FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Lösung von Ratengleichungen mit Expression Templates auf der Grafikkarte

Peter Herbst

Bachelorarbeit

Lösung von Ratengleichungen mit Expression Templates auf der Grafikkarte

Peter Herbst

Bachelorarbeit

Aufgabensteller:	Prof. Dr. Christoph Pflaum
Betreuer:	Prof. Dr. Christoph Pflaum
Bearbeitungszeitraum:	1.5.2020 - 30.9.2020

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 28. September 2020 .....

Inhaltsverzeichnis

1 Einleitung

Viele Gesetze der Natur lassen sich mit Differentialgleichungen beschreiben, diese sind aber oft, je nach Komplexitätsgrad, nicht exakt lösbar. Ihre Lösung muss man mit numerischen Methoden, wie zum Beispiel der Finite-Elemente-Methode annähern, welche erst mit steigender Rechenleistung der Computer durchführbar wurde. Heute ist es sehr preiswert solche Berechnungen durchzuführen und demnach verdrängen numerische Methoden die frühere Alternative - kostenaufwendige experimentelle Untersuchungen. Bei der genannten Methode werden die zu simulierenden Strukturen aus endlichen Elementen nachgebaut. Je mehr und feiner die benutzten Elemente gewählt werden, desto genauer wird das Endergebnis gewöhnlich, aber desto länger dauert auch die Berechnung. Insofern setzt man viel daran, die verwendeten Algorithmen so gut wie möglich zu optimieren. [9]

Wie sich auch in dieser Bachelorarbeit herausstellt, ist die Verwendung von GPUs derzeit eine der besten und preiswertesten Möglichkeiten, um die Geschwindigkeit der Berechnungen enorm zu steigern. Dabei werden Grafikkarten erst seit etwa 20 Jahren für allgemeine Berechnungen verwendet, da diese zunächst nur speziell für das Rendern von 3D Grafik konzipiert waren. Aufgrund des beschränkten Anwendungsgebietes war in der Grafikkarte ziemlich genau festgelegt, wie die hereinkommenden Daten verarbeitet werden. Mit der Möglichkeit, sogenannte Shaderprogramme auf die Grafikkarte zu laden, änderte sich das und ab diesem Zeitpunkt konnte man GPUs für neue Zwecke verwenden. Zunächst war das allerdings immer noch relativ aufwändig, da Shaderprogramme nur für Grafikberechnungen gedacht waren, doch der Startschuss für **General Purpose computing on Graphics Processing Units(GPGPU)** war gesetzt. Grafikkarten wurden für massiv parallele Berechnungen konzipiert, und boten nun unerreichte Werte an Rechenleistung pro Energieverbrauch. Möglich wird das durch sehr viele leichtgewichtige Recheneinheiten, die von der Komplexität und den Möglichkeiten her nicht an den Rechenkern einer CPU herankommen, durch ihre große Anzahl aber viel größere Datenmengen gleichzeitig verarbeiten können. Allerdings ist klar, dass die Komplexität für den Programmierer weiter angestiegen ist, da er nun Programme schreiben muss, die auf hunderten bis tausenden Rechenkernen gleichzeitig ausgeführt werden. Dagen hat man bei der CPU Programmierung gewöhnlicherweise nur eine einstellige Anzahl an Kernen hat und pro Kern wenige Threads am laufen. Der Entwickler steht nun vor der Aufgabe, die Datenmengen sinnvoll zu unterteilen und dadurch effizient zu verarbeiten. Außerdem sind die Speicher der GPU und CPU gewöhnlicherweise getrennt, weswegen sich der Programmierer auch noch zusätzlich darum kümmern muss, die Daten zum richtigen Zeitpunkt hin und her zu kopieren. Dabei muss auch stark darauf geachtet werden, unnötig viele Kopieoperationen zu vermeiden, da Datentransfer nach wie vor einer der großen Flaschenhälse für die Performance ist. Es gibt bereits Ansätze, diese Problematik zu entschärfen, beispielsweise entwickelt AMD inzwischen **Accelerated Processing Units(APU)**, bei denen sich GPU und CPU auf einem Chip befinden und auf den selben Speicher zugreifen können. Nvidia entwickelte Unified Memory, eine Technik bei der die Speichersynchronisation im Hintergrund automatisch geschieht. [1]

