einstieg in die GPU-Programmierung zu finden, da ein Anfänger leicht durch die vielen Marker und Direktiven, sowie die zusätzliche Speicherverwaltung auf den GPUs überfordert ist. Insbesondere da das Framework meist zusätzlich vom Grafikkartenhersteller vorgeschrieben ist und somit vom Hersteller abhängt welches Framework verwendet werden muss – so z. B. CUDA bei NVIDIA.

Im Gegensatz zu diesen auf GPUs spezialisierten Frameworks, sind die meisten C++ Programmierer sehr gut mit der *Standard Library* (STL) vertraut, die es den Nutzern seit dem C++17 Standard ermöglicht Algorithmen mittels Exekutoren parallel auszuführen. Mitte 2020 hat der Grafikkartenhersteller NVIDIA den neuen `nvc++` Compiler in sein *NVIDIA High Performance Computing Software Development Kit* (NVHPC SDK) aufgenommen. Dieser Compiler ermöglicht es die parallelen Algorithmen des C++17 Standards auf GPUs auszuführen. Ferner kann `nvc++` denselben Code auch für die parallele Ausführung auf CPUs kompilieren.

In dieser Ausarbeitung soll ein Vergleich zwischen dem *Compute Unified Device Architecture* (CUDA) Framework und den parallelen Algorithmen der *Parallel Standard Library* (Parallel STL) mittels des NVHPC SDK gezogen werden. Dafür werden ausgewählte Algorithmen und Probleme sowohl unter CUDA als auch unter der Parallel STL ohne wirklich größere Optimierungen implementiert, um zu verhindern, dass eine der beiden Implementierungen begünstigt wird. Dennoch wurde ein Beispiel unter CUDA optimiert, um die Vorteile einer Optimierung aufzuzeigen.

Anschließend werden die Laufzeiten und die Codelänge verglichen und analysiert. Dabei ergab die Auswertung der Laufzeiten eine Verbesserung der Performance von knapp 50 % der Parallel STL gegenüber CUDA. Jedoch ist zu beachten, dass dieser Wert nur für diejenigen Implementierung gültig ist, für die sich der CUDA Code eins zu eins in Parallel STL übertragen ließ. Wenn man *Unified Shared Memory* (USM) für die CUDA Implementierung verwendet, ist CUDA ungefähr 7 % schneller als die Parallel STL. Im Fall der Algorithmen, die sich nicht komplett durch Funktionen der Parallel STL ersetzen ließen, ist die Parallel STL bis zu mehr als 1 000 mal langsamer. Das

optimierte CUDA Beispiel ist rund 3 mal schneller als die Parallel STL Implementierung.
Die Quellcodelänge ist auffallend, da diese von CUDA zur Parallel STL Implementierung stark abweichen kann. Die Analyse der Codelänge, der für diese Arbeit verwendeten Algorithmen und Probleme, hat ergeben, dass die Parallel STL durchschnittlich 42,23 % weniger Codezeilen als CUDA vorweist.

Die weiteren Teile dieser Ausarbeitung setzen sich wie folgt zusammen: Kapitel 2 präsentiert den Stand der aktuellen Forschung zum Thema der Arbeit, sowohl aus wissenschaftlicher als auch industrieller Sicht, da die Umsetzung der Parallel STL ein relativ neues Feld ist und mehr Informationen vonseiten der Industrie gefunden wurden. In Kapitel 3 werden grundsätzliche Konzepte und mathematische Probleme vorgestellt, die zum Verständnis der restlichen Arbeit notwendig sind. Kapitel 4 geht auf die ausgewählten Algorithmen und Probleme ein und wie diese für den Vergleich implementiert wurden. Ferner wird auf die Art und Weise der Laufzeitmessungen für das folgende Kapitel eingegangen. Der eigentliche Vergleich der Laufzeiten und Codelängen erfolgt dann in Kapitel 5. In diesem Kapitel wird der Versuchsaufbau mit der verwendeten Hardware und Compilern präsentiert. Abschließend fasst Kapitel 6 die Ausarbeitung zusammen. Kapitel 7 soll einen Ausblick für eine mögliche Fortsetzung dieser Arbeit liefern.

2. Verwandte Arbeiten

Im Rahmen dieser Ausarbeitung werden verschiedene Beispielimplementierungen unter CUDA und der Parallel STL angefertigt und deren Laufzeiten miteinander verglichen. Das Ziel der Arbeit ist es die Parallel STL mit dem nativen CUDA Framework zu vergleichen, da die Parallel STL noch relativ neu ist und nach derzeitigem Kenntnisstand keine vorhandene Forschungsarbeit existiert, die solch einen Vergleich anstrebt. Es wurde lediglich eine Arbeit gefunden, in der die *Lattice-Boltzmann*-Methode in der Parallel STL umgesetzt wurde [LMS+]. Ferner wurden technische Blogs von NVIDIA gefunden: Olsen et al. übersetzen die *LULESH* Hydrodynamik Mini-Applikation, die bereits unter anderem in CUDA zur Verfügung steht, in die Parallel STL und vergleichen die Performance anschließend mit einer Ausführung auf einer CPU und unter *OpenACC* [OLL20]. Singal implementiert *saxpy* ($a \cdot \vec{x} + \vec{y}$ mit Fließkommazahlen in einfacher Präzision) unter anderem auch in Parallel STL und vergleicht diese mit anderen Implementierungen in verschiedenen Frameworks wie bspw. CUDA oder *Thrust* [Sin21].

In einer Präsentation beschreiben Latt et al. zuerst die Vorteile und Besonderheiten der Parallel STL und anschließend ihr Vorgehen beim Überführen von *Palabos* [LMK+20] in die Parallel STL [LMS+]. Der *Palabos* (Parallel Lattice Boltzmann Solver) ist eine seit 2010 entwickelte Softwarebibliothek, mit der *Lattice-Boltzmann*-Schemata (*lattice*, zu dt. *Gitter*) modelliert werden können. Ferner dient er als Referenzimplementierung für eine Großzahl existierender *Lattice-Boltzmann*-Modelle.
Die Präsentation verdeutlicht, dass die Übersetzung in die Parallel STL nicht so simpel ist, wie sie scheint und Änderungen auf einer höheren Ebene benötigen, um den objektorientierten Code parallel auf einer GPU verwenden zu können. Wie generell für GPU Code zu empfehlen ist, mussten Latt et al. das Speicherlayout eines *Array-of-structure*, bei dem jedes Datenobjekt mit seinen Attributen nacheinander im Speicher liegt, in ein *Structure-of-array*, bei dem alle Attribute nacheinander im Speicher liegen, umwandeln, damit der Code effizienter auf der GPU ausgeführt werden kann. Virtuelle Funktionen (engl. *virtual functions*) im Datenobjekt selber können nicht verwendet werden, da der Mechanismus, der Funktionszeiger aufruft, unter Parallel STL nicht unterstützt wird. In ihrer *3D Taylor-Green vortex* Benchmark zeigen Latt et al., dass ihre Parallel STL Implementierung 21 mal schneller auf der *RTX 3090* GPU wie auf der 48-Kerne *Xeon Gold* CPU ist. Jedoch ist die Implementierung immer noch 38 % von der Spitzenperformance entfernt – an diesen Wert könnte man sich, so Latt et al., durch einen einfacheren Code, ohne die Einschränkungen von *Palabos*, oder durch optimierten CUDA Code um ungefähr 20 % annähern. Weitere Tests haben eine Beschleunigung, verglichen mit einer Ausführung auf der CPU, um einen Faktor von 9,5 bei *flow through porous media* (dt. *Strömung durch poröses Medium*) und 6 bei *3D Rayleigh-Taylor instability* (dt. *3D Rayleigh-Taylor Instabilität*) ergeben. Beide Tests wurde erneut mit einer *RTX 3090* GPU und einer 48-Kern *Xeon Gold* CPU durchgeführt.
Des Weiteren führen sie *STLBM* [LCB21] als Beispiel einer komplett in Parallel STL geschriebenen Bibliothek auf.

In einem technischen Blog führen Olsen et al. *LULESH* [KKN13] als großes Beispiel auf, das von ihnen in die Parallel STL überführt wurde [OLL20]. *LULESH* steht für *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics* und ist eine sehr stark vereinfachte Applikation, die nur ein *Sedov blast problem* löst. *LULESH* habe sehr viele Schleifen, die sich leicht durch ein `std::for_each_n` aus der STL austauschen ließen. Der Schleifenkörper wird dann zum Körper des Lambdaausdrucks. An einer anderen Stelle wird ein 52 Zeilen lange Code mittels `std::transform_reduce` auf 12 Zeilen reduziert, da hier das temporäre Array wegfällt.
In einem Vergleich haben Olsen et al. [OLL20] gezeigt, dass die Performance der Parallel STL auf demselben Niveau wie die ursprüngliche OpenACC Implementierung ist – beide sind auf einer *A100* GPU knapp 7 mal performanter als eine Ausführung auf einer 40-Kern Dual-Socket *Xeon Gold* CPU.

Searles zeigt in einer Präsentation erneut *LULESH* als Beispiel, führt jedoch zusätzlich zur Beschleunigung der Laufzeit um den Faktor 13,57 bei der Parallel STL auf einer GPU durch `nvc++` auch die verbesserte Leistung der Parallel STL auf einer CPU durch `nvc++` (2,08 mal schneller) im Vergleich zum `gcc` Compiler (nur 1,53 mal schneller) auf [Sea22]. Ferner wird eine neue Benchmark namens *M-AIA* erwähnt, bei der ein in Parallel STL übersetzter Code auf vier *A100* GPUs 8,74 mal schneller als der *OpenMP* Code auf zwei *EPYC 7742* CPUs ist. Auch hier wird *STLBM* als Beispiel einer Implementierung auf mehreren Plattformen, einschließlich Multi-Core-CPUs und GPUs, die ohne Spracherweiterungen, externe Bibliotheken, Hersteller-spezifischen Markern oder zusätzlicher Schritte vor dem Kompilieren auskommt, präsentiert. Für *STLBM* ist eine *A100 PCIE 40GB* mit der Parallel STL unter GPU etwas mehr als 12 mal schneller als eine 2-Socket *Xeon 6148* mit 40 Kernen, die denselben Code der Parallel STL unter CPU ausführt.

Singal beschreibt in einem technischen Blog insgesamt fünf verschiedene Implementierungen einer *saxpy* Berechnung [Sin21]. Darunter sind neben *cuBLAS*, *OpenACC* und *Thrust* auch CUDA und die Parallel STL. Im Vergleich schneidet die *cuBLAS* Implementierung am besten ab, während die anderen vier alle gleich auf sind. Allesamt sind sie mehr als 20 mal schneller als die Parallel STL Implementierung, die auf einer CPU ausgeführt wurde. Singal verwendete hierbei zwei AMD *EPYC 7742* CPUs mit insgesamt 128 Kernen und 256 Threads und eine einzige *A100* für seinen Vergleich.

Die Arbeiten von Latt et al. [LMS+], Olsen et al. [OLL20] und Searles [Sea22] implementieren jeweils Projekte, für die teilweise bereits Code von anderen Frameworks für Hardware-Beschleuniger vorhanden ist, vergleichen aber die Performance ihres Resultats nur mit einer Ausführung auf einer CPU, nicht aber mit den anderen Frameworks. Singal [Sin21] hingegen vergleicht die einzelnen Frameworks untereinander. Diese Ausarbeitung soll nun an der Arbeit von Singal anknüpfen, da seine Arbeit lediglich verschiedene *saxpy* Implementierungen vergleicht und sonst keine anderen Algorithmen oder Probleme. Im Rahmen dieser Ausarbeitung wurden CUDA, da es das native Framework für NVIDIA GPUs ist, und die Parallel STL Implementierung von NVIDIA durch NVHPC, da diese noch relativ neu ist und für C++ Programmierer mit STL Kenntnis leichter zu verwenden ist, als Frameworks für einen Vergleich ausgewählt. Die *saxpy* Implementierungen von Singal zeigten, dass CUDA und die Parallel STL eine ähnliche Performance aufweisen, dies soll durch weitere Algorithmen und Probleme in dieser Arbeit belegt werden.

3. Grundlagen

Dieses Kapitel dient dazu grundsätzliche Konzepte und mathematische Probleme zu beschreiben, die notwendig für das Verständnis der folgenden Ausarbeitung sind.
Dabei wird zuerst die momentane Relevanz der *Graphics Processing Unit* (GPU) (Abschnitt 3.1) erläutert und deren Aufbau. Anschließend werden zwei Frameworks vorgestellt, mit denen man Programmcode für GPUs schreiben kann: Die weit verbreitete *Compute Unified Device Architecture* (CUDA) *API* (Abschnitt 3.2) und die neuere *Parallel Standard Library* (Parallel STL) (Abschnitt 3.3) Implementierung. Ferner stellt Abschnitt 3.4 die *Mandelbrot-Menge* vor, deren Grundkenntnisse für die Implementierung bedeutsam sind.

3.1. Die *Graphics Processing Unit* (GPU)

Graphics Processing Units stellen einen immer wichtigeren Aspekt der Parallelisierung und Beschleunigung von Algorithmen dar. In diesem Abschnitt soll der Grund der Wichtigkeit, sowie der Aufbau einer GPU konkretisiert werden.

Obwohl sich die Anzahl der Transistoren auf einem integrierten Schaltkreis nach dem mooreschen Gesetz [Moo06] alle zwei Jahre verdoppelt, stoßen wir in den letzten Jahren langsam an die Grenzen des physikalisch Möglichen. Die Transistoren sind mittlerweile so klein geworden, dass sie nicht länger kleiner gemacht werden können, ohne dass Elektronen überspringen könnten. Die Lösung ist es *Central Processing Units* (CPUs) mit mehreren Kernen zu entwickeln, die eine parallele Ausführung ermöglichen und dadurch die Leistung gemäß Moore weiterhin erhöhen. Dennoch reicht diese Geschwindigkeit meist nicht aus, wenn man mehrere Millionen oder Milliarden Berechnungen in einer sehr kurzen Zeit ausführen möchte – Abhilfe können GPUs schaffen. Im Gegensatz zur CPU, die neben dem Berechnen von arithmetischen und logischen Funktionen auch Bit-Verschiebungen durchführen kann, können diese GPUs zwar pro Kern immer nur Fließkomma- oder Ganzzahloperationen durchführen, haben dafür aber auch deutlich mehr Kerne. Heutzutage haben GPUs mehrere Tausend Kerne und ermöglichen trotz vergleichsweise langsamer Taktung mit nur 1,4 GHz-1,8 GHz, eine schnellere Alternative um große Mengen an Daten zu verarbeiten.

Zum Beispiel kann die 2017 erschienene NVIDIA GPU *GeForce GTX 1080 Ti* mit ihren 3584 CUDA Kernen und einer Basistaktung von ungefähr 1,5 GHz 11,34 TFLOPS (Billionen Fließkommaoperationen pro Sekunde, engl. *(tera) floating point operations per second* (FLOPS)) durchführen [1080-ti]. Die Zahl der CUDA Kerne ergibt sich aus der Anzahl der *Streaming Multiprocessors* (SMs) und der Zahl der CUDA Kerne pro SM: $28\ \text{SM} \cdot 128\ \frac{\text{Cores}}{\text{SM}} = 3584\ \text{Cores}$.
Die im selben Jahr von Intel erschienene CPU *i7-8700K* kann mit ihren 6 Kernen und einer Basistaktung von 4,7 GHz nur ungefähr 122 GFLOPS nachweisen. Dieser Wert wurde lokal mittels *Geekbench 4* [geekbench4] und der SGEMM CPU-Workload ermittelt. Beide Werte beziehen sich auf `float`, also Fließkommazahlen mit einfacher Präzision (engl. *single percision*). Die GPU ist

damit ungefähr 93-mal schneller.
Wenn man die theoretische Maximalleistung der CPU mittels

CPU Geschwindigkeit · Anzahl an Kernen · CPU Befehle pro Zyklus

berechnet, kommt man für die *i7-8700K* auf 710,4 GFLOPS. Dieser Wert ist beinahe 6-mal größer als der Wert der Benchmark, jedoch immer noch knapp 16-mal kleiner als der der GPU.
GPUs werden – wie der Name bereits sagt – häufig für die grafische Darstellung von beispielsweise Computerspielen auf einem Monitor verwendet, bei denen meist zwei Framebuffer in sehr schnellen Abständen vorbereitet und ausgetauscht werden müssen. Die Verwendung in *High Performance Computing* (HPC) Systemen für Simulationen oder andere Prozessor-intensive Berechnungen ist in den letzten Jahren immer weiter gestiegen.

3.2. Die *Compute Unified Device Architecture* (CUDA) API

Die CUDA API wurde 2006 von NVIDIA vorgestellt und am 15. Februar 2007 veröffentlicht. Die ersten GPUs, die CUDA unterstützen sind Teil der G80 Serie und basieren auf der *Tesla* Mikroarchitektur.
Die CUDA Entwicklungsplattform unterstützt Industriestandard-Programmiersprachen wie C, C++ und Fortran nativ, aber auch moderne Sprachen wie Python oder Java werden durch Bibliotheken von Drittanbieter unterstützt. Der C/C++ Code wird durch den eigens von NVIDIA entwickelten, auf *Low Level Virtual Machine* (LLVM) basierenden `nvcc` Compiler in den Zwischencode *Parallel Thread Execution* (PTX) übersetzt. Dieser wird dann von NVIDIA verwenden, um CUDA Code parallel auszuführen. Der PTX Zwischencode wird vor der Ausführung mit einem *Just-In-Time* (JIT) Compiler in den, der verwendeten Maschinenarchitektur entsprechenden, Maschinencode kompiliert.

Codebeispiel 3.1 Allokierung des *host* und *device* Eingabe- und Ausgabespeichers. CUDA *device* Zeiger müssen mit eigenen Methoden allokiert und Werte vom *host* auf das *device* kopiert werden. Anschließend kann der Kernel mit der Anzahl der *Blöcke* pro *Grid* und *Threads* pro *Block* aufgerufen werden. Das Ergebnis muss erneut kopiert werden. Abschließend wird der Speicher wieder freigegeben, auch hier müssen *device* Zeiger mit eigenen CUDA-Funktionen freigegeben werden.

```
1  // Allocate host input/output arrays
2  auto *h_A = (float *) malloc(size);
3  auto *h_B = (float *) malloc(size);
4  auto *h_C = (float *) malloc(size);
5
6  ... // Initialize input arrays
7
8  // Allocate device input arrays
9  float* d_A = nullptr, d_B = nullptr, d_C = nullptr;
10 cudaMalloc((void **) &d_A, size); cudaMalloc((void **) &d_B, size); cudaMalloc((void **) &d_C, size);
11
12 // Copy host arrays to device arrays
13 cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
14 cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
15
16 // Prepare grid size and block size
17 int threadsPerBlock = 256;
18 int blocksPerGrid = numElements / threadsPerBlock;
19 // Launch kernel
20 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
21
22 // Copy device array to host array
23 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
24
25 ... // Use result
26
27 // Free device memory
28 cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
29 // Free host memory
30 free(h_A); free(h_B); free(h_C);
```

CUDA unterscheidet grundsätzlich zwischen dem *host*, der den CPU-Code ausführt und dem *device*, das den GPU-Code, die als *Kernel* bezeichneten Methoden, ausführt. Zur Verwendung der CUDA API muss Speicher sowohl auf der CPU als auch auf der GPU angelegt werden.
Auf dem *host* funktioniert die Speicherverwaltung, wie üblich in C/C++, mit einem `malloc`, das Gleiche gilt jedoch nicht für das *device*. Hier benötigt man die CUDA eigene Methode `cudaMalloc` (siehe Zeile 10 in Codebeispiel 3.1).
Um Daten auf die GPU zu kopieren wird die `cudaMemcpy(target, source, size, cudaMemcpyHostToDevice)` Funktion (siehe Zeile 13f.) verwendet, deren letzter Parameter die Kopierrichtung angibt. Zu berücksichtigen ist, dass das Kopieren, verglichen mit der Ausführung, eine beträchtliche Zeit in Anspruch nehmen kann. Besonders stark tritt dies in Erscheinung, wenn der ausführende Code auf der GPU eine vergleichsweise einfache Berechnung, wie hier die Addition zweier Vektoren, berechnet. Beim Kopieren mehrere Werte kann die Dauer verringert werden,

indem man sie *asynchron* mittels `cudaMemcpyAsync` auf das *device* kopiert.
Außerdem kann ein *unified memory* angelegt werden, auf den sowohl CPU als auch GPU zugreifen können.

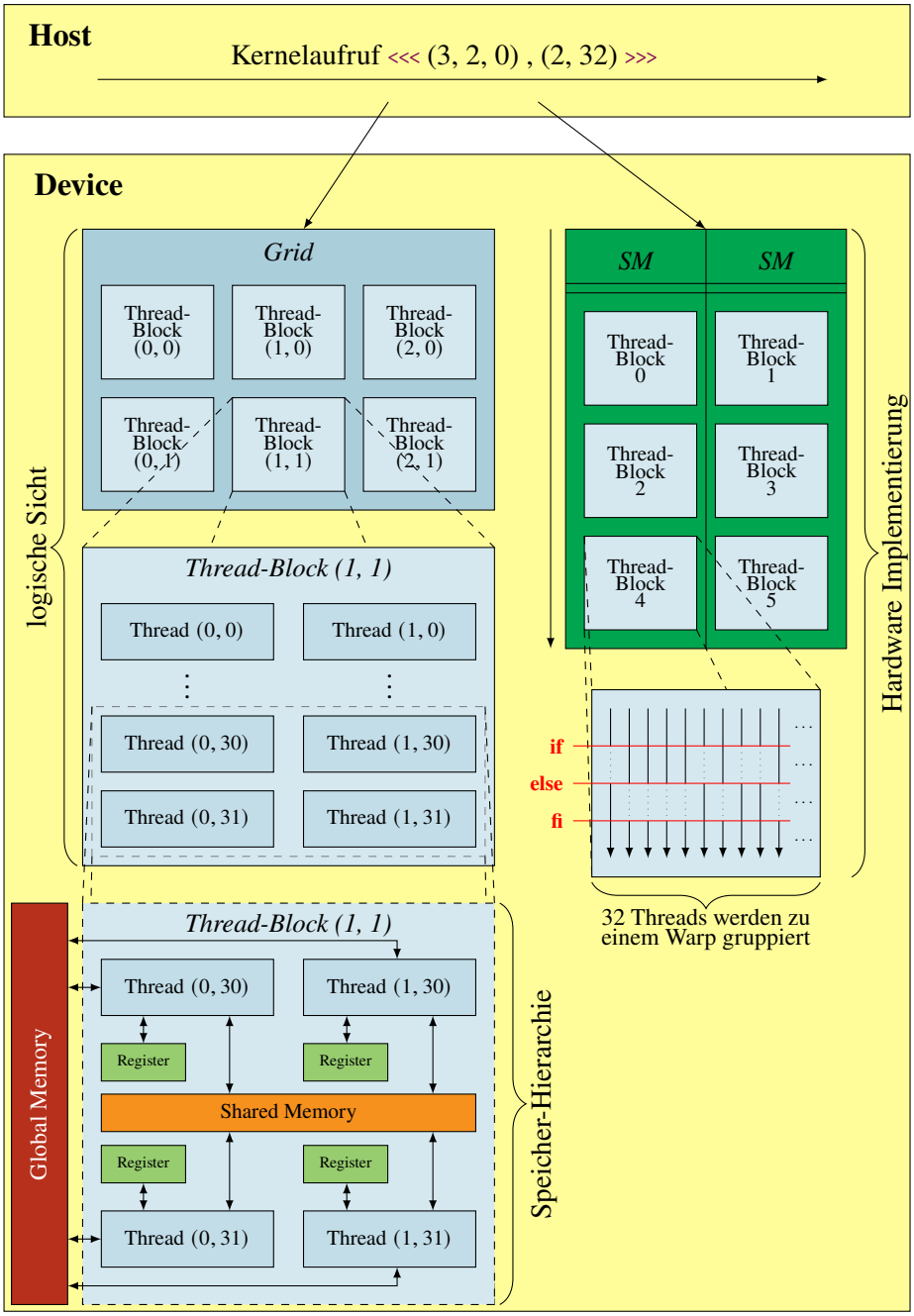

Abbildung 3.1.: Beim Aufruf eines CUDA Kernels werden die *Blöcke* in einem *Grid* organisiert. Bis zu 1024 *Threads* können einem Thread-*Block* angehören, üblicherweise sind es aber nur $16 * 16 = 256$ *Threads*. Jeder Thread hat Zugriff auf seinen eigenen lokalen Speicher, einem *Register*, den geteilten Speicher (engl. *shared memory*) und dem *globalen Speicher*. Alle Thread-*Blöcke* werden gleichmäßig auf die einzelnen SMs der GPU verteilt, wo sie in *Warps* zu je 32 *Threads* gruppiert sind. (Erstellt von Marcel Breyer [Bre18] und Alexander Van Craen [Cra18])

Die GPUs von NVIDIA, um die es in dieser Ausarbeitung hauptsächlich geht, sind grundsätzlich alle ähnlich aufgebaut: Auf der Hardware Seite (siehe *Hardware Implementierung*, rechts in Abbildung 3.1) bestehen sie aus mehreren *Streaming Multiprocessors* (SMs), die die CUDA Kerne beinhalten. Die Kerne können jeweils eine Fließkomma- oder Ganzahloperation pro Taktzyklus ausführen.
Aus logischer Sicht (links in Abbildung 3.1) ist der kleinste Teil eines CUDA Kernels ein *Thread*. CUDA C++ organisiert *Threads* dabei in folgender Hierarchie: Bis zu 1024 *Threads* können einem *Block* angehören und beliebig viele dieser *Blöcke* sind Teil eines *Grids* (engl. für *Gitter*). Üblicherweise besteht ein Block aus $16 * 16 = 256$ Threads. Thread-Blöcke werden wiederum in sogenannte *Warps* aufgeteilt, die jeweils 32 Threads beinhalten. Diese werden dann von einem *warp scheduler* zur Ausführung geplant (engl. *to schedule*). Mehrere Thread-Blöcke können gleichzeitig auf einem SM ausgeführt werden. Jedoch ist zu beachten, dass diese immer unabhängig voneinander sequentiell oder parallel ausgeführt werden können müssen, da bei der Ausführung schnellere SMs einen weiteren Block bearbeiten könnten.

Die Speicherverwaltung ist ebenfalls hierarchisch aufgeteilt und reicht vom lokalen Speicher, dem *Register*, der Threads über den *shared memory* der Blöcke bis hin zum *globalen Speicher* der Grids (siehe *Speicher-Hierarchie*, unten in Abbildung 3.1).
Der geteilte Speicher (engl. *shared memory*) der Threads innerhalb eines Blocks hat eine ähnlich niedrige Latenz wie ein Level-1 Cache einer CPU und ermöglicht so den schnellen Datenaustausch zwischen Threads. Wenn die Implementierung es zulässt, kann dadurch die Ausführung zusätzlich beschleunigt werden. Beim Arbeiten auf einem *shared memory* kann es notwendig sein, dass man die Ausführung pausieren will bis alle Threads an der gleichen Stelle angekommen sind. Dadurch verhindert man eine Wettlaufsituation (engl. *data races*), die immer dann auftreten kann, wenn ein Thread auf dieselbe Ressource zu greifen möchte wie ein anderer und dieser dann gegebenenfalls den Wert des anderen überschreibt oder den Wert des anderen gar nicht erst lesen kann, weil er zu schnell ist. Das gegenseitige Warten der Threads kann durch ein `__syncthreads()` erreicht werden. Der *shared memory* wird mittels `__shared__` markiert und kann dynamisch allokiert werden, indem man dem Aufruf des Kernels einen weiteren Parameter übergibt: `kernel<<<blocksPerGrid, threadsPerBlock, shared_mem_size>>>(...)`.

Codebeispiel 3.2 Auszug aus Codebeispiel 3.1 mit Kernel-Aufruf und Parametern.

```
16 // Prepare grid size and block size
17 int threadsPerBlock = 256;
18 int blocksPerGrid = numElements / threadsPerBlock;
19 // Launch kernel
20 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Mit der <<<. . .>>>-Syntax beim Aufruf des Kernel (Zeile 20 in Codebeispiel 3.2) wird zuerst die Anzahl der *Threads pro Block* (Zeile 17) und danach die *Blöcke pro Grid* (Zeile 18) festgelegt. Beide dieser Werte können entweder ein eindimensionaler `int` Wert oder ein dreidimensionaler `dim3` sein. Auch zweidimensionale Blöcke oder Grids werden durch `dim3 twoDimensional(x,y)` unterstützt.

Codebeispiel 3.3 CUDA Kernel Aufbau mit Kernel-spezifischen Variablen.

```
1 __global__ void vectorAdd(const float *A, const float *B, float *C, int numElements) {
2         int i = blockDim.x * blockIdx.x + threadIdx.x;
3
4         if (i < numElements)
5                 C[i] = A[i] + B[i];
6 }
```

Kernel werden mittels des *declaration specifiers* `__global__` (siehe Zeile 1 in Codebeispiel 3.3) als solche markieren. Das Grundprinzip hinter CUDA ist es, ein Problem in kleinere Unterprobleme aufzuteilen und diese dann parallel auf den CUDA Kernen auszuführen. Die Unterprobleme werden in Kernels realisiert und durch die parallele Ausführung kann das Problem meist schneller gelöst werden als eine rein sequentielle. Ein Kernel erhält Zugriff auf seinen Thread Index durch den dreidimensionalen Vektor `threadIdx` (siehe Zeile 2 in Codebeispiel 3.3). Ferner stehen `blockIdx` und `blockDim` (siehe ebenfalls Zeile 2 in Codebeispiel 3.3) zur Bestimmung des ausführenden Blocks und dessen Dimensionen zur Verfügung.

3.3. Die *Parallel Standard Library* (Parallel STL)

Bis vor zwei Jahren war es nur möglich C++ Code durch eine GPU zu beschleunigen in dem man Spracherweiterungen wie CUDA C++ oder OpenACC[1], Bibliotheken von Drittanbietern wie Thrust[2], Kokkos[3] oder Frameworks wie SYCL[4] und OpenCL[5] verwendet. Der zusätzliche Aufwand, Kernelcode zu schreiben oder Methoden mittels Pragmas und Markern zu versehen hat sich gelohnt, da die Ausführung auf der GPU deutlich schneller war als auf einer CPU.
Am 4. August 2022 veröffentlichte NVIDIA den `nvc++` Compiler in Verbindung mit dem NVIDIA HPC SDK (engl. *Software Development Kit*), im Weiteren abgekürzt als NVHPC SDK). Dieser Compiler ermöglichte es erstmals GPU-Beschleunigung ohne Spracherweiterungen oder Bibliotheken zu nutzen.

Im Folgenden werden die Parallelitätsmodelle der *C++ Standard Parallelisierung* (Abschnitt 3.3.1) vorgestellt und erklärt. Des Weiteren wird in Abschnitt 3.3.2 das NVHPC SDK genauer beschrieben und auf dessen Arbeitsweise, sowie einige Besonderheiten des `nvc++` Compilers eingegangen.

3.3.1. C++ Standard Parallelisierung

Bereits mit C++11 wurden ein Speichermodell, ein Nebenläufigkeitsmodell und -bibliothek vorgestellt, allerdings gab es noch keine Funktionen der parallelen Programmierung auf höherer Ebene. Erst mit dem C++17 Standard wurden Parallelitätsfunktionen auf höherer Ebene hinzugefügt, die es möglich machten Algorithmen der *Standard Library* (STL) parallel zu nutzen. Die meisten

[1] https://www.openacc.org/
[2] https://thrust.github.io/
[3] https://kokkos.org/ bzw. https://github.com/kokkos/kokkos
[4] https://www.khronos.org/sycl/
[5] https://www.khronos.org/opencl/

Algorithmen der STL wurden mit C++17 so angepasst, dass sie einen zusätzlichen Ausführungsrichtlinienparameter unterstützen und dadurch parallel ausführbar sind. Darunter auch die hilfreichen Algorithmen `std::for_each`, `std::transform` oder `std::reduce`. Die C++ Standard Parallelisierung (engl. *C++ standard parallelism*) wird im Folgenden mit *stdpar* abgekürzt.

Der Standard definiert die folgenden 4 Ausführungsrichtlinien:

- `std::execution::seq`: eine reine sequentielle Ausführung mit keinerlei Parallelisierung
- `std::execution::unseq`: eine vektorisierte Ausführung auf dem aufrufenden Thread (erst mit C++20 in den Standard aufgenommen)
- `std::execution::par`: eine parallele Ausführung auf einem oder mehreren Threads
- `std::execution::par_unseq`: eine parallele Ausführung auf einem oder mehreren Threads, wobei jeder Thread vektorisiert sein kann

Diese sind eigentlich vielmehr für CPUs ausgelegt, da eine GPU keine Threads vektorisieren kann. Es ist bereits möglich, die *stdpar* für CPU mittels GCC 9.1 und der *Intel Thread Building Blocks* (Intel TBB)[6] Bibliothek oder *Microsoft Visual C++* (MSVC)[7] zu verwenden. Die Intel TBB Bibliothek ist dabei hauptsächlich für Intel CPUs ausgelegt, unterstützt aber auch neuere AMD CPUs. Der MSVC Compiler ist nur unter Windows verfügbar. Ferner kann die Intel *oneAPI*[8] genutzt werden, die mittels *SYCL* verschiedene Zielplattformen wie GPUs oder auch *Field-Programmable Gate Arrays* (FPGAs) unterstützt.

Zu beachten ist, dass das Verwenden einer Richtlinie nur eine Präferenz äußert und keineswegs eine Garantie, dass dieser Code auch parallel ausgeführt wird. Wenn ein Compiler *nur* mit dem Standard konform sein will, kann dieser den Hinweis ignorieren. Eine hochwertige Implementierung tut dies für gewöhnlich jedoch nicht, beachtet den Hinweis und führt den Code, wenn ausreichend Ressourcen zur Verfügung stehen, parallel aus. Wenn nicht, dann wird der Code sequentiell ausgeführt.
`std::execution::par` und `std::execution::par_unseq` dürfen zudem über keine Wettlaufsituationen (engl. *data races*) verfügen, wenn sie nebenläufig und auf unterschiedlichen Threads ausgeführt werden. Ebenso dürfen `std::execution::unseq` und `std::execution::par_unseq` zusätzlich dazu, dass sie keine *data races* beinhalten dürfen, auch keine *deadlocks* haben, wenn mehrere Aufrufe auf demselben Thread verschachtelt werden, was bei der Vektorisierung einer Schleife geschieht.
Vektorisierung bedeutet, dass die Ausführung eines Schleifenkörpers parallelisiert werden kann, also mehrere Schleifeniterationen gleichzeitig ausgeführt werden können. Tritt währenddessen kein *deadlock* ein, so spricht man von einer *sicheren Vektorisierung* (engl. *vectorization-safe*). Ein *deadlock* kann immer dann entstehen, wenn zwei Prozesse voneinander abhängig sind und deren parallele Ausführung dazu führt, dass beide gegenseitig auf das Ende der Ausführung des jeweils anderen warten. Dadurch kommt es dann zu einer *Verklemmung*.

Parallele Algorithmen der STL, die für diese Ausarbeitung von Wichtigkeit sind, sind unter anderem:

[6] https://github.com/oneapi-src/oneTBB
[7] https://docs.microsoft.com/en-us/cpp/?view=msvc-170
[8] https://www.oneapi.io

- `std::for_each` und `std::for_each_n` sind nebenläufige Implementierungen einer `for`-Schleife, wobei bei letzterer der Schleifenkörper nur auf die ersten `n` Elemente angewandt wird.
- `std::transform` *transformiert* eine Sequenz, d. h. dass eine Funktion wird auf jedes Element angewendet.
- `std::reduce` *reduziert* die Werte einer Sequenz, im Normalfall (engl. *default case*) durch eine Addition. Die Elemente können – im Gegensatz zu `std::accumulate` – in beliebiger Reihenfolge gruppiert und umgeordnet werden. Die Funktion, die auf der Sequenz ausgeführt wird, kann selbst festgelegt werden.
- `std::transform_reduce` *reduziert* die Werte nach einer Transformation, erneut normalerweise durch eine Addition. Die Transformations- und Reduktionsfunktion lassen sich aber auch eigens definieren. Für zwei Eingabesequenzen, ohne angegebener Funktion wird das Skalarprodukt, also die Summe der Produkte der einzelnen Werte, berechnet.

3.3.2. NVHPC und der `nvc++` Compiler

Neben Analysewerkzeugen umfasst das NVHPC SDK auch verschiedene Bibliotheken und Compiler, die für das Entwickeln von HPC-Anwendungen von essenzieller Bedeutung sind. Das SDK unterstützt dabei die Programmiersprachen C++, Fortran, die OpenACC und OpenMP Direktiven und das CUDA Framework, die durch `nvc++`, `nvfortran`, `nvc` und `nvcc` compiliert werden können. Eingebundenen Bibliotheken sind unter anderem *Thrust*, *cuBLAS* (GPU-beschleunigte, grundlegende Unterprogramme der linearen Algebra, engl. *basic linear algebra subroutines*) oder *Open MPI*. Sowohl der CUDA Profiler *Nsight*, also auch der `cuda-gdb` Debugger sind Teil der Analysewerkzeuge. Somit ist eine komplette CUDA-Installation im NVHPC SDK enthalten.

`nvc++` kann unter Linux Code für NVIDIA GPUs, sowie x86-64, OpenPOWER und ARM CPUs zur parallelen Ausführung kompilieren. Damit ist es möglich dasselbe Programm für CPUs unterschiedlicher Architektur (genannt *cross-compile*), wie auch für GPU zu kompilieren. NVIDIA verwendet hierfür die *OpenMP*-Laufzeit über das *OpenMP*-Backend der *Thrust* Bibliothek [Ms22], die beide im NVHPC SDK eingebunden sind. Dies geschieht mittels der `-stdpar` Flag des Compilers: `-stdpar=gpu` für eine Ausführung auf einer GPU und `-stdpar=multicore` für eine CPU. Wenn kein Wert übergeben wird, verwendet der Compiler die GPU. Der Compiler unterstützt den C++17 Standard, *C++ Standard Parallism* (stdpar) und *OpenAcc* für CPU und GPU, sowie OpenMP für CPU.

Parallele Algorithmen der STL mit paralleler Ausführungsrichtlinie (`std::execution::par` oder `std::execution::par_unseq`) können dadurch auf einer GPU ausgeführt werden. Der `nvc++` Compiler überprüft den Aufrufsgraphen jeder Quelldatei auf Code, der für die GPU bestimmt ist und kompiliert diesen dementsprechend. Um die Speicherverwaltung so einfach wie möglich zu halten, verwendet `nvc++` *Unified Shared Memory* (USM), also Speicher dessen Adressbereich, sich CPU und GPU teilen. Dadurch wird das Allokieren von Speicher auf der GPU und das Kopieren von *host* nach *device* eingespart. Dynamisch allokierter Speicher innerhalb GPU Code ist aber dennoch nur für die GPU sichtbar.
Dabei ist immer zu beachten, dass ein Pointer immer auch auf ein Objekt im dynamischen Speicher der CPU zeigen muss. Beim Dereferenzieren von Pointern auf dem CPU Stack oder dem globalen Speicher, kommt es sonst zu einer Speicherverletzung im GPU Code. Deshalb kann ein `std::vector`,

der auf dem CPU Heap liegt, verwendet werden, ein `std::array`, dessen Inhalt im Stapel liegt, jedoch nicht. Insbesondere ist es wichtig bei Lambdaausdrücken darauf zu achten, ob man *capture by reference* oder *capture by value* verwendet (Codebeispiel 3.4).

Codebeispiel 3.4 Zwei saxpy Implementierungen, wobei die erste den Wert von `a` als Referenz übergibt und die zweite den Wert kopiert. Lambdaausdrücke mit *capture by reference* führen zu einer Speicherverletzung. Verwendet man *capture by value* wird der Wert auf die GPU kopiert und muss dort nicht mehr dereferenziert werden. Der ursprüngliche Code stammt aus dem technischen Blog *Accelerating Standard C++ with GPUs Using stdpar* [OLL20] und ist im Kapitel *C++ Parallel Algorithms and CUDA Unified Memory* zu finden.