Im Rahmen dieser Bachelorarbeit wird die numerische Lösung von Differentialgleichungen mit Hilfe von Expression Templates und Grafikkarten erörtert. Dafür wird die Bibliothek UGBlocks, welche bereits gewöhnliche Expression Templates[13] verwendet, erweitert, sodass Teile der Berechnung auf der Grafikkarte stattfinden. Als Orientierungshilfe diente hierfür der Code von Knauer[8]. Stellenweise wurden Strukturen übernommen, wie zum Beispiel Lambdafunktionen als Kernel, andere Codestücke wurden angepasst oder ganz verworfen. Vor allem die Datenstruktur zur Verwaltung des Grafikkartenspeichers wurde abgeändert. Zusätzlich wurde in einem Alternativversuch die Speicherverwaltung komplett mit Unified Memory vereinfacht, da sich dies im konkreten Anwendungsfall sehr angeboten hat. Vorallem die Tatsache, dass die Daten hauptsächlich nur im Grafikkartenspeicher gehalten werden müssen, ließ eine nicht viel schlechtere Performance als manuelle Speicherverwaltung vermuten. Um die verschiedenen Versionen zu vergleichen und bewerten zu können, ob die Verwendung von Unified Memory sinnvoll ist, wurden umfassende Messungen der Rechenleistung durchgeführt.

2 Grundlagen

2.1 Ratengleichungen

Um die erfolgreiche Erweiterung von UGBlocks zu testen, wird die Bibliothek verwendet, um eine Problemstellung der Laserphysik numerisch zu lösen. Die Funktion von Lasern basiert auf stimulierter Emission. Atome können sich in Zuständen unterschiedlicher Energie befinden. Stimulierte Emission tritt auf, wenn ein Photon auf Materie in einem energetisch hohen Zustand trifft und das Atom dadurch in den niedrigeren Zustand wechselt. Die Energie geht nicht verloren, stattdessen wird bei diesem Prozess ein Photon emittiert, das in allen Eigenschaften identisch zum eintreffenden Photon ist. Durch diesen Effekt wird eine Lichtverstärkung erreicht. In einem Zwei-Niveau-System gibt es zwei diskrete Energieniveaus, die angenommen werden können. Wir beschreiben die Anzahl der Teilchen, welche sich im oberen Energieniveau E_2 befinden mit N_2 und die im unteren Energieniveau E_1 mit N_1. Damit tatsächlich der Verstärkungseffekt auftritt, muss $N_2 > N_1$ sein, was auch Besetzungsinversion genannt wird. Im thermischen Gleichgewicht kann man mit der Boltzmann Verteilung zeigen, dass keine Besetzungsinversion erreicht werden kann. Man muss Energie ins System pumpen. Ohne auf die genauen Details einzugehen, sei noch erwähnt, dass eine Besetzungsinversion im Zwei-Niveau-System nicht erreicht werden kann. Es werden mindestens drei Niveaus benötigt.[10]

Letztendlich können mit Ratengleichungen der Verlauf der Besetzung und die Photonenanzahl beschrieben werden. In Gleichungen (1) und (2) ist $N = N_2 - N_1$.

Die Gleichungen und die Verwendete Lösungsmethode Dynamic Mode Analysis werden in [15] beschrieben.

$$\frac{dN}{dt} = -\frac{\sigma * c}{V_{eff}} * N * \phi * U_{mode} - \frac{N}{\tau_f} + R_{pump} * \frac{N_{tot} - N}{N_{tot}} \tag{1}$$

$$\frac{d\phi}{dt} = \frac{\sigma * c}{V_{eff}} * \phi * \sum_{c=cell} (cellVol(c) * N(c) * U_{mode}(c)) - \frac{\phi}{\tau_c} \tag{2}$$