```
1 void saxpy_reference(float* x, float* y, int N, float a) {
2         std::transform(std::execution::par_unseq, x, x + N, y, y,
3         [&](float xi, float yi){ return a * xi + yi; }); // capture by reference -> memory violation
4 }
5
6 void saxpy_value(float* x, float* y, int N, float a) {
7         std::transform(std::execution::par_unseq, x, x + N, y, y,
8         [=](float xi, float yi){ return a * xi + yi; }); // capture by value
9 }
```

Die Implementierungen der `<cmath>` Bibliothek von `nvc++` nutzen dieselbe Implementierung wie auch CUDA C++. Der, vor allem für das Debuggen nützliche, `printf` Befehl wird ebenfalls, wie auch in CUDA C++ seit der G100 Serie basierend auf der *Fermi* Architektur, innerhalb von GPU Code unterstützt.

3.4. Die Mandelbrot-Menge

Die Mandelbrot-Menge beschreibt die Menge der komplexen Zahlen $c \in \mathbb{C}$, für die die rekursive Folge $z_{n+1} = z_n^2 + c$ mit dem Anfangswert $z_0 = 0$ beschränkt ist, d. h. dass sie entweder gegen einen Fixpunkt konvergieren oder in einen Zyklus übergehen. Im Folgenden wird die Anzahl der Iterationen auf 10 000 beschränkt

Wenn man die Mandelbrot-Menge geometrisch darstellt, bildet sie ein Fraktal (siehe Abbildung 3.2). Für die Berechnung des Bildes weist man jedem Pixel eine komplexe Zahl c zu und überprüft, ob die Folge endlich bleibt. Der Grauwert des Pixels beschreibt dann die Anzahl der Iterationen, die benötigt wurden, um auszuschließen, dass c nicht mehr der Mandelbrot-Menge angehört.

Das Erstellen des Bildes ist dabei ein passendes Beispiel für die Arbeit mit einer GPU, weil man hier pro Pixel einen sehr hohen Rechenaufwand haben kann und die Anzahl an parallel laufenden Berechnungen das Generieren des Bildes essenziell beschleunigt.
Nimmt man ein 8192x8192 Pixel großes Bild und beschränkt die Anzahl der Iterationen auf 10 000, so hat man im schlimmsten Fall mehr als 670 Milliarden Iterationen, die sequentiell ausgeführt werden. Parallelisiert man den Vorgang jedoch auf beispielsweise 3000 Kernen, dann sind es nur noch ungefähr 220 Tausend Iterationen, die gleichzeitig ausgeführt werden können. Da die Kerne hierbei keinerlei Informationen teilen müssen, kann das Erzeugen des Bildes leicht aufgeteilt werden: Jeder Kern berechnet den Wert für ein Pixel. Hierbei ist die Anzahl der Kerne wichtiger als deren Geschwindigkeit.

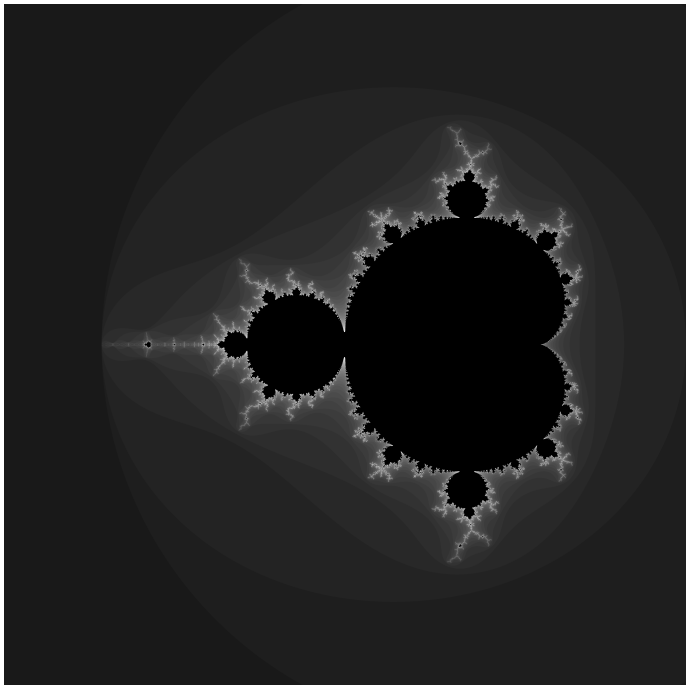

Abbildung 3.2.: Die Mandelbrot-Menge (schwarz) und die in Grauwerten kodierte Anzahl in Iterationen, die benötigt wurden um auszuschließen, dass der Wert innerhalb der Mandelbrot-Menge liegt. Am Rand liegt die Anzahl bei 5 Iterationen und nimmt mit jedem Farbwechsel um 1 zu. Dies geschieht so lange bis die Anzahl so groß ist und der Wert dennoch innerhalb der Grenze liegt, dass wir annehmen können, dass die komplexe Zahl Teil der Mandelbrot-Menge ist und schwarz gefärbt wird.

Somit ist die Mandelbrot-Menge ein gutes Beispiel für die Parallelisierung, aber kein gutes für die GPU. Deren Kerne zwar alle angesprochen werden, aber sehr viele sehr schnell mit der Berechnung fertig sein werden, da sehr viele Pixel wenige Iterationen benötigen. Diese Kerne sind dann alle untätig und müssen auf die anderen warten, bis sie wieder weiter rechnen können.

4. Implementierung

In diesem Kapitel werden die für diese Ausarbeitung ausgewählten Algorithmen vorgestellt und auf deren Implementierung eingegangen.
Die *Basic Linear Algebra Subprograms* (BLAS) Spezifikation beschreibt Basisroutinen gängiger linearer Algebra Operationen mit Vektoren und Matrizen. Diese sind in 3 *Stufen* (engl. *Levels*) unterteilt. In *Stufe 1* sind Funktionen mit zwei Vektoren enthalten: Skalarprodukt, Vektornormierung und eine allgemeine Vektoraddition der Form $y \leftarrow \alpha x + y$.
Davon verwendet wird einerseits die *Vektoraddition* in Abschnitt 4.1, bei der $\alpha = 1$ ist, und die Kombination aus Skalarprodukt und der Addition zweier Vektoren in Abschnitt 4.2. Zusätzlich zu den Vektorrechnungen der BLAS kommen noch die Summe aller Werte eines Vektors (Abschnitt 4.3) und ein *Filter-Map-Reduce* Algorithmus (Abschnitt 4.4) hinzu.
Die *Stufe 2* der BLAS Spezifikation sind Matrix-Vektor-Operationen wie eine allgemeine Matrix-Vektor-Multiplikation (`gemv`, engl, *general matrix-vector multiplication*) $y \leftarrow \alpha Ax + \beta y$, sowie ein Löser der linearen Gleichung $Tx = y$, der nach x auflöst. Diese Stufe ist nicht Teil dieser Ausarbeitung, da sie einer Matrix-Matrix-Multiplikation entspricht, bei der die Breite der zweiten Matrix 1 beträgt.
Zu den *Stufe 3*-Funktionen gehört unter anderem die allgemeine Matrixmultiplikation `gemm` (engl. *general matrix multiplication*) der Form $C \leftarrow \alpha AB + \beta C$. Die Implementierung in Abschnitt 4.5 verwendet die einfache Matrixmultiplikation AB, die man erhält, wenn entweder C eine Nullmatrix oder $\beta = 0$ ist. Ferner ist auch eine Funktion für das Berechnen von $B \leftarrow \alpha T^{-1}B$, wobei T eine Dreiecksmatrix ist, in der *Stufe 3* enthalten.
Des Weiteren soll die genaue Umsetzung der *Mandelbrot-Menge* in Abschnitt 4.6 erläutert werden. In Abschnitt 4.7 soll darüber hinaus erklärt werden, wie die Laufzeiten der jeweiligen Implementierungen gemessen wurde.

Die vollständigen Implementierungen sind online im öffentlichen GitHub Repo[1] der Ausarbeitung zu finden. Dabei zeigt der Tag v0.1 den Stand, mit dem der Vergleich durchgeführt wurde.

4.1. Vektoraddition

Die *Vektoraddition* stellt eine grundlegende Operation im Umgang mit Vektoren dar. Dabei werden alle Elemente der Vektoren paarweise addiert. Im Zuge dieser Ausarbeitung wird das Ergebnis, im Gegensatz zur BLAS Spezifikation, in einem neuen Vektor gespeichert. Um ein Drittel des Speicherplatzes zu sparen, könnten die Werte auch in einen der Eingabevektor (engl. *in-place*) gespeichert werden, jedoch auf Kosten des Verlusts von dessen Werten.

[1] https://github.com/heidrifx/cuda-stdpar-samples

Codebeispiel 4.1 CUDA Kernel der Vektoraddition. Der Code stammt aus den CUDA Beispielen [NVI] (*Samples/0_Introduction/vectorAdd*).

```
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements) {
        int i = blockDim.x * blockIdx.x + threadIdx.x;      // get current index
        if (i < numElements)
                C[i] = A[i] + B[i];
}
```

Die Implementierung als CUDA Kernel ist sehr simpel gehalten, da hier nicht viel beachtet werden muss. Zuerst wir mit `int i = blockDim.x * blockIdx.x + threadIdx.x` der Index des ausführenden Threads bestimmt. Dann werden die Elemente der Eingabearrays `A` und `B` addiert und in den Ausgabearray `C` gespeichert.

Codebeispiel 4.2 Parallel STL Implementierung der Vektoraddition.

```
template<class T>
void vectorAdd(const std::vector<T> &A, const std::vector<T> &B, std::vector<T> &C) {
        std::transform(std::execution::par_unseq,           // execution policy
                A.begin(), A.end(),                         // input 1
                B.begin(),                                  // input 2
                C.begin(),                                  // output
                [=](auto a, auto b) { return a + b; }       // may use std::plus<>{} instead
        );
}
```

Der Parallel STL Code *transformiert* die beiden Eingabevektoren `A` und `B` mittels des Lambdaausdrucks `[=](auto a, auto b) { return a + b; }` in einen neuen Vektor, der im Ausgabevektor `C` gespeichert wird. Dabei dienen die ersten beiden Parameter nach der Ausführungsrichtlinie als *Anfang* und *Ende* der ersten Eingabe und bestimmen damit auch die Länge der Eingabe. Es folgen Iteratoren der anderen Eingabe und der Ausgabe. Die Ausgabe kann auch einer der beiden Eingabevektoren sein. Anstelle des Lambdaausdrucks kann auch die `std::plus<>{}` Funktion verwendet werden, was hier jedoch nicht umgesetzt wurde.

4.2. saxpy und daxpy

`axpy` ($a \cdot \vec{x} + \vec{y}$) ist Teil der *Stufe 1* (engl. *Level 1*) Funktionalität der BLAS. Die hier verwendeten `saxpy` und `daxpy` sind Implementierungen mit einfacher (engl. *single-percision*), respektive doppelter (engl. *double-percision*) Genauigkeit von Fließkommazahlen.

Codebeispiel 4.3 CUDA Kernel für saxpy und daxpy.

```
1  __global__ void saxpy(const int n, const float a, const float *x, float *y) {
2          int i = blockIdx.x * blockDim.x + threadIdx.x;
3          if (i < n)
4                  y[i] = a * x[i] + y[i];
5  }
6
7  __global__ void daxpy(const size_t n, const double a, const double *x, double *y) {
8          size_t i = blockIdx.x * blockDim.x + threadIdx.x;
9          if (i < n)
10                 y[i] = a * x[i] + y[i];
11 }
```

Die axpy Implementierung bestimmt zuerst den Index des Threads und sollte dieser innerhalb der Arraygröße n liegt, berechnet der CUDA Kernel `y[i] = a * x[i] + y[i]`. Das Ergebnis wird in den zweiten Array y geschrieben. Der saxpy Kernel verwendet einen 32-bit Integer (`int`) für die Größe, während daxpy einen 64-bit Integer (`size_t`) verwendet.

Codebeispiel 4.4 Parallel STL Implementierung von saxpy und daxpy.

```
1  void saxpy(const float a, const std::vector<float> &x, std::vector<float> &y) {
2          std::transform(std::execution::par_unseq, x.begin(), x.end(), y.begin(), y.begin(),
3                  [=](auto x, auto y) { return y + a * x; });
4  }
5
6  void daxpy(const double a, const std::vector<double> &x, std::vector<double> &y) {
7          std::transform(std::execution::par_unseq, x.begin(), x.end(), y.begin(), y.begin(),
8                  [=](auto x, auto y) { return y + a * x; });
9  }
```

Unter Parallel STL werden mittels des Lambdaausdrucks `[=](auto x, auto y) { return y + a * x; }` die Vektoren x und y *transformiert* und das Ergebnis in Vektor y geschrieben. Hier muss die Vektorgröße nicht als zusätzlicher Parameter den axpy Funktion übergeben werden, da der `std::transform` Aufruf die Größe aus den beiden Iteratoren des ersten Vektors ermittelt.

4.3. Vektorreduktion

Ein Reduktionsoperator (engl. *reduction operator*) ist ein Operator, der eine Ansammlung von Werten auf einen einzelnen Wert *reduziert*. Zusätzlich muss der Operator das Gesamtergebnis auch aus Teilergebnissen berechnen können. Im Folgenden wird das Aufsummieren aller Werte eines Vektors betrachtet.
Für einen Vektor der Größe n benötigt man hierfür $n - 1$ Schritte, wendet man das Teile-und-herrsche-Verfahren (engl. *divide and conquer*) an so benötigt man nur noch $\log_2 n$ Schritte.

Harris beschreibt in einem CUDA Webinar insgesamt sieben verschiedene Ansätze einer Implementierung der Vektorreduktion [Har]. In dieser Ausarbeitung wurden *Reduction #2* (Zeile 1ff. in Codebeispiel 4.6) und *Reduction #5* (Zeile 20ff. in Codebeispiel 4.6) verwendet.

Codebeispiel 4.5 Naiver Ansatz der Vektorreduktion in CUDA. Das Beispiel stammt aus dem CUDA Webinar *Optimizing Parallel Reduction in CUDA* [Har].

```
__global__ void reduce0(int *g_idata, int *g_odata) {
        extern __shared__ int sdata[];

        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x;
        unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
        sdata[tid] = g_idata[i];
        __syncthreads();

        // do reduction in shared mem
        for (unsigned int s=1; s < blockDim.x; s *= 2) {
                if (tid % (2*s) == 0)
                        sdata[tid] += sdata[tid + s];
                __syncthreads();
        }

        // write result for this block to global mem
        if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Bei der naiven Implementierung (Codebeispiel 4.5), von der Harris [Har] aus geht, kopiert zuerst jeder Thread ein Element der Eingabe, mittels des zuvor ermittelten Thread- und Element-Index, in den geteilten Speicher (engl. *shared memory*) `sdata[]`. Danach müssen alle Threads synchronisiert werden, um ein *data race* zu verhindern. Anschließend wir die Reduktion innerhalb der Schleife durchgeführt. Der Abstand der Elemente, auf die zugegriffen wird, ist dabei eine immer größer werdende Zweierpotenz. Nach jedem Schleifendurchgang müssen die Threads synchronisiert werden, da es sonst erneut zu einem *data race* kommen könnte. Abschließend kopiert der Thread mit `tid == 0` das Ergebnis aus dem geteilten Speicher in die Ausgabe `g_odata[blockIdx.x] = sdata[0]`. Das Ergebnis liegt in Index `0`, weil Thread `0` in jedem Schleifendurchgang die Zwischenergebnisse aufsummiert.
Der Ansatz ist aus zwei Gründen noch nicht performant genug: Der Modulo-Operator `%` ist sehr langsam und durch die Verzweigung `if (tid % (2*s) == 0)` befinden sich immer mehr Threads im Leerlauf (engl. *idle*). Dadurch arbeiten immer weniger Threads am tatsächlichen Problem, bis am Ende nur noch Thread `0` arbeitet.

Codebeispiel 4.6 Zwei verschieden stark optimierte CUDA Kernels für die Berechnung der Vektorreduktion. Der leicht optimierte Kernel ist nahe an einer naiven Implementierung, während der stärker optimierte, deutlich verändert wurde. Beide stammen aus dem CUDA Webinar *Optimizing Parallel Reduction in CUDA* [Har].

```
template<class T>
__global__ void reduce2(const T *g_idata, T *g_odata) {
        extern __shared__ T sdata0[]; // shared memory for intermediate results
        auto tid = threadIdx.x,
                i = blockIdx.x * blockDim.x + threadIdx.x;
        sdata0[tid] = g_idata[i];
        __syncthreads(); // wait until all threads copied their values to shared memory

        // sum up values in pairs of 2
        for (auto s = 1; s < blockDim.x; s *= 2) {
                auto index = 2 * s * tid;
                if (index < blockDim.x) sdata0[index] += sdata0[index + s];
                __syncthreads();
        }

        // only 1 thread has to copy the result
        if (tid == 0) g_odata[blockIdx.x] = sdata0[0];
}

template<class T>
__device__ void warpReduce(volatile T *sdata, uint tid) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid + 8];
        sdata[tid] += sdata[tid + 4];
        sdata[tid] += sdata[tid + 2];
        sdata[tid] += sdata[tid + 1];
}

template<class T>
__global__ void reduce5(const T *g_idata, T *g_odata) {
        extern __shared__ T sdata1[];
        auto tid = threadIdx.x;
        // start ahaed because...
        auto i = blockIdx.x * (blockDim.x * 2) + threadIdx.x;

        // ... we add the first half of values in this assignment
        sdata1[tid] = g_idata[i] + g_idata[i + blockDim.x];
        __syncthreads();

        // reversed loop with threadID-based indexing
        for (uint s = blockDim.x / 2; s > 32; s >>= 1) {
                if (tid < s) sdata1[tid] += sdata1[tid + s];
                __syncthreads();
        }

        // at the end we only habe one warp left
        // -> SIMD synchronous within -> no __syncthreads() or if (tid < s) needed
        if (tid < 32) warpReduce<T>(sdata1, tid);

        if (tid == 0) g_odata[blockIdx.x] = sdata1[0];
}
```

Die erste Optimierung entsteht durch das Verwenden von nicht divergierenden Pfaden (engl. *non-divergent branches*) (siehe Zeile 10 in Codebeispiel 4.6). Indem man einen Index durch `auto index = 2 * s * tid` bestimmt ist es möglich, dass jeder Thread bis zum Ende arbeiten kann. Ebenfalls kann das Entfernen des Modulo-Operators (`%`) (siehe Zeile 11 in Codebeispiel 4.6) die Ausführung zusätzlich beschleunigen.
Als Nächstes kann die Schleifenrichtung umgedreht und der Index mittels `tid` bestimmt werden (siehe Zeile 42ff. in Codebeispiel 4.6). Dadurch ist die Hälfte der Threads jedoch während der ersten Schleifeniteration im Leerlauf. Durch das Addieren der ersten Werte beim Kopieren der Daten in den *shared memory* (siehe Zeile 35 in Codebeispiel 4.6), kann das Problem behoben werden. Im letzten Warp kann man die Schleife entpacken (engl. *unroll*), da hier die Befehle SIMD-synchron ausgeführt werden und man die Threads nicht mehr mittels `__syncthreads()` synchronisiert werden müssen. `if (tid < s)` muss ebenfalls nicht mehr ausgeführt werden. In Zeile 49 von Codebeispiel 4.6 wird diese neue Funktion mit dem entpackten Schleifenkörper aufgerufen – die Definition ist in Zeile 21 zu finden.

Codebeispiel 4.7 Parallel STL Implementierung der Vektorreduktion

```
1 template <class T>
2 T reduce(const std::vector<T> &v) {
3         return std::reduce(std::execution::par_unseq, v.begin(), v.end(), T{});
4 }
```

Die Parallel STL Implementierung ist vergleichsweise sehr kurz, da man hier die gegebene `std::reduce` Funktion verwenden kann. Diese *reduziert* einen Vektor – übergeben durch einen Anfangs- und Enditerator – auf einen Wert und gibt diesen zurück. Dabei ist `T{}` der Anfangswert der Summe. Wenn, wie hier, keine zusätzliche Funktion übergeben wird, wird der `std::plus<>()` Operator als Reduktionsoperator verwendet.
`std::reduce` funktioniert ähnlich wie das nicht parallelisierbare `std::accumulate`, jedoch können die Elemente bei `std::reduce` in beliebiger Reihenfolge gruppiert und umsortiert werden.

4.4. Filter-Map-Reduce

Ein *Filter-Map-Reduce*-Algorithmus (zu dt. etwa *filtern-abbilden-reduzieren*) besteht grundsätzlich aus drei Teilen:

- *Filter* filtert bestimmte Werte der Eingabe heraus und übergibt dann die übrig gebliebenen.
- *Map* bildet die restlichen Werte mittels einer Funktion auf einen neuen Wert ab.
- *Reduce* reduziert die neuen Werte auf einen einzelnen Wert (siehe Abschnitt 4.3).

Dieses Beispiel wurde gewählt, da die STL für alle drei Teile eine eigene Funktion anbietet, man diese aber auch in nur einem einzigen Aufruf ausführen kann. Dadurch können auch mehrere Funktionsaufrufe der Parallel STL mit nur einem verglichen werden. Unter CUDA wird ein einzelner Kernel verwendet, der den kompletten *Filter-Map-Reduce*-Algorithmus ausführt.
Das gewählte Beispiel filtert ungerade Zahlen aus einem Array bzw. Vektor heraus, multipliziert alle geraden Zahlen dann mit 2 und reduziert diese Werte mittels der Addition. Der Algorithmus wurde

so gewählt, da er simpel und dadurch leicht zu verstehen und implementieren ist. Ein komplexerer *Filter-Map-Reduce*-Algorithmus birgt keinen Unterschied im Vergleich zwischen CUDA und der Parallel STL. Einen sinnvollen Nutzen hat der Algorithmus jedoch nicht.

Codebeispiel 4.8 CUDA Kernel des Filter-Map-Reduce-Algorithmus. Die Reduktion wurde von Codebeispiel 4.6 übernommen.

```
1  __global__ void fmr(const int *g_inputData, int *g_outputData) {
2          extern __shared__ int sharedData[];
3          auto tid = threadIdx.x,
4          i = blockIdx.x * blockDim.x + threadIdx.x;
5
6          // copy to shared memory
7          sharedData[tid] = g_inputData[i];
8          // wait for all to finish
9          __syncthreads();
10
11         // filter odd numbers out (i.e. set them to neutral element of addition 0)
12         // map all even numbers to double their amount
13         if (tid < blockDim.x && sharedData[tid] > 0)
14                 (sharedData[tid] & 1) ? (sharedData[tid] = 0) : (sharedData[tid] *= 2);
15         __syncthreads();
16
17         // reduce
18         for (auto s = 1; s < blockDim.x; s *= 2) {
19                 auto index = 2 * s * tid;
20                 if (index < blockDim.x)
21                         sharedData[index] += sharedData[index + s];
22                 __syncthreads();
23         }
24
25         if (tid == 0) g_outputData[blockIdx.x] = sharedData[0];
26 }
```

Der CUDA Kernel kopiert zuerst mittels der Indizes `tid` und `i` (Zeile 3ff. in Codebeispiel 4.8) die Einträge des Eingabearrays in den geteilten Speicher `sharedData[]`. Danach werden alle Threads synchronisiert, um ein *data race* zu verhindern. Anschließend werden alle Werte von `sharedData[]`, die nicht `0` sind, durch den bedingten Ausdruck (engl. *ternary operator*) `(sharedData[tid] & 1) ? (sharedData[tid] = 0) : (sharedData[tid] *= 2)` gefiltert und auf den doppelten Wert abgebildet, sofern der Wert gerade war. Die Reduktion der restlichen, abgebildeten Werte erfolgt dann analog zur Funktion `reduce2` von Codebeispiel 4.6. Abschließend wird das Ergebnis von Thread `0` in die Ausgabe kopiert.

Codebeispiel 4.9 Parallel STL Implementierungen von Filter-Map-Reduce mit drei, respektive einem Funktionsaufruf.

```
template<class T>
T fmr(const std::vector<T> &x) {
        std::vector<T> tmp(x.size());
        // filter
        std::transform(std::execution::par_unseq, x.begin(), x.end(), tmp.begin(),
                [=](auto x) { return x & 1 ? 0 : x; });
        // map
        std::transform(std::execution::par_unseq, tmp.begin(), tmp.end(), tmp.begin(),
                [=](auto x) { return x * 2; });
        // reduce
        return std::reduce(std::execution::par_unseq, tmp.begin(), tmp.end(), T{});
}

template<class T>
T optimized_fmr(const std::vector<T> &x) {
        return std::transform_reduce(std::execution::par_unseq, x.begin(), x.end(), T{},
                std::plus<>(),                              // reduce
                [=](auto x) { return x & 1 ? 0 : x * 2; });   // filter + map
}
```

Die nicht optimierte Parallel STL Implementierung verwendet zwei `std::transform` Aufrufe, um die Eingabe zu filtern und abzubilden. Die Zwischenergebnisse werden im Vektor `tmp` gespeichert, da die Parametersignatur des `fmr` Algorithmus einen konstanten Eingabevektor (`const std::vector`) fordert. Dies wurde so gewählt, damit sich lediglich die beiden Funktionskörper voneinander unterscheiden.
`std::reduce` summiert anschließend die Ergebnisse auf und addiert sie zum Anfangswert `T{}`.
Die optimierte Variante hat nur einen `std::transform_reduce` Aufruf. Dieser hat zwei Lambdaausdrücke als Parameter: die Reduktions- und die Transformationsfunktion. Die Transformationsfunktion bildet hier die ungeraden Zahlen auf 0 und die gerade auf den doppelten Wert ab – effektiv filtert und bildet er somit die Werte in einem Schritt ab.
Wenn keine Funktionen als Parameter übergeben werden, wird `std::plus<>()` als Reduktions- und `std::multiplies<>()` als Transformationsfunktion verwendet, was einem *inneren Produkt* entspricht.

4.5. Matrixmultiplikation

Die Matrixmultiplikation ist, wie `axpy` (Abschnitt 4.2), teil der BLAS Spezifikation. Sie gehört der *Stufe 3* (engl. *Level 3*) an. Die hier verwendete `sgemm` verwendet Fließkommazahlen mit einfacher Präzision (engl. *single-percision*).

Codebeispiel 4.10 CUDA Kernel der `sgemm`. Der Code stammt aus den CUDA Beispielen [NVI] (*Samples/0_Introduction/matrixMul*).

```
template<class T>
__global__ void sgemm(const T *A, const T *B, T *C, const int wA, const int wB) {
        // Block index
        auto bx = blockIdx.x;
        auto by = blockIdx.y;
        // Thread index
        auto tx = threadIdx.x;
        auto ty = threadIdx.y;

        // Index of the first sub-matrix of A processed by the block
        auto aBegin = wA * BLOCK_SIZE * by;
        // Index of the last sub-matrix of A processed by the block
        auto aEnd   = aBegin + wA - 1;
        // Step size used to iterate trough the sub-matrices of A
        auto aStep = BLOCK_SIZE;

        // Index of the first sub-matrix of B processed by the block
        auto bBegin = BLOCK_SIZE * bx;
        // Step size used to iterate through the sub-matrices of B
        auto bStep = BLOCK_SIZE;

        // Csub is used to store the element of the block sub-matrix
        // that is computed by the thread
        auto Csub = T{};

        // Loop over all the sub-matrices of A and B
        // required to compute the block sub-matrix
        for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
                // shared memory for the sub-matrices of A and B
                __shared__ T Asub[BLOCK_SIZE][BLOCK_SIZE];
                __shared__ T Bsub[BLOCK_SIZE][BLOCK_SIZE];

                // load matrices from device to shared memory
                // each threads load one element of each matrix
                Asub[ty][tx] = A[a + wA * ty + tx];
                Bsub[ty][tx] = B[b + wB * ty + tx];
                __syncthreads();

                // Multiply the two matrices together;
                // each thread computes one element
                // of the block sub-matrix
                #pragma unroll
                for (int k = 0; k < BLOCK_SIZE; ++k)
                        Csub += Asub[ty][k] * Bsub[k][tx];

                // Synchronize to make sure that the preceding
                // computation is done before loading two new
                // sub-matrices of A and B in the next iteration
                __syncthreads();
        }
        // Write the block sub-matrix to device memory;
        // each thread writes one element
        auto c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + wB * ty + tx] = Csub;
}
```

Der CUDA Kernel bestimmt zuerst Indizes `aBegin`, `aEnd` und `bBegin` (Zeile 10ff.) für die Teilmatrizen von `A` und `B`, für die der aktuelle Block zuständig ist. Des Weiteren werden deren Schrittgröße für das Iterieren durch die Teilmatrizen auf die `BLOCK_SIZE` gesetzt. `Csub` dient als Zwischenspeicher für die Teilergebnisse der einzelnen Threads. Die äußere Schleife in Zeile 28 iteriert über alle Teilmatrizen, für die der aktuelle Block zuständig ist, und füllt die Teilmatrizen `Asub` und `Bsub` im geteilten Speicher mit Werten vom globalen Speicher. Anschließend müssen die Threads in Zeile 37 synchronisiert werden, wodurch sichergestellt werden kann, dass die Teilmatrizen alle komplett initialisiert wurden. Die innere Schleife in Zeile 43 kann mittels des `#pragma unroll` Pragmas für ausreichend kleine Blockgrößen vom Compiler entpackt (engl. *unroll*) werden. Dadurch ist es möglich ohne Schleifenvariable, die initialisiert, inkrementiert und verglichen werden muss, den Schleifenkörper auszuführen – also effektiv mehrmals `Csub += Asub[ty][k] * Bsub[k][tx]` nacheinander aufrufen, wobei `k` dann immer einen festen Wert hat. Sie multipliziert die beiden Teilmatrizen miteinander, wobei jeder Thread einen Wert ermittelt und diesen auf `Csub` addiert (siehe Zeile 44). Nach dieser Schleife müssen erneut die Threads in Zeile 49 synchronisiert werden, damit sichergestellt werden kann, dass die Berechnung abgeschlossen ist. Zum Schluss wird nach der äußeren Schleife die Teilmatrix des Blocks in die Ausgabe geschrieben (siehe Zeile 53f.). Hierbei kopiert jeder Thread einen Wert.

Codebeispiel 4.11 Parallel STL Implementierung der `sgemm`.

```
1  template<class T>
2  Matrix<T> sgemm(Matrix<T> const &A, Matrix<T> const &B) {
3          auto rows = A.size();
4          auto cols = B[0].size();
5          Matrix<T> result;
6
7          // transpose matrix B
8          Matrix<T> tB(B[0].size(), std::vector<T>(B.size()));
9          #pragma omp parallel for collapse(2)
10         for (auto r = 0; r < B.size(); ++r)
11                 for (auto c = 0; c < B[0].size(); ++c)
12                         tB[c][r] = B[r][c];
13
14         // parallel inner_product
15         for (auto &a: A) {
16                 std::vector<T> tmp(cols);
17                 #pragma omp parallel for
18                 for (auto i = 0; i < tB.size(); ++i)
19                         tmp[i] = std::transform_reduce(
20                                 std::execution::par_unseq,
21                                 a.begin(), a.end(),
22                                 tB[i].begin(), T{});
23                 result.push_back(std::move(tmp));
24         }
25         return result;
26 }
```

Für die Parallel STL Implementierung wird die zweite Matrix zuerst in Zeile 7ff. transponiert. Dadurch muss nur noch zeilenweise multipliziert werden. Unter CUDA wurde die Matrix nicht transponiert, da man hier mittels der Indizes der Matrizen arbeiten konnte. In der Parallel STL Implementierung könnte man zwar *einen* Zeiger an den Schleifenkörper übergeben, aber ohne

zusätzliche Schleife innerhalb eines `std::for_each` diesen nicht in einen Index umwandeln. Der Index würde über den Zeiger `ptr`, dem aktuell betrachteten Element der `std::for_each` Schleife `d` und der Spalten- bzw. Reihenanzahl `num_[rows,cols]` bestimmt werden: `row = (&d - ptr) / num_cols` respektive `col = (&d - ptr) % num_rows`. Diese Schleife hätte dann die größte Rechenlast und wäre nicht länger parallel.
Die Matrix der Parallel STL Implementierung wird mittels zweier Schleifen transponiert, die durch *OpenMP*s `#pragma omp parallel for collapse(2)` in Zeile 9 parallel auf der CPU ausgeführt werden. Da den Funktionen der Parallel STL keine Referenzen übergeben werden können (siehe Codebeispiel 3.4 in Abschnitt 3.3.2) und man diese dringend benötigt, kann nur eine der folgenden drei Schleifen auf der GPU parallelisiert werden: die innere, die das Skalarprodukt berechnet. Die mittlere Schleife, die über die zweite Matrix iteriert, wird erneut in Zeile 17 mittels *OpenMP* parallel auf der CPU ausgeführt. Die innerste Schleife berechnet mit `std::transform_reduce` der Parallel STL dann das *innere Produkt*.

4.6. Implementierung der Mandelbrot-Menge

Die Implementierung der Mandelbrot-Menge verwendet ein quadratisches Bild mit 2^n, $n > 0$ Pixeln pro Seite. Die Iterationen sind auf 10 000 beschränkt. Da beide Implementierungen denselben Code im Kern (Codebeispiel 4.12) verwenden und man diesen mithilfe der Parallel STL nur durch ein `std::for_each` parallelisieren kann, müssen nur die Aufrufe und die Berechnung bestimmter Variablen angepasst werden.

Codebeispiel 4.12 Implementierung der Mandelbrot-Menge. Der Code stammt von Stack Overflow [AD13].

```
1  // c = x0 + iy0
2  float x0 = ((float)col / WIDTH) * 3.5f - 2.5f;
3  float y0 = ((float)row / HEIGHT) * 3.5f - 1.75f;
4
5  // z = x + iy
6  float x = 0.0f;
7  float y = 0.0f;
8  int iter = 0;
9  float xtemp;
10
11 while((x * x + y * y <= 4.0f) && (iter < MAX_ITER)) {
12         // z^2 = x^2 + i2xy - y^2
13         // Re(z^2 + c) = x^2 - y^2 + x0
14         xtemp = x * x - y * y + x0;
15
16         // Im(z^2 + c) = 2xy + y0
17         y = 2.0f * x * y + y0;
18
19         x = xtemp;
20         iter++;
21 }
22
23 int color = iter * 5;
24 if (color >= 256) color = 0;
```

Zuerst wird mittels der Breite und Höhe des Bildes und der Pixelposition der komplexen Zahl $c = x_0 + iy_0$ die Koordinate so skaliert, dass das resultierende Bild auf die Mandelbrot-Menge zentriert ist. Anschließend werden in Zeile 6ff. die temporären Variablen `x` und `y` ($z = x + iy$), sowie die Anzahl der Iterationen `iter` auf `0` gesetzt. Die Schleife erhöht dann mit jeder Iteration die komplexe Zahl z, bis entweder `x * x + y * y > 4.0f` oder `iter >= MAX_ITER` erreicht ist. Der Schleifenkörper nutzt dabei aus, dass der reale Teil von z^2, $\mathrm{Re}(z^2 + c) = x^2 - y^2 + x_0$, und der imaginäre Teil $\mathrm{Im}(z^2 + c) = 2xy + y_0$ sind. `iter` gibt dann die Anzahl an benötigten Iterationen an, bis die Schleife abgebrochen hat. Dieser Wert wird in mit `color = iter * 5` in einen Grauwert umgewandelt, der dann die komplexe Zahl an diesem Pixel repräsentiert. Wenn `color >= 256` kann man davon ausgehen, dass z Teil der Mandelbrot-Menge ist und der Pixel wird schwarz eingefärbt.

Codebeispiel 4.13 Umschließender CUDA Kernel-Code der Mandelbrot-Mengen-Implementierung aus Codebeispiel 4.12. Hier wird zusätzlich die Reihe, Spalte und der Index des Pixels bestimmt. Das Ergebnis wird anschließend in den Eingabearray kopiert.

```
1  __global__ void calc(int *pos) {
2          int row = blockIdx.y * blockDim.y + threadIdx.y;  // WIDTH
3          int col = blockIdx.x * blockDim.x + threadIdx.x;  // HEIGHT
4          int idx = row * WIDTH + col;
5
6          if(col >= WIDTH  row >= HEIGHT  idx >= N) return;
7
8          // main code  (siehe Codebeispiel 4.12)
9
10         pos[idx] = color;
11 }
```

Der CUDA Kernel bestimmt die Koordinaten und den Index im Array des Pixels durch `blockIdx.[y,x] * blockDim.[y,x] + threadIdx.[y,x]` und `idx = row * WIDTH + col`. Wenn die Koordinaten außerhalb des Bildes liegen, wird die Berechnung abgebrochen. Das Ergebnis wird in Zeile 10 in den Eingabearray kopiert.

Codebeispiel 4.14 Umschließender Parallel STL Code der Mandelbrot-Mengen-Implementierung aus Codebeispiel 4.12. Hier wird zusätzlich die Reihe und Spalte aus dem Datenzeiger und den beiden Höhe und Breite Konstanten bestimmt. Das Ergebnis wird anschließend in den Parameter des Lambdaausdrucks kopiert.

```
1  void mandelbrot(std::vector<int> &in) {
2          std::for_each(std::execution::par_unseq,
3                  in.begin(), in.end(),
4                  [ptr = in.data(), num_rows = HEIGHT, num_cols = WIDTH](int& n) {
5                          const size_t row = (&n - ptr) / num_cols;
6                          const size_t col = (&n - ptr) % num_rows;
7
8                          // main code  (siehe Codebeispiel 4.12)
9
10                         n = color;
11                 }
12         );
13 }
```

Der Parallel STL Code setzt den Kernel mittels `std::for_each` um. Dem Lambdausdruck in Zeile 4 werden zusätzlich der Datenzeiger `ptr = in.data()` und die Höhe und Breite des Bildes als Kopie (engl. *capture-by-value*) übergeben. Mit diesen Werten werden dann Reihe und Spalte des Pixels ermittelt. Das Ergebnis wird in Zeile 10 in den Parameter `n` des Lambdaausdrucks kopiert.

4.7. Zeitmessung

Sämtliche Implementierungen verwenden dieselbe Zeitmessung. Verwendet wird dabei die `std::chrono::steady_clock` der `<chrono>` Bibliothek. Mit dem Aufruf `std::chrono::steady_clock::now()` wird der aktuelle Zeitstempel (engl. *timestamp*) ausgegeben.

Codebeispiel 4.15 Unter CUDA beginnt die Zeitmessung vor allen `cudaMemcpy`. Sie endet, wenn die Ergebnisse von *device* auf *host* kopiert wurden. Bei den Parallel STL Implementierungen ist nur der eigentliche Funktionsaufruf von der Zeitmessung umschlossen. Die Ausgabe der Laufzeitdifferenz ist bei beiden gleich.

```
1  void cuda_main() {
2          ... // all cudaMallocs
3          // start timer
4          auto start = std::chrono::steady_clock::now();
5          ... // cudaMemcpys
6          kernel<<<grid_size, block_size>>>(...);
7          ... // cudaDeviceSynchronize as barrier or implicit with cudaMemcpy
8          // stop timer
9          auto end   = std::chrono::steady_clock::now();
10         printf("Total time elapsed: %lims\n",
11                 std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count());
12 }
13
14 void stl_main() {
15         ...
16         // start timer
17         auto start = std::chrono::steady_clock::now();
18         func(...);
19         ... // array access as barrier
20         // stop timer
21         auto end   = std::chrono::steady_clock::now();
22         printf("Total time elapsed: %liµs\n",
23                 std::chrono::duration_cast<std::chrono::microseconds>(end - start).count());
24 }
```

Die Zeitmessung beginnt bei CUDA stets *nach* dem letzten `cudaMalloc` und *vor* dem ersten Kopieren der Daten (`cudaMemcpy`) auf das *device*, da davon ausgegangen wird, dass die Parallel STL ebenfalls Speicher kopieren, jedoch nicht reservieren, muss. Bei der Parallel STL beginnt sie direkt vor dem Aufruf.
Die Messung wird stets nach einer *Barriere* beendet. Unter CUDA ist diese *Barriere* ein `cudaDeviceSynchronize` oder wird implizit durch ein `cudaMemcpy` gegeben (siehe Zeile 7). Bei der Parallel STL ist die Barriere ein Zugriff auf den Ergebnisvektor (siehe Zei-

le 19). Nach dieser *Barriere* ist dann sichergestellt, dass die Ausführung auch komplett abgeschlossen ist. Anschließend wird die Differenz der beiden Zeitstempel mit der Funktion `std::chrono::duration_cast<std::chrono::[milliseconds,microseconds]>(end-start)` berechnet. Diese gibt dann das Ergebnis in der entsprechenden Einheit zurück.

5. Auswertung

Das folgende Kapitel befasst sich mit dem Versuchsaufbau, sowie dessen Ausführung in dieser Arbeit. Außerdem werden die Eckdaten des Experiments erläutert.
Abschnitt 5.1 stellt die *verwendete Hardware* – also die GPUs und CPUs, die für den Vergleich von Wichtigkeit sind – vor. In den darauf folgenden Abschnitten werden dann die Versuchsergebnisse der Ausarbeitung präsentiert. Dabei werden zuerst die *Laufzeiten* (Abschnitt 5.3) der in Kapitel 4 vorgestellten Algorithmen präsentiert und verglichen. Anschließend wird in Abschnitt 5.4 die unterschiedliche *Codelänge* der Implementierungen mit der Komplexität des Codes in Verbindung gebracht.

5.1. Verwendete Hardware

Der Versuch wurde auf insgesamt vier verschiedenen Grafikkarten ausgeführt: zwei Konsumenten-GPUs (engl. *consumer GPU*), einer *GeForce GTX 1070 Ti* und einer *GeForce GTX 1080 Ti* und zwei Datencenter-GPUs (engl. *data center GPU*), einer *Tesla P100 PCIe 16GB* und einer *Quadro GP100*. Die verwendeten GPUs sind allesamt von NVIDIA. Tabelle 5.1 soll wichtige Teile der Spezifikationen der Grafikkarten auflisten. Weitere Informationen zur *Tesla P100* Grafikkarte sind dem Whitepaper von NVIDIA zu entnehmen [NVI16]. Für die *Quadro GP100* Grafikkarte wurde kein Whitepaper gefunden, ebenso wie für die beiden anderen *Konsumenten*grafikkarten. Für diese wird üblicherweise kein Whitepaper zur Verfügung gestellt.

GPU:	**GTX 1070 Ti**	**GTX 1080 Ti**	**Tesla P100**	**Quadro GP100**
Architektur:	Pascal	Pascal	Pascal	Pascal
Anzahl CUDA Kerne:	2 432	3 584	3 584	3 584
Single-Percision:	8,186 TFLOPS	11,34 TFLOPS	9,526 TFLOPS	10,34 TFLOPS
Double-Percision:	255,8 GFLOPS	354,4 GFLOPS	4,763 TFLOPS	5,168 TFLOPS
Speicher:	8 GB	11 GB	16 GB	16 GB
Compute Capability:	6.1	6.1	6.0	6.0

Tabelle 5.1.: Auszug der Spezifikationen der verwendeten GPUs. Die Zahlen stammen unter anderem von der Drittanbieter-Webseite *TechPowerUp*[1], da keine Whitepaper für die *Ti*-Modelle gefunden wurden.

Für das Unterkapitel *Vektoraddition mit unterschiedlich großen Eingaben* in Abschnitt 5.3.7 wurde der Code nicht nur für eine GPU kompiliert, sondern auch mittels `nvc++` für eine CPU (siehe Abschnitt 3.3.2). Die dabei verwendete CPU ist eine *Intel i7-8700k*. Für die restlichen

[1] https://www.techpowerup.com

Implementierungen ist die Leistung der CPU nicht von größter Wichtigkeit, da hier nur die Zeit der eigentlichen Ausführung auf der GPU gemessen wurde. Die CPUs wurden hauptsächlich für das Allokieren der *host* Arrays und Vektoren und deren Initialisierung, sowie die Verifizierung der Daten am Ende verwendet. Der Vollständigkeit halber – und weil die CPUs nicht alle gleich stark waren – sollen dennoch alle CPUs in der folgenden Tabelle aufgelistet werden.

CPU:	**Intel i7-8700K**	**Intel Xeon Gold 5120**	**Intel Xeon Silver 4116**
Architektur:	Coffee Lake	Skylake	Skylake
Kerne:	6	14	12
Threads:	12	28	24
Basistaktung:	3,70 GHz	2,2 GHz	2,4 GHz
L3 Cache:	12 MB	19,25 MB	16,5 MB
Verwendet mit:	GTX 1070 Ti	GTX 1080 Ti	Tesla P100, Quadro GP100

Tabelle 5.2.: Auszug der Spezifikation der verwendeten CPUs. Die Zahlen sind der Webseite von Intel[2] entnommen.

5.2. Verwendete Compiler

Für die Kompilierung der CUDA Kernels auf den Grafikkarten *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* wurde Version V11.4.152 des `nvcc` CUDA-Compilers verwendet. Für die *GTX 1070 Ti* wurde V11.6.124 verwendet. Dabei wurden keine zusätzlichen Compiler-Flags gesetzt, da `nvcc` standardmäßig das Optimierungslevel 3 (`-O3`) für *device* Code verwendet [lRn19].
Sämtlicher Parallel STL Code wurde mit `nvc++` Version 22.5 kompiliert. Hier wurde die Flag `-stdpar` verwendet, damit der Code für die parallel Ausführung auf einer GPU kompiliert wird. Beim Kompilieren der *Matrixmultiplikation* für Abschnitt 5.3.5 wurde dem Compiler die `-fopenmp` Flag übergeben, da hier *OpenMP* Direktiven verwendet wurden. Für Abschnitt 5.3.7 und Abschnitt 5.3.8 wurde der Code zusätzlich noch mit `-stdpar=multicore` für eine CPU kompiliert, da hier auch ein CPU-Kompilat verglichen werden soll.
Angemerkt werden sollte noch, dass alle CUDA Kernels mit dem `nvc++` Compiler kompiliert werden könnten. Darauf wurde verzichtet, um eine reine CUDA- und eine reine Parallel STL-Entwicklungsumgebung zu haben. Erste Tests haben aber gezeigt, dass es keine Laufzeiteinbusen der kompilierten Programme gibt, egal ob `nvcc` oder `nvc++` verwendet wurde.

5.3. Vergleich der Laufzeiten

In diesem Absatz werden die Ergebnisse des Vergleichs, zwischen den zuvor erläuterten CUDA und Parallel STL Implementierungen vorgestellt und erörtert. Dabei werden zuerst die *Laufzeiten* veranschaulicht und verglichen (Abschnitt 5.3) und anschließend in Abschnitt 5.4 die Länge des Codes.

[2] https://www.intel.com/content/www/us/en/products/overview.html

Die Laufzeit wurde dabei – wie in Abschnitt 4.7 beschrieben – gemessen und anschließend über die Konsole ausgegeben. Jede Implementierung wurde $N = 10$-mal ausgeführt und die Werte für die Auswertung anschließend mit einem Skript in einer CSV-Datei gespeichert. Sämtliche Daten, mit Ausnahme von der *Matrixmultiplikation* in Abschnitt 5.3.5 und der *Mandelbrot-Menge* in Abschnitt 5.3.6, wurden randomisiert und nach der Ausführung des CUDA Kernel, respektive des Parallel STL Codes, mittels der CPU verifiziert. Die Größe des Datensatzes wurde stets so gewählt, dass sie der größten Zweierpotenz 2^n, die noch in den Speicher der GPU passt, entspricht. Es spielt dabei keine Rolle, ob man Zweierpotenzen verwendet – diese Wahl ist willkürlich und wurde so gewählt, da der GPUs Speicher meist in Zweierpotenzen angegeben wird. Die Speichergröße kann mit einem Parameter der ausführenden Datei (engl. *executable*) übergeben werden. Diese bestimmt dann die Größe der Datensätze.

Alle Diagramme, außer die von Abschnitt 5.3.7 und Abschnitt 5.3.8, zeigen Boxplots. Diese ermöglichen es auf einen Blick zu sehen, wie sehr die Werte um den Durchschnitt streuen – zu sehen an der Namens gebenden *Box*. Der Durchschnitt ist durch eine horizontale Linie innerhalb der Box gekennzeichnet und wurde nochmals extra als Zahl annotiert. Die vertikal abstehenden Linien mit einer kleineren waagerechten werden *Antennen* (engl. *Whiskers*, zu dt. *Schnurrhaar*) genannt und zeigen den letzten Datenwert, der noch innerhalb

$$Q_3 + 1.5 * IQR \text{ bzw.} Q_1 - 1.5 * IQR$$

liegt. Hierbei beschreiben Q_1 und Q_3 das untere, respektive das obere Quartil und

$$IQR = Q_3 - Q_1$$

den *Interquartilsabstand* IQR. Ausreißer werden mit Rauten (♦) außerhalb der *Antennen* gekennzeichnet.
Alle Diagramme zeigen die Laufzeit auf der y-Achse, jedoch ist diese von Diagramm zu Diagramm unterschiedlich skaliert. Die Messung erfolgte immer in Millisekunden, sofern nichts anderes erwähnt wird.

5.3.1. Vektoraddition

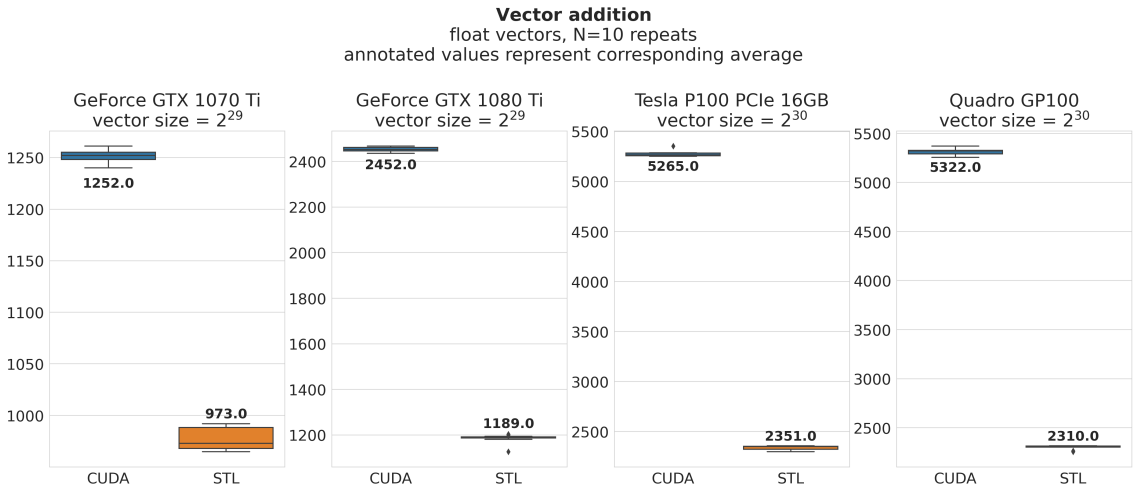

Abbildung 5.1.: Boxplots der Laufzeit der Vektoraddition auf den vier GPUs. Der Versuch wurde mit 2^{29} Vektoreinträgen für *GTX 1070 Ti* und *GTX 1080 Ti* und 2^{30} für *Tesla P100* und *Quadro GP100* durchgeführt.

Abbildung 5.1 zeigt die Ergebnisse der Laufzeitmessungen der Vektoraddition von Vektoren aus Fließkommazahlen mit einfacher Präzision. Dabei wurden auf der *GTX 1070 Ti* und der *GTX 1080 Ti* jeweils zwei Vektoren mit 2^{29} Einträgen und auf der *Tesla P100* und *Quadro GP100* Vektoren mit 2^{30} Einträgen addiert. Die Diagramme zeigen die Laufzeiten des CUDA Kernels links und die der Parallel STL Implementierung rechts. Da die Antennen nahe an den Boxen sind und nur wenige Ausreißer zu sehen sind, kann davon ausgegangen werden, dass die Messwerte nur wenig streuen. Die durchschnittlichen Laufzeiten für 2^{29} Vektoreinträge liegen dabei zwischen 973 ms bei der Parallel STL Implementierung auf der *GTX 1070 Ti* und 2 452 ms bei der CUDA Implementierung auf der *GTX 1080 Ti*. Bei den Messwerten mit Vektoren der Größe 2^{30} durchschnittlich zwischen 2 310 ms auf der *Quadro GP100* unter Parallel STL und 5 322 ms für den CUDA Kernel auf derselben Grafikkarte. Die Parallel STL Implementierung ist durchschnittlich 22,3 % bei der *GTX 1070 Ti*, 51,5 % bei der *GTX 1080 Ti*, 55,3 % bei der *Tesla P100* und 56,6 % bei der *Quadro GP100* schneller als die Messungen, die die CUDA verwendeten – durchschnittlich 46,43 %.

Die Ergebnisse der Parallel STL waren schneller als CUDA. Da der eigentliche CUDA Kernel lediglich `C[i] = A[i] + B[i]` berechnet und somit keine großen Optimierungen erübrigt, ist es unwahrscheinlich, dass es an dessen Code, sondern vielmehr an der Speicherverwaltung liegt. Der Parallel STL Code ist, verglichen damit, nicht stärker optimiert, auch er berechnet eine simple Addition: `a + b`, wobei `a` und `b` Elemente der Vektoren `A` und `B` sind.
Der CUDA Code wurde zusätzlich mit einem USM implementiert und mittels des `nvprof` Profilers die benötigte Zeit der einzelnen CUDA Aufrufe analysiert. Die Analyse hat – für eine Eingabelänge von 2^{28} auf der *GTX 1070 Ti* – ergeben, dass die CUDA Implementierung *ohne* USM ungefähr 97,7 % der Zeit mit dem Kopieren der Daten beschäftigt ist, *mit* USM wird laut dem Profiler 100 % der Zeit mit für die Berechnung verwendet. Allerdings ist zu beobachten, dass die Analyse des USM ergibt, das auch hier 95,3 % der Zeit für das Kopieren benötigt wird. Die Implementierung *mit* USM ist 35,9 % schneller als *ohne* USM, was daran liegt, dass der USM es ermöglicht zusätzlich

zum Kopieren der Daten, parallel Berechnungen durchgeführt werden können. Die Ergebnisse des Profilers sind in Anhang A zu finden.
Bei CUDA *ohne* USM müssen mit `cudaMemcpy` immer zuerst die allokierten Arrays auf die GPU kopiert werden, bevor die eigentliche Berechnung beginnen kann, dies benötigt viel Zeit. Die Parallel STL hingegen verwendet USM, der dynamisch auf der GPU benutzt werden kann, d. h. dass nicht immer der ganze Vektor in den GPU-Speicher kopiert wird, sondern immer nur der Teil, der gerade bearbeitet wird. Ferner kann während dem Kopieren bereits an Daten im Speicher gearbeitet werden.

Die längere Berechnung von *Tesla P100* und *Quadro GP100* lässt sich mit der größeren Eingabe und der damit verbundenen, längeren Kopierzeit erklären. Wenn man davon ausgeht, dass die *reine* Vektoraddition auf der GPU für 2^{30} ungefähr doppelt so lange wie für 2^{29} Vektoreinträge benötigt hat (vgl. Durchschnittswerte der Parallel STL Implementierung in Abbildung 5.1), dann sind die Ergebnisse der beiden Datencenter-GPUs, die eine etwas schlechtere *single-percision* Performance als die *GTX 1080 Ti* aufweisen (vgl. Tabelle 5.1), plausibel. Diese brauchen etwas mehr als doppelt so lange wie die *GTX 1080 Ti*. Warum die *GTX 1070 Ti* beinahe doppelt so schnell unter CUDA ist, wie die *GTX 1080 Ti* und ungefähr 18 % schneller bei der Parallel STL, lässt sich nur schwer erklären.

5.3.2. saxpy und daxpy

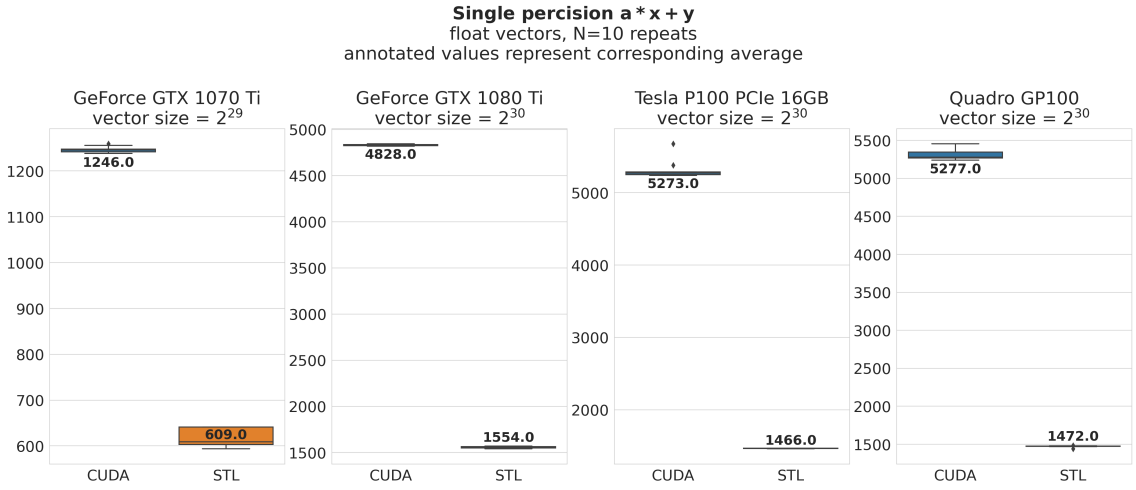

Abbildung 5.2.: Boxplots der Laufzeit von `axpy` mit einfacher Präzision (`saxpy`) auf den vier Grafikkarten. Auf der *GTX 1070 Ti* wurde `saxpy` mit 2^{29} Einträgen und auf *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* mit 2^{30} ausgeführt.

Die Ergebnisse der Laufzeitmessungen der `saxpy` Implementierungen werden in Abbildung 5.2 dargestellt. Die `axpy` Implementierung mit Gleitkommazahlen in einfacher Präzision verwendete dabei auf der *GTX 1070 Ti* zwei Vektoren mit 2^{29} Einträgen, während *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* 2^{30} Einträge benutzten. Die Ergebnisse der CUDA Kernels werden in jedem Diagramm links und die der Parallel STL rechts dargestellt. Die Messwerte streuen nur sehr wenig um den Durchschnitt, was an den eng anliegenden Antennen zu erkennen ist. Nur die Messungen

auf der *Tesla P100* zeigen zwei Ausreißer, die sich etwas weiter von der oberen Antenne entfernt befinden. Die Durchschnittswerte der `saxpy` Messungen bei Vektoren mit 2^{29} Einträgen liegen dabei zwischen 609 ms bei der Parallel STL Implementierung auf der *GTX 1070 Ti* und 1 246 ms bei der CUDA Version auf derselben Grafikkarte. Für 2^{30} Vektoreinträge liegt der durchschnittliche Messwert zwischen 1 466 ms auf der *Tesla P100* unter Parallel STL und 5 277 ms beim CUDA Kernel auf der *Quadro GP100*. Im Schnitt ist die Parallel STL für `saxpy` zwischen 51,1 % bei der *GTX 1070 Ti* und 72,2 % bei der *Tesla P100* schneller als CUDA. Insgesamt ist die Parallel STL 65,8 % performanter als CUDA.

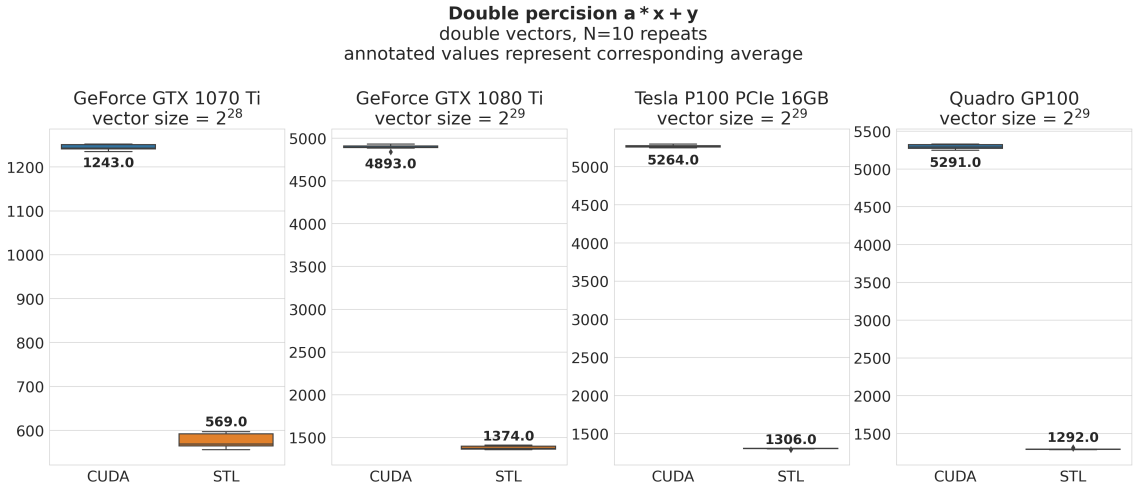

Abbildung 5.3.: Boxplots der Laufzeiten von `axpy` mit doppelter Präzision (`daxpy`) auf den vier GPUs. Auf der *GTX 1070 Ti* wurde `daxpy` mit 2^{28} Einträgen und auf *GTX 1080 TI*, *Tesla P100* und *Quadro GP100* mit 2^{28} ausgeführt.

Die Ergebnisse der Laufzeitmessungen der `daxpy` Implementierungen werden in Abbildung 5.3 präsentiert. Für die `axpy` Implementierung mit Fließkommazahlen in doppelter Präzision wurden Vektoren der Größe 2^{28} für die *GTX 1070 Ti* und 2^{29} für *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* verwendet. Erneut werden die Ergebnisse der CUDA Kernels links und die der Parallel STL rechts dargestellt. Die Laufzeitmessungen streuen ebenfalls nur sehr wenig um den Durchschnitt. Die durchschnittlichen Messwerte der `daxpy` Implementierungen für 2^{28} Vektoreinträge auf der *GTX 1070 Ti* liegen zwischen 569 ms bei der Parallel STL und 1 243 ms bei der CUDA Implementierung. Für die Vektoren mit 2^{29} Einträgen liegt der Durchschnittswert zwischen 1 292 ms für die Parallel STL auf der *Quadro GP100* und 5 291 ms für CUDA auf derselben GPU. Bei den `daxpy` Implementierungen sieht es ähnlich wie bei den `saxpy` Implementierungen aus: Auch hier ist die Parallel STL durchschnittlich zwischen 54,2 % (bei der *GTX 1070 Ti*) und 75,6 % (bei der *Quadro GP100*) schneller. Im Schnitt ist die Parallel STL zusammengerechnet 69,23 % schneller.

Die Laufzeitmessungen zeigen, dass die Vektoraddition mit der Parallel STL schneller ist als mit CUDA. Auch hier ähneln sich die Implementierungen sehr: Der CUDA Kernel berechnet `y[i] = a * x[i] + y[i]` und im verwendeten Parallel STL Code wird `y + a * x` als Berechnung verwendet. Beide Stellen lassen sich rein rechnerisch nicht weiter optimieren, somit kann die langsamere Laufzeit der CUDA Kernels nur durch das zusätzliche `cudaMemcpy` erklärt werden. Der USM der Parallel STL ist einer der wichtigsten Gründe für die schnellere Ausführung. Auch hier haben erste Tests gezeigt, dass eine CUDA Implementierung *mit* USM die Laufzeit verbessern.

Für doppelt so große Eingaben (2^{30} statt 2^{29} bei `saxpy`, bzw. 2^{29} statt 2^{28} bei `daxpy`) benötigen die Implementierungen, die CUDA verwenden, 3,87-4,24 mal mehr Zeit bei `saxpy` und 3,94-4,26 mal mehr bei `daxpy`. Bei der Parallel STL hingegen durchschnittlich nur 2,41-2,55 mal mehr Zeit bei `saxpy` und 2,27-2,41 mal mehr bei `daxpy`. Für die Parallel STL wirken die Zahlen plausibel, da hier, verglichen mit der Vektoraddition, noch zusätzlich eine Multiplikation hinzukommt und somit der Rechenaufwand leicht gestiegen ist – daher die 46 % bzw. 32 % längere Laufzeit. Bei den CUDA Implementierungen ist ebenfalls der Aufwand gestiegen, aber die längere Zeit kann nur durch das zusätzliche Kopieren der Daten erklärt werden.
Die schnelleren Laufzeiten der beiden Datencenter-GPUs bei der `daxpy` Implementierung unter Parallel STL kann mit der höheren Rechenleistung für Fließkommazahlen mit doppelter Präzision (siehe Tabelle 5.1) erklärt werden. Warum diese aber auch bei `saxpy` schneller sind als die *GTX 1080 Ti*, die eigentlich eine höhere Leistung aufweist, kann nicht wirklich erklärt werden. Dies könnte mit dem zufälligen Datensatz oder der Speicherbandbreite zusammenhängen. Die CUDA Kernel sind allesamt langsamer, je mehr Rechenleistung die Grafikkarten aufweisen, was an der Plausibilität dieser Messungen zweifeln lässt.

5.3.3. Vektorreduktion

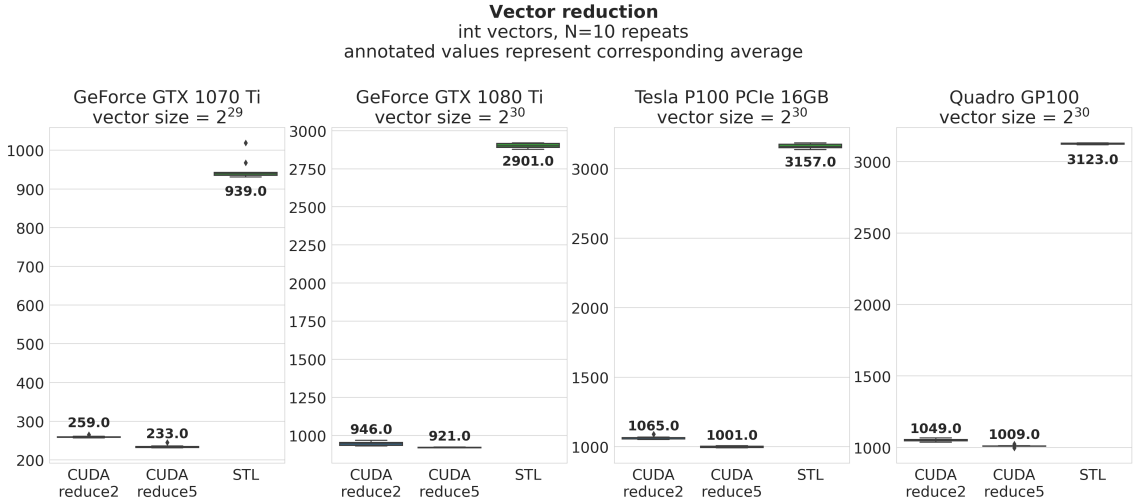

Abbildung 5.4.: Boxplots der Laufzeit der Vektorreduktion auf den view GPUs. Dabei wird auf der *GTX 1070 Ti* ein Vektor der Länge 2^{29} reduziert, während *GTX 1080 TI*, *Tesla P100* und *Quadro GP100* 2^{30} Einträge verarbeiten. *CUDA reduce2* und *CUDA reduce5* entsprechen dabei den gleichnamigen Funktionen aus Codebeispiel 4.6 in Abschnitt 4.3.

Die Messergebnisse der Vektorreduktions-Laufzeiten werden in Abbildung 5.4 veranschaulicht. Auf der *GTX 1070 Ti* wurden Ganzzahlvektoren der Größe 2^{29} verwendet, während *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* eine Vektorgröße von 2^{30} verwenden. Die linken beiden Boxplots in jedem Diagramm zeigen die Laufzeiten von CUDA und der rechte die von der Parallel STL. Wieder streuen die Laufzeiten nur minimal um den Mittelwert. Die durchschnittliche, gemessene Laufzeit von CUDA auf der *GTX 1070 Ti* mit einer Vektorlänge von 2^{29} liegt zwischen 233 ms und 259 ms

– für die Parallel STL bei 939 ms. Für Vektoren der Größe 2^{30} auf *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* ist die Laufzeit im Schnitt zwischen 921 ms und 1 065 ms für CUDA und zwischen 2,9 s und 3,2 s für die Parallel STL. Die Parallel STL Implementierung ist auf der *GTX 1070 Ti* durchschnittlich 4,0 mal langsamer als die `reduce5` Implementierung, die optimierte Version der Vektorreduktion unter CUDA, und 3,6 mal als `reduce2`. Für *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* ist Parallel STL nur ungefähr 3,1 mal langsamer als `reduce5` auf CUDA und ungefähr 3,0 mal als `reduce2`.

Die Messwerte der Laufzeiten der Vektorreduktion zeigen, dass CUDA trotz der zusätzlichen `cudaMemcpy` schneller ist als die Parallel STL. Die Ergebnisse scheinen plausibel zu sein, da hier die CUDA Implementierungen deutliche Optimierung verglichen mit der Parallel STL vorweisen. Der Coder der Parallel STL lässt sich nicht so feingranular wie die CUDA Kernel optimieren, weil hier ein `std::reduce` Befehl verwendet werden *muss* und lediglich die Reduktionsfunktion manuell übergeben werden kann. Da hier aber eine simple Addition durchgeführt wird, macht es wenig Sinn nicht die standardmäßige `std::plus<>{}` Funktion des Compilers zu verwenden. Somit lässt sich die Parallel STL Implementierung nicht weiter optimieren – die CUDA Kernel hingegen schon (siehe Abschnitt 4.3). Bei dieser können beispielsweise Additionen bereits beim Kopieren in den geteilten Speicher durchgeführt werden oder die letzten 32 Aufrufe entpackt und in einem Warp abgearbeitet werden.
Die *GTX 1080 Ti* weist die schnellste Laufzeit bei der Parallel STL für Vektoren der Größe 2^{30} auf, dies war mit den Spezifikationen der Grafikkarten (siehe Tabelle 5.1) zu erwarten. Für CUDA ist die *GTX 1080 Ti* unter den GPUs, die eine Reduktion mit 2^{30} Vektoreinträgen berechnet haben, ebenfalls die schnellste. Die *GTX 1070 Ti* ist bei allen drei Messergebnissen nochmal schneller, hat aber auch eine kleinere Eingabe (2^{29} Vektoreinträge). Verglichen mit der Leistungsärmsten GPU, die ein doppelt so großes Problem löst, der *Tesla P100*, die ungefähr 1,2 mal schneller ist, ist die *GTX 1070 Ti* bei der Parallel STL Implementierung 3,4 mal langsamer. Die Diskrepanz von $3.4 - (2.0 + 1.2) = 0.2$ ist noch im Rahmen und die Messwerte scheinen plausibel zu sein. Für CUDA ist der Unterschied zwischen $3.7 - (2.0 + 1.2) = 0.5$ bei `reduce2` und $4.0 - (2.0 + 1.2) = 0.8$ bei `reduce5`, was diese Werte nicht ganz so plausibel aussehen lässt.
Des Weiteren ist zu sehen, dass die Optimierungen von `reduce5` die Laufzeit im Vergleich zu `reduce2` um 2,6 %-10,0 % senken.

5.3.4. Filter-Map-Reduce

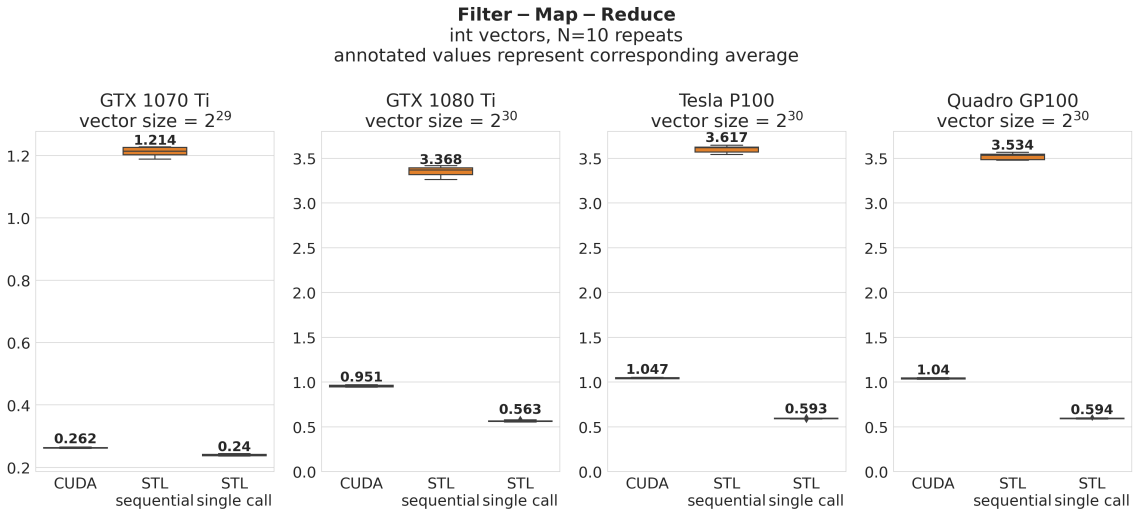

Abbildung 5.5.: Boxplots der Laufzeiten von Filter-Map-Reduce in Sekunden auf den vier Grafikkarten. Erneut wird auf der *GTX 1070 Ti* ein Vektor der Länge 2^{29} bearbeitet, während *GTX 1080 TI*, *Tesla P100* und *Quadro GP100* 2^{30} Einträge verarbeiten. *STL sequential* entspricht dabei der Funktion `fmr`, die insgesamt drei Funktionen der STL aufruft, und *STL single call* der Funktion `optimized_fmr`, die nur einen Funktionsaufruf hat, aus Codebeispiel 4.9 in Abschnitt 4.4.

In Abbildung 5.5 werden die Messungen der Laufzeit des Filter-Map-Reduce Algorithmus dargestellt. Hier wurde auf der *GTX 1070 Ti* ein Vektor mit 2^{29} Ganzzahleinträgen verwendet, während *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* 2^{30} Vektoreinträge hatten. Die Diagramme zeigen, von links nach rechts, die Laufzeiten der CUDA Kernel, der sequentiellen Parallel STL Implementierung und der Parallel STL Implementierung mit nur einem Funktionsaufruf. Die Messungen streuen nur sehr wenig um den Mittelwert. Die durchschnittliche Laufzeit für 2^{29} Vektoreinträge liegt zwischen 0,24 s bei der Parallel STL Implementierung mit nur einem Aufruf und 1,214 s bei der Parallel STL Implementierung mit drei Funktionsaufrufen auf der *GTX 1070 Ti*. Die Durchschnittswerte der Messungen mit 2^{30} Einträgen liegen zwischen 0,563 s beim Parallel STL Code mit einem Aufruf auf der *GTX 1080 Ti* und 3,617 s beim Parallel STL Code mit drei Aufrufen auf der *Tesla P100*. Die Parallel STL ist mit nur einem Funktionsaufruf durchschnittlich 8,4 % auf der *GTX 1070 Ti*, 40,8 % auf der *GTX 1080 Ti*, 43,4 % auf der *Tesla P100* und 43,9 % auf der *Quadro GP100* schneller als die CUDA Kernel – insgesamt durchschnittlich 34,13 %. Für die Implementierung, die drei Funktionen sequentiell aufruft, ist die Parallel STL langsamer als CUDA: 4,634 mal für die *GTX 1070 Ti*, 3,542 mal für die *GTX 1080 Ti*, 3,455 mal für die *Tesla P100* und 3,398 mal für die *Quadro GP100* – zusammen im Schnitt 3,76 mal.

Diese Ergebnisse bestätigen erneut, was in Abschnitt 5.3.1 und Abschnitt 5.3.2 gezeigt wurde: die Parallel STL ist schneller als CUDA. Da in die CUDA Kernel der gleiche Aufwand wie für die Parallel STL Implementierungen investiert wurde, sind diese eventuell nicht zu 100 % optimiert (gerade bei der Reduktion kann noch optimiert werden, siehe Abschnitt 4.3). Dennoch kann beobachtet werden, dass die Parallel STL nur dann schneller ist, wenn all drei Schritte – filtern,

abbilden und reduzieren – in einem Funktionsaufruf berechnet werden. Wenn dies nicht der Fall ist, so ist die Parallel STL durchschnittlich ungefähr 5,77 mal langsamer. Dies liegt daran, dass die Werte insgesamt dreimal auf die GPU kopiert werden müssen und anschließend wieder von der GPU auf den *host*, auch wenn hier USM verwendet wird.
Für die doppelte Eingabelänge wurde bei CUDA ungefähr die 3,87 fache Laufzeit gemessen, während die Parallel STL bei einem Funktionsaufruf nur durchschnittlich 2,43 und bei drei Aufrufen 2,89-mal länger gebraucht hat. Für die Parallel STL wirkt dieser Wert plausibel. Die *GTX 1080 Ti* ist für *single-percision* Rechnungen die schnellste GPU, gefolgt von der *Quadro GP100* und der *Tesla P100* (siehe Tabelle 5.1), was sich auch in den Messungen widerspiegelt. Die *GTX 1070 Ti* ist die langsamste Grafikkarte, daher sollten *GTX 1080 Ti*, *Tesla P100* und *Quadro GP100* allesamt schneller als das Doppelte sein, was sie jedoch nicht sind. Die 43 % bzw. 89 % längere Laufzeit entsteht wahrscheinlich durch die Multiplikation der gefilterten Werte im Abbildungsschritt und beim Code mit den drei Aufrufen durch das zusätzliche Kopieren zwischen den Funktionsaufrufen und dem Warten auf die komplette Ausführung der vorherigen Funktionen. Die längere Laufzeit der CUDA Kernel kann nur durch den schlecht optimierten Reduktionsschritt und die `cudaMemcpy`, erklärt werden.

5.3.5. Matrixmultiplikation

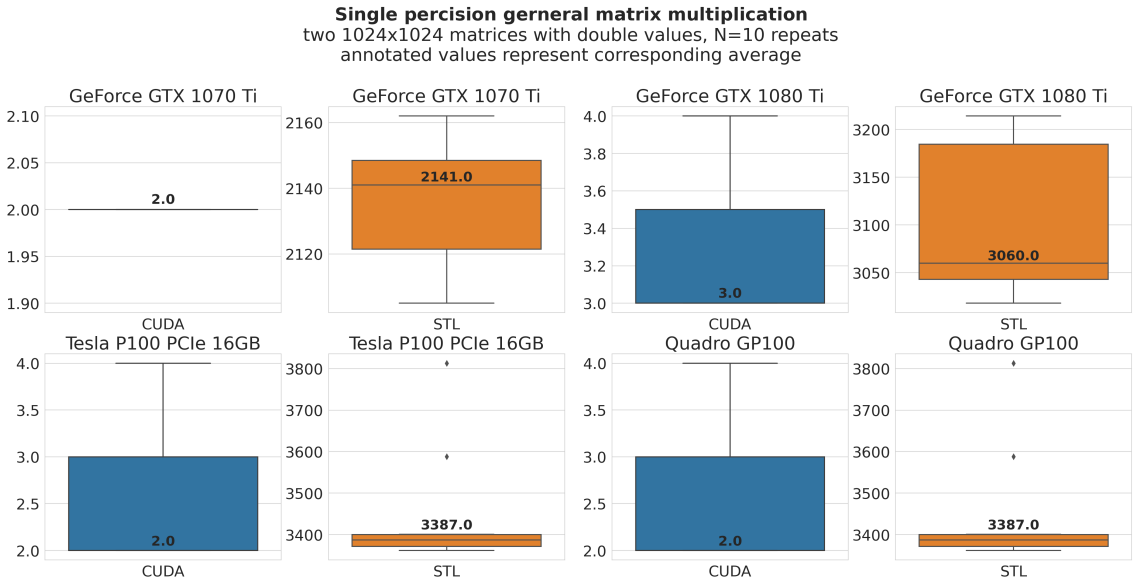

Abbildung 5.6.: Boxplots der Matrixmultiplikation-Laufzeiten auf den vier GPUs. Die Boxplots wurden in einzelnen Diagrammen dargestellt, da die Werte von CUDA und STL sehr weit auseinander liegen. Auf allen Grafikkarten wurden zwei Matrizen der Größe 1024x1024 miteinander multipliziert. Die y-Achsen der jeweiligen Diagramme ist hier auch auf derselben Grafikkarte unterschiedlich.

Die Ergebnisse der Laufzeitmessungen der Matrixmultiplikation werden in Abbildung 5.6 gezeigt. Es wurden auf allen GPUs zwei 1024x1024 Matrizen miteinander multipliziert. Die Einträge der Matrizen sind dabei Gleitkommazahlen mit doppelter Präzision (engl. *double*). Links sind die

Messwerte der CUDA Kernel und rechts die der Parallel STL zu sehen. Die Laufzeiten der CUDA Implementierung sind alle an einem sehr kleinen Wert (2 ms bzw. 3 ms) angelegt und streuen maximal um 1 ms nach oben. Bei der Parallel STL hingegen ist die Streuung größer, obwohl die Diagramme ähnlich aussehen, was aber hauptsächlich an der anderen Skala dieser Messung liegt. Die durchschnittliche Laufzeit der Implementierung, die CUDA verwenden, liegt zwischen 2 ms auf der *GTX 1070 Ti*, der *Tesla P100* und der *Quadro GP100* und 3 ms auf der *GTX 1080 Ti*. Dagegen braucht die Parallel STL Implementierung zwischen 2 141 ms auf der *GTX 1070 Ti* und 3 387 ms auf der *Tesla P100* und *Quadro GP100*. Die CUDA Kernel ist auf der *GTX 1070 Ti* 1 070-mal, auf der *GTX 1080 Ti* 1 019-mal und auf *Tesla P100* und *Quadro GP100* 1 693-mal schneller als der Parallel STL Code.

Die Laufzeit ist unter Parallel STL deutlich langsamer als dieselbe Berechnung mit CUDA. Der Grund hierfür liegt an der Implementierung, wie bereits in Abschnitt 4.5 erläutert, die es unmöglich macht den zweidimensional CUDA Kernel gut mittels Parallel STL zu parallelisieren. Die Größe der Matrizen wurde absichtlich klein gewählt, da die Parallel STL Implementierung sonst zu lange – verglichen mit der CUDA Implementierung – rechnen würde. Bereits für zwei Matrizen der Größe 2048x2048 würde die Messung viermal länger dauern. Der CUDA Kernel können jedoch deutlich größere Matrizen multiplizieren.
Weshalb *Tesla P100* und *Quadro GP100* beide langsamer als die anderen GPUs sind, ist verwunderlich. Eigentlich wäre mit der höheren Performance der Grafikkarten (siehe Tabelle 5.1) zu erwarten, dass die *Quadro GP100* am schnellsten und die *GTX 1070 Ti* am langsamsten ist. Vor allem da die Eingabegröße auf allen GPUs gleich war.

5.3.6. Mandelbrot-Menge

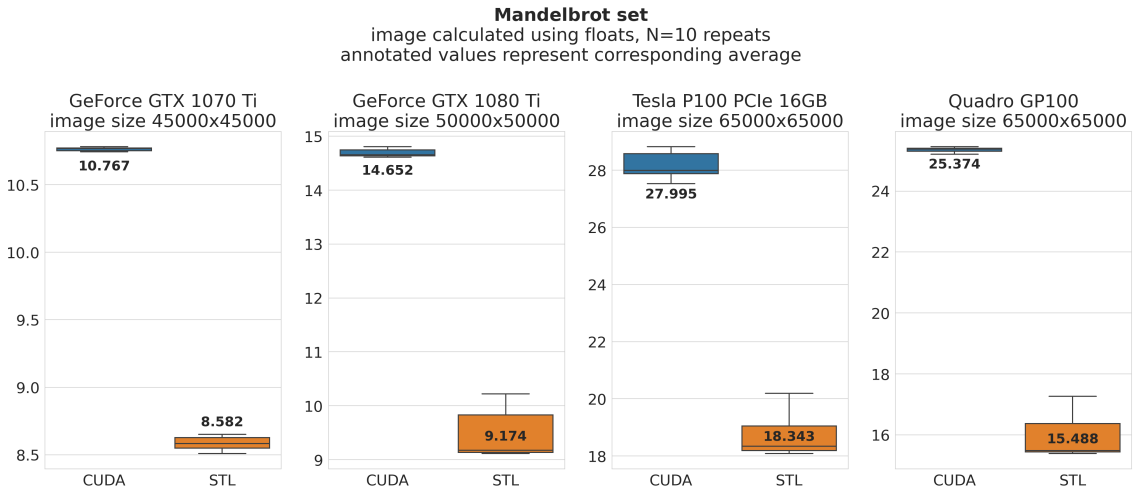

Abbildung 5.7.: Boxplots der Laufzeit der Mandelbrot-Menge auf den vier Grafikkarten. Die Bilder der Mandelbrot-Menge sind immer quadratisch und auf der *GTX 1070Ti* haben eine Seitenlänge von 45 000 Pixeln, auf der *GTX 1080 Ti* 50 000 Pixeln und auf *Tesla P100* und *Quadro GP100* 65 000 Pixeln. Die Laufzeit wurde in Sekunden gemessen.

Abbildung 5.7 zeigt die Ergebnisse der Laufzeitmessungen der Mandelbrot-Mengen Implementierungen. Auf der *GTX 1070 Ti* wurde das quadratische Bild der Mandelbrot-Menge mit 45 000 Pixeln und auf der *GTX 1080 Ti* mit 50 000 Pixeln erstellt. Die beiden Datacenter-GPUs – *Tesla P100* und *Quadro GP100* – verwenden jeweils 65 000 Pixel für das generierte Bild. Zur Berechnung der einzelnen Pixel wurden Fließkommazahlen mit einfacher Präzision (engl. *float*) verwendet. Die einzelnen Bilder wurden nur im Speicher erstellt und nicht als Datei abgespeichert, da diese zu viel Speicherplatz benötigen würden. Erneut wird CUDA links und die Parallel STL rechts im Diagramm dargestellt. Die Laufzeiten liegen dabei zwischen 8,6 s bei der Parallel STL Implementierung auf der *GTX 1070 Ti* und 28,0 s beim CUDA Kernel auf der *Tesla P100*. Die Parallel STL ist durchschnittlich 20,3 % auf der *GTX 1070 Ti*, 37,4 % auf der *GTX 1080 Ti*, 34,5 % auf der *Tesla P100* und 39,0 % auf der *Quadro GP100* schneller als die CUDA Implementierung. Nimmt man den Durchschnitt aller Messungen, so ist die Parallel STL 32,8 % schneller als CUDA.

Die Messungen belegen wieder die Ergebnisse der *Filter-Map-Reduce* Messungen in Abschnitt 5.3.4 und den Abschnitten davor. Die Parallel STL weist erneut deutliche, zeitliche Vorteile im Vergleich zu CUDA vor. Der Code der beiden Implementierungen ist beinahe äquivalent (siehe Abschnitt 4.4) und dennoch birgt die Parallel STL eine bessere Performance. Dies liegt am USM, den die Parallel STL verwenden.
Wie bei Abschnitt 4.1 wurde auch hier erneut der CUDA Code mit USM implementiert und mittels des `nvprof` Profilers die beiden CUDA Ausführungen verglichen. Der Profiler hat dabei – für 35 000 Pixel Seitenlänge auf der *GTX 1070 Ti* – ergeben, dass CUDA *ohne* USM 72,6 % der Zeit (6,0 s) mit der Berechnung und 27,4 % (2,3 s) mit dem Kopieren des Ergebnisses auf den *host*. Bei der Implementierung *mit* USM wurde 100 % der Zeit mit der Berechnung verbracht. Die Analyse des USM zeigt, dass keine zusätzliche Zeit für das Kopieren der Daten verwendet wurde. CUDA *mit* USM ist 25,3 % schneller als *ohne*, was erneut daran liegt, dass parallel zur Berechnung die fertigen Pixelwerte auf den *host* kopiert werden können. Die Ergebnisse des Profilers sind in Anhang B zu finden.

Es sollte noch erwähnt werden, dass unter der Parallel STL und CUDA *mit* USM die Eingabegröße nicht auf die Speichergröße der GPU begrenzt ist, sondern auf das Maximum von ebenjener und dem RAM des Rechners. Der Grund hierfür ist der USM, der nur dann Werte tatsächlich auf die GPU kopiert, wenn diese auch verwendet werden. Beim CUDA Kernel *muss* immer direkt der komplette Speicher belegt werden, wenn der *globale Speicher* wie hier verwendet wurde, der durch die Speichergröße der GPU begrenzt ist.
Die Ergebnisse sind plausibel. Die Laufzeiten sollten von der *GTX 1070 Ti* mit 45 000 Pixeln pro Seite auf die 65 000 Pixel der *Tesla P100* und *Quadro GP100* ungefähr

$$\left(\frac{65\,000}{45\,000}\right)^2 \approx 2.09$$

mal größer sein. Tatsächlich sind sie 2,6-mal von *GTX 1070 Ti* auf *Tesla P100* und 2,4-mal auf *Quadro GP100* bei den CUDA Kernel und 2,1-mal bzw. 1,8-mal für die Parallel STL Implementierung langsamer. Wenn man das länger dauernde Kopieren bei der CUDA Implementierung berücksichtigt, kann die Differenz von 0,5 bzw. 0,3 vernachlässigt werden. Da die *Quadro GP100* im Vergleich zur *Tesla P100* eine bessere *single-percision* Performance (10,34 TFLOPS anstelle von 9,526 TFLOPS, siehe Tabelle 5.1) ist die schnellere Laufzeit der GPU nachvollziehbar. Dies würde auch die besser Messung der *Quadro GP100* – nur 1,8-mal langsamer statt den erwarteten 2,09 – erklären.

5.3.7. Vektoraddition mit unterschiedlich großen Eingaben

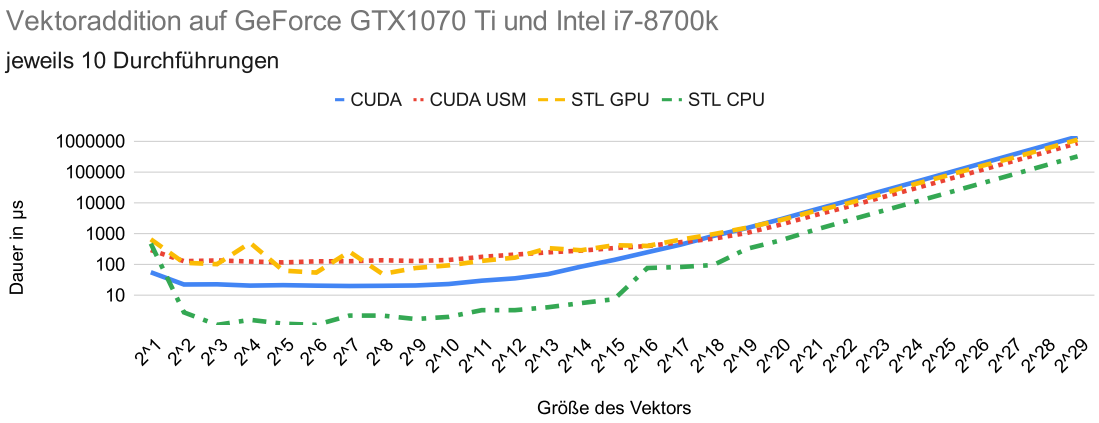

Abbildung 5.8.: Diagramm der Vektoraddition mit wachsender Eingabelänge. Der Versuch wurde nur auf der *GTX 1070 Ti* durchgeführt, enthält aber zusätzlich eine Ausführung auf der *i7-8700k* CPU von Intel mittels der `-stdpar=multicore` Flag des `nvc++` Compilers. Die x-Achse zeigt die Größe der Eingabe angefangen bei 2^1 und endet mit 2^{28}. Die durchgezogene Linie zeigt CUDA, die gepunktete CUDA mit USM, die gestrichelte Parallel STL für GPU und die gepunktet-gestrichelte Parallel STL für CPU. Die y-Achse zeigt die Laufzeit in µs auf einer logarithmischen Skala. Der Versuch wurde 10-mal wiederholt.

In Abbildung 5.8 ist die Laufzeit der Vektoraddition mit verschiedenen Eingabegrößen für CUDA mit und ohne USM und der Parallel STL auf einer GPU und einer CPU zu sehen. Verwendet wurde die *GTX 1070 Ti* als GPU und die *i7-8700k* als CPU. Der Versuch wurde zehnmal wiederholt. Die Eingabegrößen gehen von 2^1 bin 2^{28} mit Gleitkommazahlen in einfacher Präzision als Vektoreinträge. Die y-Achse ist logarithmisch skaliert, da die Laufzeiten sehr schnell anwachsen. Die durchgezogene Linie zeigt CUDA, die gepunktete CUDA mit USM, die gestrichelte Parallel STL für GPU und die gepunktet-gestrichelte Parallel STL für CPU. Bis auf ein paar Ausreißer und die Ergebnisse bei 2^1 nimmt die Laufzeit mit steigender Eingabegröße zu. Die Zunahme ist ab einer Eingabegröße von ungefähr 2^{11}-2^{13} exponentiell, was an den linear, auf der logarithmischen Skala, wachsenden Messungen zu erkennen ist. Davor sind die Laufzeiten bis auf ein paar Ausreißer ungefähr konstant. Bis einschließlich 2^{19} ist CUDA ohne USM schneller als die Parallel STL auf der GPU, danach jedoch nicht mehr. Ab einer Eingabelänge von 2^{18} ist CUDA mit USM schneller als ohne und ab 2^{16} durchgängig schneller als die Parallel STL für GPU. Die Parallel STL auf der CPU ist immer schneller als die beiden anderen Implementierungen. Lediglich für 2^1 nicht.

Dieses Ergebnis ist sehr interessant, da bisher davon auszugehen war, dass die Parallel STL immer die bessere Wahl ist, wenn es der Code zulässt (siehe Abschnitt 5.3.5). Anscheinend ist CUDA ohne USM bei kleineren Eingaben schneller – sogar schneller als CUDA mit USM. CUDA ohne USM ist zwischen 371,1 % und 2,7 % schneller als die Parallel STL Implementierung und nach einer Eingabelänge von 2^{17} nur zwischen 6,1 % und 21,2 % langsamer. Die Messwerte der Parallel STL auf einer CPU waren, wie zu erwarten bei einer einfachen Aufgabe wie der Vektoraddition, immer schneller wie die der GPUs, da für diese Werte zwischen CPU und GPU kopiert werden

müssen, was weitaus mehr Zeit in Anspruch nimmt als die eigentliche Berechnung. Im Durchschnitt ist CUDA ohne USM zwischen 69,2 % und 95,2 % langsamer als die Parallel STL auf der CPU, CUDA mit USM zwischen 62,5 % und 99,2 % und die Parallel STL auf der GPU zwischen 71,1 % und 99,7 % langsamer.
CUDA mit USM ist zwischen 64,1 % langsamer und 48,2 % schneller als Parallel STL für GPU – durchschnittlich ist CUDA mit USM 6,9 % schneller. Für ausreichend große Eingaben, also die Messwerte, bei denen CUDA mit USM schneller war als Parallel STL für GPU, ist sie rund 31,4 % schneller. Somit ist kein großer Unterschied zwischen den beiden Implementierungen festzustellen. Die Fluktuationen können durch die randomisierten Datensätze erklärt werden.
Die Ausreißer wurden bei diesen Berechnungen außen vor gelassen. Die Messungen von Eingaben der Länge 2^1 können nur als Anomalien aufgefasst werden und wurde für die Auswertung nicht weiter beachtet.

5.3.8. Mandelbrot-Menge mit unterschiedlich großen Eingaben

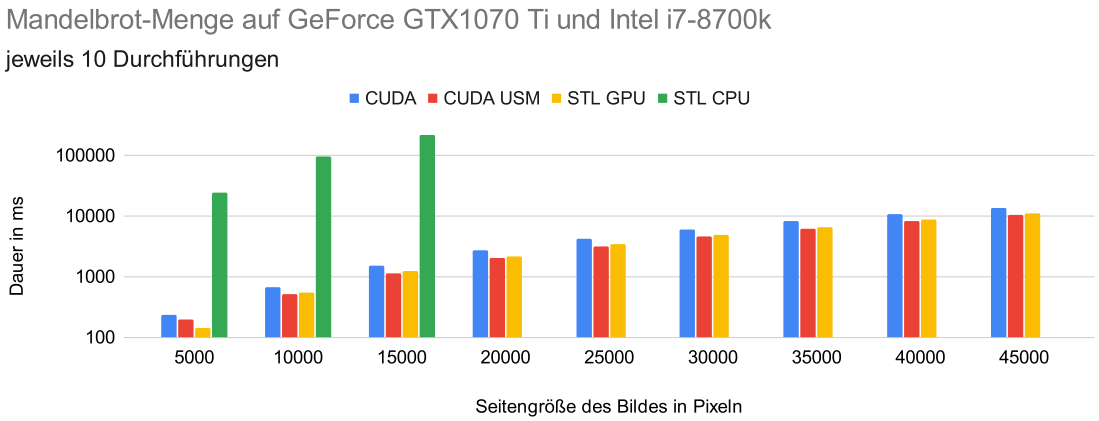

Abbildung 5.9.: Säulendiagramm der Mandelbrot-Menge mit wachsender Bildgröße. Der Versuch wurde nur auf der *GTX 1070 Ti* durchgeführt, enthält aber zusätzlich eine Ausführung auf der *i7-8700k* CPU von Intel mittels der `-stdpar=multicore` Flag des `nvc++` Compilers. Die x-Achse zeigt die Größe der Eingabe angefangen bei 5 000 und endet mit 45 000. Die Säulen sind für jede Bildgröße in einer Vierergruppe (CUDA mit USM, CUDA ohne USM, Parallel STL für GPU und Parallel STL für CPU) angeordnet. Die Parallel STL Implementierung für CPU hat nur Messungen bis 15 000 Pixel Seitenlänge, da schon hier die Laufzeit, verglichen mit den anderen Messungen, deutlich größer ist. Die y-Achse zeigt die Laufzeit in ms auf einer logarithmischen Skala. Der Versuch wurde 10-mal wiederholt.

Abbildung 5.9 zeigt die Laufzeitmessungen der Mandelbrot-Menge mit unterschiedlich großen Eingaben mit CUDA mit und ohne USM, der Parallel STL auf einer GPU und der Parallel STL auf einer CPU. Es wurde erneut die *GTX 1070 Ti* als Grafikkarte und die *Intel i7-8700k* CPU verwendet. Der Versuch wurde zehnmal wiederholt. Die Eingabegröße, also die Seitenlänge des Mandelbrot-Mengen-Bildes, variiert zwischen 5 000 und 45 000 Pixeln. Wie bereits in Abschnitt 5.3.6 wurde auch hier für jeden Pixel die Berechnung mit Fließkommazahlen in einfacher Präzision durchgeführt.

Die Laufzeiten werden in Millisekunden, in einer logarithmischen Skala auf der y-Achse dargestellt. Bei jeder Seitenlänge sind vier Säulen, die, von links nach rechts, die CUDA Kernel ohne und mit USM, die Parallel STL Implementierung auf der GPU und die Parallel STL Implementierung auf der CPU zeigen. Die durchschnittlich gemessene Laufzeit des CUDA Kernels ohne USM geht von 234 ms bei 5 000 Pixeln bis zu 13,5 s bei 45 000 Pixeln, die für CUDA mit USM von 198,9 ms bis 10,3 s. Bei der Parallel STL Implementierung auf der GPU von 140,9 ms bei 5 000 Pixeln bis zu 10,9 s bei 45 000 Pixeln und für Parallel STL auf der CPU von 24,0 s bei 5 000 Pixeln bis zu 215,2 s $\approx$ 3,59 min bei 15 000 Pixeln. Die Parallel STL Implementierung auf der GPU ist immer schneller als der CUDA Kernel. CUDA mit USM ist, bis auf die Eingabelänge von 5 000 Pixel, immer schneller als Parallel STL für GPU und bei allen Seitenlängen schneller als CUDA ohne USM. Der für CPU kompilierte Parallel STL Code ist deutlich langsamer – so viel langsamer, dass nur Laufzeiten für maximal 15 000 Pixel gemessen wurden, da bereits hier die Laufzeit länger dauerte als 3 min.

Das Ergebnis der Messungen zeigt, dass CUDA ohne USM zwischen 23,9 % und 20,9 % langsamer ist als die Parallel STL für GPU. Für eine Seitenlänge von 5 000 ist CUDA ohne USM sogar 66,1 % langsamer. Dies hängt erneut mit dem USM zusammen, wie bei den Messungen von CUDA mit USM zu erkennen ist. Die Laufzeiten von Parallel STL für GPU liegen sehr nah an denen von CUDA mit USM und sind nur 5,3 %-9,3 % (29,2 % wenn man den Messwert von 5 000 Pixeln hinzuzählt) langsamer. Dass CUDA ohne USM und die Parallel STL für GPU deutlich schneller als die Parallel STL für CPU sind, war zu erwarten, da die GPU mit ihren vielen SMs hier einen deutlich höheren Durchsatz an komplexeren Rechnungen im Vergleich zu einer *Vektoraddition* hat und nicht die Hauptzeit im Kopieren der Daten steckt. Die Parallel STL für CPU ist im Schnitt 170,0-175,0 mal langsamer als die Parallel STL für GPU, 102,4-142,8 mal langsamer als CUDA ohne USM und 120,4-188,2 mal langsamer als CUDA mit USM.

5.4. Codelänge

In diesem Kapitel wird die Codelänge der Beispiele verglichen, da bereits bei der Implementierung aufgefallen ist, dass sich die Länge der CUDA Implementierungen drastisch von der Länge der Parallel STL unterscheidet. Verglichen wird die Anzahl der Programmzeilen (engl. *Source Lines Of Code* (SLOC)), das heißt sämtliche Codezeilen, die keine Leerzeilen oder Zeilen mit ausschließlich Kommentaren sind.
Dabei wird beispielhaft eine Implementierung gewählt und an dieser die Unterschiede gezeigt. Anschließend wird die Codelänge der anderen Implementierungen in Tabelle 5.3 präsentiert. In Tabelle 5.4 soll abschließend der Unterschied zu den Parallel STL Implementierungen aufgezeigt werden.

```
1  // may use template as well
2  __global__ void vectorAdd(const float *A, const
   ↪  float *B, float *C, int numElements) {
3    int i = blockDim.x * blockIdx.x + threadIdx.x;
4    if (i < numElements)
5      C[i] = A[i] + B[i];
6  }
7
8
9
10 ...
11 cudaMalloc((void **) &d_A, size),
12   cudaMalloc((void **) &d_B, size),
13   cudaMalloc((void **) &d_C, size);
14 cudaMemcpy(d_A, h_A, size,
15   cudaMemcpyHostToDevice);
16 cudaMemcpy(d_B, h_B, size,
17   cudaMemcpyHostToDevice);
18 int threadsPerBlock = 1<<9,
19   blocksPerGrid = numElements / threadsPerBlock;
20 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>
21   (d_A, d_B, d_C, numElements);
22 cudaMemcpy(h_C, d_C, size,
23   cudaMemcpyDeviceToHost);
24 // use data in C
25 cudaFree(d_A);
26 cudaFree(d_B);
27 cudaFree(d_C);
```

Codebeispiel 5.1

```
1  template<class T>
2  void vectorAdd(const std::vector<T> &A, const
   ↪  std::vector<T> &B, std::vector<T> &C) {
3    std::transform(
4      std::execution::par_unseq,
5      A.begin(), A.end(),
6      B.begin(), C.begin(),
7      [=](auto a, auto b) { return a + b; }
8    );
9  }
10 ...
11 std::vector<float> A(numElements),
12   B(numElements),
13   C(numElements);
14 // no memcpy necessary
15
16
17
18 // no possibilty to set threadsPerBlock
19 // or blocksPerGrid
20 vectorAdd<float>(A, B, C);
21
22 // no memcpy necessary
23
24 // use data in C
25 // no cudaFree necessary
26
27
```

Codebeispiel 5.2

Abbildung 5.10.: CUDA und Parallel STL Code der Vektoraddition für SLOC-Vergleich. Es wurden Leerzeilen und Kommentare eingefügt, um übereinstimmende Codestellen zu verdeutlichen.

Codebeispiel 5.1 und Codebeispiel 5.2 zeigen einen direkten Vergleich der Implementierungen der Vektoraddition unter CUDA und der Parallel STL. Es wurden dabei Leerzeilen und Kommentare an den Stellen eingefügt, an denen der jeweils andere Code keine Funktionsaufrufe hat. Wenn man nur den CUDA Kernel mit der `vectorAdd` Methode der Parallel STL vergleicht so unterscheiden sie sich lediglich in 2 SLOC, da man Zeile 3 bis Zeile 8 von Codebeispiel 5.2 zu einer Zeile zusammenfassen kann. Der größte Unterschied an SLOC von CUDA entsteht durch die zusätzlichen `cudaMemcpy` (Zeile 18ff. und Zeile 22f. in Codebeispiel 5.1) und Kernelaufrufsparameter (Zeile 18f. in Codebeispiel 5.1), sowie `cudaFree` (Zeile 25 in Codebeispiel 5.1).
Der CUDA Code hat 17 SLOC, wenn man die aus Platzgründen auf zwei Zeilen verteilten Funktionsaufrufe in nur einer Zeile schreibt. Der entsprechende Parallel STL Code dagegen nur 8 SLOC. Somit weist CUDA mehr als das Doppelte an SLOC auf.
Verwendet man USM unter CUDA, so fallen 3 SLOC weg, da keine `cudaMemcpy` mehr benötigt werden und nur `cudaMalloc` durch `cudaMallocManaged` ausgetauscht werden muss. Aber selbst dann ist der CUDA Code immer noch 1,875 mal länger als der Parallel STL Code.

Codebeispiel	CUDA	Parallel STL	zusätzliche Implementierungen
Vektoraddition	61	27	43 (CUDA mit USM)
saxpy/daxpy	52	29	
Vektorreduktion	61	24	71 (`reduce5`, CUDA optimiert)
Filter-Map-Reduce	64	29	27 (Parallel STL optimiert)
SGEMM	81	37	
Mandelbrot-Menge	57	51	54 (CUDA mit USM)

Tabelle 5.3.: Codelängen der Beispiele in *Source Lines Of Code* (SLOC). Dabei werden in der rechten Spalte alle zusätzlichen Implementierungen, wie etwa CUDA mit USM, die `reduce5` Optimierung der Vektorreduktion oder die optimierte Parallel STL Filter-Map-Reduce Implementierung, gezeigt.

Tabelle 5.3 zeigt die SLOC der anderen Implementierungen. Hier wurde jedoch der gesamte Quellcode (wie er auch im öffentlichen GitHub Repo[3] zu finden ist), mit Initialisierung der Variablen, der Zeitmessung und der Verifikation, wenn gegeben, bei der Berechnung der SLOC verwendet und nicht wie zuvor eine gekürzte Version. Dabei sollte erwähnt werden, dass die Zeiterfassung und die Verifikation für jedes Beispiel bei allen Implementierungen gleich war und somit nicht in die Differenz der SLOC einfließt. Auch hier wurde sämtlich Leerzeilen bzw. Zeilen ausschließlich mit Kommentaren entfernt. In der rechten Spalte von Tabelle 5.3 sind die zusätzlichen Implementierungen von CUDA mit USM bei Vektoraddition und der Mandelbrot-Menge, der optimierte `reduce5` Algorithmus für die Vektorreduktion unter CUDA und der optimierte Parallel STL Code von Filter-Map-Reduce zu finden.
Generell ist der Parallel STL Code kürzer, da hier sämtliche `cudaMemcpy`, `cudaFree` und Kernelaufrufsparameterberechnungen wegfallen. Dass der CUDA Code mit USM ebenfalls kürzer wie ohne USM ist, war zu erwarten, da hier alle `cudaMemcpy` wegfallen. `reduce5` ist länger als `reduce2`, was an den zusätzlichen Optimierungen, sowie der neuen `warpReduce` Funktion (siehe Zeile 20ff. in Codebeispiel 4.6) liegt. Der optimierte Filter-Map-Reduce Code der Parallel STL ist 2 SLOC kürzer, da hier die drei STL Aufrufe in einem berechnet werden. Die Implementierungen der Mandelbrot-Menge sind nur 6 SLOC auseinander, was an dem sehr ähnlichen Code der eigentlichen Berechnung (siehe Abschnitt 4.6) liegt. Die 6 SLOC entstehen lediglich durch einen zusätzlichen Vergleich im CUDA Kernel und der *device* Variablen, die allokiert, kopiert und freigegeben werden müssen.

Tabelle 5.4 zeigt die Einsparnisse an SLOC, wenn die Parallel STL im Vergleich zu CUDA bzw. den zusätzlichen Implementierungen verwendet wurde. Der Parallel STL Code ist zwischen 66,2 % bei `reduce5` und 5,6 % bei der Mandelbrot-Menge mit USM kürzer als CUDA Code. Im Schnitt ist der Parallel STL Code 43,23 % kürzer als der entsprechende CUDA Code. Die nicht optimierte Filter-Map-Reduce Implementierung der Parallel STL ist 7,4 % länger.
Der kürzere Code führt zu einer besseren Lesbarkeit und dadurch zu einer besseren Verständlichkeit für den Leser.

[3] https://github.com/heidrifx/cuda-stdpar-samples

Codebeispiel	CUDA	zusätzliche Implementierungen	
Vektoraddition	−55,7 %	−37,2 %	(CUDA mit USM)
saxpy/daxpy	−44,2 %		
Vektorreduktion	−60,7 %	−66,2 %	(`reduce5`, CUDA optimiert)
Filter-Map-Reduce	−54,7 %	+7,4 %	(Parallel STL optimiert)
SGEMM	−54,3 %		
Mandelbrot-Menge	−10,5 %	−5,6 %	(CUDA mit USM)
Durchschnittlich	43,23 %		

Tabelle 5.4.: Codelängen der Parallel STL Implementierungen im Vergleich mit CUDA und den zusätzlichen Implementierungen.

5.5. Fazit

Abschließend kann gesagt werden, dass die Performance der Parallel STL Implementierung, wenn diese sich von CUDA übersetzen lässt, ungefähr 49,68 % besser ist als die reine CUDA Implementierung. Dieser Wert beschreibt die durchschnittliche Laufzeitverbesserung, der in dieser Ausarbeitung verglichenen Beispiele, bei denen die Parallel STL schneller war.
Wenn USM bei CUDA verwendet wird, ist die Parallel STL vernachlässigbar langsamer. Beim Vergleich der Laufzeiten der Vektoraddition in Abschnitt 5.3.7 war CUDA mit USM 6,9 % schneller als die Parallel STL und bei der Mandelbrot-Menge in Abschnitt 5.3.8 7,3 %.
CUDA ist – im Fall der in dieser Arbeit behandelten Beispiele – immer dann performanter, wenn sich der Code nicht gut unter Parallel STL umsetzen lässt (siehe *Matrixmultiplikation* in Abschnitt 5.3.5, dort war die Parallel STL ungefähr 1 363,75 mal langsamer) oder der CUDA Kernel stark optimiert ist (siehe `reduce5` in Abschnitt 5.3.3, hier war CUDA 3,1 mal schneller).

Der Quellcode, der hier verwendeten Parallel STL Implementierungen, ist im Vergleich zu CUDA durchschnittlich 43,23 % kürzer. Da ein kürzerer Code auch dazu führt, dass man ihn schneller lesen und auffassen kann, führt er zu einem besseren Verständnis. Es fallen nicht nur sehr viele Funktionsaufrufe wie `cudaMalloc`, `cudaMemcpy` und `cudaFree` weg, was es für Anfänger leichter macht Code zu schreiben, sondern auch die Anordnung des Grafikkartenspeichers in Grids, Blöcke und Threads durch `kernel<<<blockPerGrid, threadsPerBlock>>>`. Deshalb kann bei der Parallel STL jedoch nur der Funktionsaufruf der STL optimiert werden.

Zusammengefasst kann gesagt werden, dass die Parallel STL demnach eine gute Alternative zu CUDA darstellt, insbesondere weil der Parallel STL Code eine ähnliche, wenn nicht sogar bessere, Leistung wie ein naiver CUDA Code nachweist, schneller geschrieben kann und durch die kürzere Codelänge leichter verständlicher ist.

6. Zusammenfassung

Am Anfang dieser Arbeit wurden die Grundlagen einer GPU erklärt und wie man diese zur Parallelisierung von Programmen mittels der CUDA API bzw. der Parallel STL verwenden kann. Es wurden mehrere Beispiele für beide Herangehensweisen ausgewählt, implementiert und anschließend auf insgesamt vier unterschiedlichen Grafikkarten (*NVIDIA GTX 1070 Ti*, *NVIDIA GTX 1080 Ti*, *NVIDIA Tesla P100 PCIe 16GB* und *NVIDIA Quadro GP100*) verglichen. Sowohl der CUDA Code, als auch der Parallel STL Code wurden – mit Ausnahme der Vektorreduktion, da hier eine sehr gute Optimierung zur Verfügung stand – laienhaft gehalten, damit keiner der beiden durch Optimierungen begünstigt wird und dadurch das Ergebnis verfälscht.

Der Vergleich hat dabei ergeben, dass die Parallel STL eine sehr gute Alternative zu CUDA darstellt und sogar eine schnellere Laufzeit nachweist, wenn keine weiteren Speicheroptimierungen am CUDA Code vorgenommen werden. Wenn der CUDA Code *Unified Shared Memory* (USM), einen Speicher, der sich einen Adressbereich auf CPU und GPU teilt, verwendet liegt dieser Code bei Laufzeitmessungen leicht vor der Parallel STL. Ein CUDA Kernel kann jedoch nicht immer optimal in Parallel STL übersetzt werden, was am Beispiel der *Matrixmultiplikation* gezeigt wurde.
Generell ist der Parallel STL Code kürzer als die entsprechende Implementierung mit CUDA, da die Speicherverwaltung für die Grafikkarte entfällt, da Parallel STL einen USM verwendet, der automatisch angelegt und verwaltet wird. Ferner fallen auch sämtliche, für den CUDA Compiler benötigten, Marker wie `__global__` oder `<<<...>>>` beim Kernelaufruf, sowie die Kernelparamter weg. Dies erleichtert die Nutzung von Parallel STL zusätzlich gegenüber CUDA, da man sich keinerlei Gedanken über `threadsPerBlock` und `blocksPerGrid` machen muss.

Wegen des kürzeren Programmcodes und der schnelleren Laufzeit ist die Parallel STL deswegen sehr gut für rapides Prototyping (engl. *rapid prototyping*) geeignet, da man im gewohnten C/C++ Code schreiben kann und dieser, ohne groß nachzudenken und Änderungen machen zu müssen, parallel auf einer GPU ausführbar ist. Der Code kann auch, wenn gewünscht, leicht für CPUs kompiliert werden, was nativ durch den NVIDIA `nvc++` Compiler unterstützt wird.

7. Ausblick

Die Implementierungen dieser Ausarbeitung wurden allesamt sehr einfach gehalten und, mit Ausnahme der *Vektorreduktion* in Abschnitt 4.3, gar nicht bzw. nur sehr wenig optimiert. Daher wäre es interessant zu wissen, ob man durch geschickte Optimierungen oder das Verwenden von USM unter CUDA an die Laufzeitwerte der Parallel STL herankommen kann. Für die *Vektoraddition* und die *Mandelbrot-Menge* wurde bereits eine Implementierung mit USM angefertigt (siehe Abschnitt 5.3.7 und Abschnitt 5.3.8), deren Laufzeit minimal schneller, als die der Parallel STL ist. Eine zukünftige Ausarbeitung könnte den CUDA Code der anderen Beispielen mit USM umsetzen oder den Code – mit dem Ziel einer besseren Laufzeit – optimieren und durch neue Messungen die bessere Performance bestätigen.

In Abschnitt 5.3.7 ist die reine CUDA Implementierung *ohne* USM für kleine Problemgrößen die schnellste Variante gewesen. Hier könnte noch dahingehend geforscht werden, warum genau CUDA *ohne* USM schneller als *mit* USM für kleine Eingaben war und ob sich dies generell für kleinere Eingabewerte sagen lässt.

Für eine zukünftige Fortsetzung dieser Arbeit sollten auch komplette Benchmark-Suiten in die Parallel STL übersetzt und mit CUDA verglichen werden. Die hier ausgewählten Beispiele sind eher als *Proof of Concept* zu verstehen. Die Übersetzung einer ganzen Benchmark-Suite hätte den Rahmen dieser Arbeit überschritten. Benchmarks-Suite, die infrage kommen, sind unter anderem:

- *miniBUDE*[1] ist eine *mini*-Applikation der Implementierung der zentralen Berechnung der *Bristol University Docking Engine* (BUDE) in verschiedenen HPC Programmiermodellen. Die Benchmark stammt aus dem biologischen Bereich und steht bereits in *CUDA*, *OpenMP target*[2], *OpenCL*[3], *OpenACC*[4], *SYCL*[5] und *Kokkos*[6] zur Verfügung.
- *Rodinia*[7] umfasst insgesamt 23 verschiedene Anwendungen aus verschiedenen Bereichen wie bspw. der *linearen Algebra* oder des *Data Minings*. Sie wurde in *CUDA* und *OpenCL* übersetzt.
- NVIDIA selbst vergleicht mehrere HPC Programme[8] unter CUDA mit einer Ausführung auf einer CPU. Diese 13 Applikationen könnten eventuell auch in Parallel STL übersetzt und mit den Zahlen von NVIDIA verglichen werden.

[1] https://github.com/UoB-HPC/miniBUDE
[2] https://enccs.github.io/openmp-gpu/target/
[3] https://www.khronos.org/opencl/
[4] https://www.openacc.org/
[5] https://www.khronos.org/sycl/
[6] https://kokkos.org/
[7] Dokumentation: https://www.cs.virginia.edu/rodinia/, Implementierung: https://github.com/yuhc/gpu-rodinia
[8] https://developer.nvidia.com/hpc-application-performance

Im Bezug auf die in dieser Arbeit verwendete *Matrixmultiplikation* (siehe Abschnitt 4.5), die sich nicht gut mit der Parallel STL realisieren ließ, wäre es auch wissenswert, ob sich alle Benchmarks komplett in die Parallel STL übersetzen lassen.

Die von Intel entworfene *oneAPI*[9] bietet eine weitere Plattform, um Code auf unterschiedlichen Beschleunigern (engl. *accelerators*) wie GPU, FPGAs oder AI-Beschleunigern, aber auch CPU, auszuführen. Dabei soll der Code nicht nur unabhängig von der Plattform, sondern auch von Herstellern, Standards und Programmiermodellen sein. So können beispielsweise sowohl NVIDIA GPUs, wie auch AMD GPUs mittels der Parallel STL *oder* dem *Data Parallel C++* (DPC++) Kompatibilitätswerkzeug für CUDA angesprochen werden.
In diesem Bereich könnte man einerseits den CUDA Code in DPC++ übersetzen[10] und diesen mit den hier vorgestellten CUDA Implementierungen vergleichen. Andererseits könnten die Parallel STL Beispiele aus dieser Arbeit an die *oneAPI* angepasst[11], anschließend mittels `dpcpp` anstelle von `nvc++` kompiliert werden und ebenfalls verglichen. Des Weiteren könnte mittels der *oneAPI* die Laufzeit noch zusätzlich auf GPUs von AMD, FPGAs oder anderen Beschleunigern gemessen werden.

[9] https://www.oneapi.io
[10] https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html#gs.dol6k0
[11] https://docs.oneapi.io/versions/latest/onedpl/pstl_main.html

Literaturverzeichnis

[1080-ti] TechPowerUp. *NVIDIA GeForce GTX 1080 Ti Specs*. URL: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877 (zitiert auf S. 19).

[AD13] Andro, R. Dahl. *CUDA Mandelbrot set*. Nov. 2013. URL: https://stackoverflow.com/questions/20205941/cuda-mandelbrot-set (zitiert auf S. 41).

[Bre18] M. Breyer. „Ein hoch-performanter (approximierter) k-Nächste-Nachbarn Algorithmus für GPUs“. B.S. thesis. 2018. DOI: 10.18419/opus-10152 (zitiert auf S. 23).

[Cra18] A. V. Craen. *GPU-beschleunigte Support-Vector Machines*. Deutsch. Bachelorarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Simulation großer Systeme. Bachelorarbeit. Apr. 2018. DOI: 10.18419/opus-10148. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-2018-59&engl=0 (zitiert auf S. 23).

[geekbench4] P. L. Inc. *Geekbench 4*. URL: https://www.geekbench.com/geekbench4/ (zitiert auf S. 19).

[Har] M. Harris. *Optimizing Parallel Reduction in CUDA*. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (zitiert auf S. 33–35).

[KKN13] I. Karlin, J. Keasler, R. Neely. *LULESH 2.0 Updates and Changes*. Techn. Ber. LLNL-TR-641973. Livermore, CA, Aug. 2013, S. 1–9 (zitiert auf S. 18).

[LCB21] J. Latt, C. Coreixas, J. Beny. „Cross-platform programming model for many-core lattice Boltzmann simulations“. In: *PLOS ONE* 16.4 (Apr. 2021), S. 1–29. DOI: 10.1371/journal.pone.0250306. URL: https://doi.org/10.1371/journal.pone.0250306 (zitiert auf S. 17).

[LMK+20] J. Latt, O. Malaspinas, D. Kontaxakis, A. Parmigiani, D. Lagrava, F. Brogi, M. B. Belgacem, Y. Thorimbert, S. Leclaire, S. Li, F. Marson, J. Lemus, C. Kotsalos, R. Conradin, C. Coreixas, R. Petkantchin, F. Raynaud, J. Beny, B. Chopard. „Palabos: Parallel Lattice Boltzmann Solver“. In: *Computers & Mathematics with Applications* (2020). ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2020.03.022 (zitiert auf S. 17).

[LMS+] J. Latt, F. Marson, J. P. de Santana Neto, K. Thyagarajan, C. Coreixas, B. Chopard. „From CPU to GPU in 80 days“. In: (). URL: https://www.researchgate.net/profile/Christophe-Coreixas-2/publication/354861047_From_CPU_to_GPU_in_80_days/links/6151b7e6154b3227a8b0d98b/From-CPU-to-GPU-in-80-days.pdf (zitiert auf S. 17, 18).

[lRn19] laurenluckiez, Robert_Crovella, njuffa. *nvcc optimization flags*. Jan. 2019. URL: https://forums.developer.nvidia.com/t/nvcc-optimization-flags/69530 (zitiert auf S. 46).

[Moo06] G. E. Moore. „Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.“ In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), S. 33–35. DOI: 10.1109/N-SSC.2006.4785860 (zitiert auf S. 19).

[Ms22] MatColgrove, sweemer. *Which threading library does -stdpar=multicore use?* Juni 2022. URL: https://forums.developer.nvidia.com/t/which-threading-library-does-stdpar-multicore-use/217978/2 (zitiert auf S. 27).

[NVI] NVIDIA. *CUDA Samples*. URL: https://github.com/NVIDIA/cuda-samples (zitiert auf S. 32, 39).

[NVI16] NVIDIA. *NVIDIA Tesla P100. The Most Advanced Datacenter Accelerator Ever Built*
Featuring Pascal GP100, the World's Fastest GPU. 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (zitiert auf S. 45).

[OLL20] D. Olsen, G. Lopez, B. A. Lelbach. *Accelerating Standard C++ with GPUs Using stdpar*. Aug. 2020. URL: https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/ (zitiert auf S. 17, 18, 28).

[Sea22] R. Searles. *NVIDIA HPC STANDARD LANGUAGE PARALLELISM, C++*. Apr. 2022. URL: https://www.nersc.gov/assets/Uploads/4-7-22-ORNL-Stdpar.pdf (zitiert auf S. 18).

[Sin21] D. Singal. *N Ways to SAXPY: Demonstrating the Breadth of GPU Programming Options*. Apr. 2021. URL: https://developer.nvidia.com/blog/n-ways-to-saxpy-demonstrating-the-breadth-of-gpu-programming-options/ (zitiert auf S. 17, 18).

Alle URLs wurden zuletzt am 02.10.2022 geprüft.

A. Ergebnisse des `nvprof` Profilers für die Vektoraddition

Die folgenden beiden Log-Dateien zeigen die Ausgabe des `nvprof` Profilers für die Vektoraddition von 2^{28} Elemente auf der *GTX 1070 Ti* unter CUDA *mit* (Codebeispiel A.1) und *ohne* (Codebeispiel A.2) USM. Dabei ist vor allem der Abschnitt `GPU activities` wichtig, da hier die eigentliche Arbeit der GPU analysiert und der Abschnitt `Unified Memory profiling result`, in dem der USM betrachtet wird.

Codebeispiel A.1 Log-Datei des `nvprof` Profilers für die Vektoraddition ohne USM von 2^{28} Elementen auf der *GTX 1070 Ti*

```
1  ==29452== NVPROF is profiling process 29452, command: ./vectorAdd 6
2  ==29452== Profiling application: ./vectorAdd 6
3  ==29452== Profiling result:
4              Type  Time(%)      Time     Calls       Avg       Min       Max  Name
5   GPU activities:   67.23%  495.76ms         1  495.76ms  495.76ms  495.76ms  [CUDA memcpy DtoH]
6                     30.45%  224.51ms         2  112.26ms  111.22ms  113.29ms  [CUDA memcpy HtoD]
7                      2.32%  17.141ms         1  17.141ms  17.141ms  17.141ms  vectorAdd(float const *,
                       ↪  float const *, float*, int)
8        API calls:   89.25%  738.51ms         3  246.17ms  111.31ms  513.86ms  cudaMemcpy
9                      9.51%  78.654ms         3  26.218ms  813.04us  76.963ms  cudaMalloc
10                     1.23%  10.166ms         3  3.3887ms  931.45us  5.0365ms  cudaFree
11                     0.01%  84.144us       101     833ns      92ns  36.276us  cuDeviceGetAttribute
12                     0.00%  30.322us         1  30.322us  30.322us  30.322us  cudaLaunchKernel
13                     0.00%  13.832us         1  13.832us  13.832us  13.832us  cuDeviceGetName
14                     0.00%  4.8640us         1  4.8640us  4.8640us  4.8640us  cuDeviceGetPCIBusId
15                     0.00%     854ns         3     284ns     147ns     543ns  cuDeviceGetCount
16                     0.00%     465ns         2     232ns      87ns     378ns  cuDeviceGet
17                     0.00%     402ns         1     402ns     402ns     402ns  cudaGetLastError
18                     0.00%     384ns         1     384ns     384ns     384ns  cuDeviceTotalMem
19                     0.00%     204ns         1     204ns     204ns     204ns  cuDeviceGetUuid
20                     0.00%     193ns         1     193ns     193ns     193ns  cuModuleGetLoadingMode
```

Codebeispiel A.2 Log-Datei des nvprof Profilers für die Vektoraddition mit USM von 2^{28} Elementen auf der *GTX 1070 Ti*

```
==29583== NVPROF is profiling process 29583, command: ./vectorAdd_usm 6
==29583== Profiling application: ./vectorAdd_usm 6
==29583== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  473.05ms         1  473.05ms  473.05ms  473.05ms  vectorAdd(float const *,
 ↪  float const *, float*, int)
      API calls:   60.68%  473.10ms         1  473.10ms  473.10ms  473.10ms  cudaDeviceSynchronize
                   26.70%  208.13ms         3  69.376ms  63.890ms  72.603ms  cudaFree
                   12.60%  98.261ms         3  32.754ms  13.715us  98.219ms  cudaMallocManaged
                    0.01%  89.029us       101     881ns      91ns  38.966us  cuDeviceGetAttribute
                    0.00%  33.089us         1  33.089us  33.089us  33.089us  cudaLaunchKernel
                    0.00%  15.815us         1  15.815us  15.815us  15.815us  cuDeviceGetName
                    0.00%  5.8940us         1  5.8940us  5.8940us  5.8940us  cuDeviceGetPCIBusId
                    0.00%     845ns         3     281ns     131ns     560ns  cuDeviceGetCount
                    0.00%     451ns         2     225ns      95ns     356ns  cuDeviceGet
                    0.00%     358ns         1     358ns     358ns     358ns  cuDeviceTotalMem
                    0.00%     225ns         1     225ns     225ns     225ns  cuModuleGetLoadingMode
                    0.00%     155ns         1     155ns     155ns     155ns  cuDeviceGetUuid

==29583== Unified Memory profiling result:
Device "NVIDIA GeForce GTX 1070 Ti (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   33397  62.794KB  4.0000KB  0.9922MB  2.000000GB  188.0696ms  Host To Device
   18432  170.67KB  4.0000KB  0.9961MB  3.000000GB  262.6451ms  Device To Host
    5051         -         -         -           -  458.3706ms  Gpu page fault groups
Total CPU Page faults: 15360
```

B. Ergebnisse des `nvprof` Profilers für die Mandelbrot-Menge

Die folgenden beiden Log-Dateien zeigen die Ausgabe des `nvprof` Profilers für die Mandelbrot-Menge mit 35 000 Pixeln Bildlänge auf der *GTX 1070 Ti* unter CUDA *mit* (Codebeispiel B.1) und *ohne* (Codebeispiel B.2) USM. Dabei ist vor allem der Abschnitt `GPU activities` wichtig, da hier die eigentliche Arbeit der GPU analysiert und der Abschnitt `Unified Memory profiling result`, in dem der USM betrachtet wird.

Codebeispiel B.1 Log-Datei des `nvprof` Profilers für die Mandelbrot-Menge ohne USM mit 35 000 Pixeln Bildlänge auf der *GTX 1070 Ti*

```
1  ==29902== NVPROF is profiling process 29902, command: ./mandelbrot 6
2  ==29902== Profiling application: ./mandelbrot 6
3  ==29902== Profiling result:
4              Type  Time(%)      Time     Calls       Avg       Min       Max  Name
5   GPU activities:   72.61%  6.00934s         1  6.00934s  6.00934s  6.00934s  calc(int*, unsigned long,
     ↪  unsigned long)
6                     27.39%  2.26720s         1  2.26720s  2.26720s  2.26720s  [CUDA memcpy DtoH]
7        API calls:   71.86%  6.00935s         1  6.00935s  6.00935s  6.00935s  cudaDeviceSynchronize
8                     27.12%  2.26772s         1  2.26772s  2.26772s  2.26772s  cudaMemcpy
9                      0.97%  81.035ms         1  81.035ms  81.035ms  81.035ms  cudaMalloc
10                     0.05%  3.8216ms         1  3.8216ms  3.8216ms  3.8216ms  cudaFree
11                     0.00%  97.185us       101     962ns      90ns  36.368us  cuDeviceGetAttribute
12                     0.00%  25.351us         1  25.351us  25.351us  25.351us  cudaLaunchKernel
13                     0.00%  15.261us         1  15.261us  15.261us  15.261us  cuDeviceGetName
14                     0.00%  5.8390us         1  5.8390us  5.8390us  5.8390us  cuDeviceGetPCIBusId
15                     0.00%     952ns         3     317ns     141ns     639ns  cuDeviceGetCount
16                     0.00%     471ns         2     235ns     101ns     370ns  cuDeviceGet
17                     0.00%     358ns         1     358ns     358ns     358ns  cuDeviceTotalMem
18                     0.00%     197ns         1     197ns     197ns     197ns  cuModuleGetLoadingMode
19                     0.00%     171ns         1     171ns     171ns     171ns  cuDeviceGetUuid
```

Codebeispiel B.2 Log-Datei des `nvprof` Profilers für die Mandelbrot-Menge mit USM mit 35 000 Pixeln Bildlänge auf der *GTX 1070 Ti*

```
==29967== NVPROF is profiling process 29967, command: ./mandelbrot_usm 6
==29967== Profiling application: ./mandelbrot_usm 6
==29967== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  6.18078s         1  6.18078s  6.18078s  6.18078s  calc(int*, unsigned long,
 ↪  unsigned long)
      API calls:   96.30%  6.18079s         1  6.18079s  6.18079s  6.18079s  cudaDeviceSynchronize
                    2.17%  139.57ms         1  139.57ms  139.57ms  139.57ms  cudaFree
                    1.52%  97.526ms         1  97.526ms  97.526ms  97.526ms  cudaMallocManaged
                    0.00%  92.018us       101     911ns      90ns  42.337us  cuDeviceGetAttribute
                    0.00%  19.400us         1  19.400us  19.400us  19.400us  cudaLaunchKernel
                    0.00%  15.414us         1  15.414us  15.414us  15.414us  cuDeviceGetName
                    0.00%  10.126us         1  10.126us  10.126us  10.126us  cuDeviceGetPCIBusId
                    0.00%     900ns         3     300ns     133ns     625ns  cuDeviceGetCount
                    0.00%     480ns         2     240ns      97ns     383ns  cuDeviceGet
                    0.00%     371ns         1     371ns     371ns     371ns  cuDeviceTotalMem
                    0.00%     212ns         1     212ns     212ns     212ns  cuModuleGetLoadingMode
                    0.00%     167ns         1     167ns     167ns     167ns  cuDeviceGetUuid

==29967== Unified Memory profiling result:
Device "NVIDIA GeForce GTX 1070 Ti (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    9347         -         -         -           -  429.4301ms  Gpu page fault groups
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Hildesheim, 5.10.22 F. Heidrich

Ort, Datum, Unterschrift