$U_{mode}, N, N_{tot}, R_{pump}$ und cellVol werden im Code durch jeweils eine Zellvariable dargestellt, die für jede Zelle im Gitter einen Wert abspeichert. Die Summation über die Zellen in Gleichung (2) stellt ein Skalarprodukt über das gesamte Gitter dar, d.h. jeder einzelne Gitterpunkt in cellVol wird mit den Gitterpunkten von N und U_{mode} multipliziert und die Ergebnisse werden aufsummiert. Für die numerische Lösung müssen die Gleichungen zunächst noch diskretisiert werden. Danach kann man die Lösung iterativ berechnen. Der hierfür verwendete Code ist in Listing 1 aufgeführt.

```
1  //Physikalische Konstanten
2  double  hplanck = 6.62607004e-34; // m2 kg / s
3  double  cLight = 3.0e8 * 1000;    // [mm/s]
4
5  // Parameter
6  double pumpWavelength = 808e-9;   // [mm]
7  double laserWavelength = 1064e-9; // [mm]
8  double pumpRadius = 0.4;
9  double sigma = 2.5e-17;           // [mm^2]
10 double length = 300;              // [mm]
11 double t_end = 0.0005;            // [s]
12 double tau_f = 0.00002;           // [s]
13 double doping = 1.39E17;          // [ions/mm^3]
14 double S = 1.0e-10;
15 double Rout = 0.98;
16 double pumpPower = 10000.0;
17
18 //Abgeleitete Parameter
19 double t_roundTrip = length / cLight * 2.0;
20 double sigma_by_roundTrip = sigma * cLight / length;
21 double tau_c = -t_roundTrip / log(Rout);
22 double factorOutpoutPower = (1.0 - Rout) * (hplanck * cLight / laserWavelength);
23
24 Cylinder crystalGeometry(0.5, 0.2, 10.0, true);
25
26 Blockgrid blockgrid(&crystalGeometry, n);
27 Cell_variable<double>  Npopinv(blockgrid, !Cuda);
28 Cell_variable<double>  NpopinvMax(blockgrid, !Cuda);
29 Cell_variable<double>  pumping(blockgrid, !Cuda);
```

```
30   Cell_variable<double>  mode(blockgrid, !Cuda);
31   Cell_variable<double>  cellVol(blockgrid, !Cuda);
32   Cell_variable<double>  speicher(blockgrid, !Cuda);
33   Cell_variable<double>  XCoord(blockgrid, !Cuda);
34   Cell_variable<double>  YCoord(blockgrid, !Cuda);
35   Cell_variable<double>  ZCoord(blockgrid, !Cuda);
36
37   Npopinv = 0.0;
38   NpopinvMax = doping;
39   double Phi = 0.001;
40   cellVol = 1.0;
41
42   // local stiffness matrix for cell volumne
43   Local_stiffness_matrix<double> lsm(blockgrid);
44   lsm.Calculate(v_() * w_());
45   cellVol.IntegrateOnCell(lsm);
46
47 ...
48   //Pumpprofil berechnen(=R_pump in Gleichung (1))
49   Function2<double> pumpingProfile(pumpProfile);
50   pumping = pumpingProfile(X * X + Y * Y, Z);
51   pumping = pumping * pumpPower * (1 / (hplanck * cLight / pumpWavelength));
52
53   //Modeprofil berechnen(=U_mode in Gleichung (1) und (2))
54   Function1<double, double> mode_profile(modeProfile);
55   mode = mode_profile(X * X + Y * Y);
56
57   //tau = Zeitschritt
58   double tau = t_end / iteration;
59   for (int i = 0; i < iteration; ++i) {
60
61     //Iterationsschritt fuer N
62     Npopinv = (Npopinv + tau * pumping) / (1.0 + tau * (1.0 / tau_f +
       sigma_by_roundTrip * Phi * mode + pumping / NpopinvMax));
63
64     //Summation aus Gleichung(2)
65     speicher = Npopinv * mode;
66     double factor = product_cell(cellVol, speicher);
67
68     double lambda = sigma_by_roundTrip * factor - 1.0 / tau_c;
69     // photons
70     if (lambda > 0.0) Phi = Phi * (1.0 + tau * lambda) + tau * S;
71     else              Phi = (Phi + tau * S) / (1.0 - tau * lambda);
72
73     //Ausgabe der Power zum Zeitschritt i
74     DATEI << i * tau << "    " << factorOutpoutPower * Phi << std::endl;
75   }
```

Listing 1: UGBlocks Code zur Lösung der Ratengleichungen.

2.2 GPU Programmierung

2.2.1 Allgemein

Grafikkarten waren ursprünglich, wie der Name auch andeutet, zusätzliche spezialisierte Hardware, die den Hauptprozessor bei der Bildberechnung für (vor allem) Computerspiele unterstützen sollte. Zunächst konnte man den Ablauf dieser Berechnungen nicht steuern, später aber unterstützten die Grafikkarten programmierbare Shader, welche zwar nur für Bildberechnung gedacht waren, aber auch alternativ benutzt werden konnten, da die Recheneinheiten der Grafikkarten einen ausreichenden Befehlssatz hierfür besaßen. Erste Berechnungen in der linearen Algebra bzw. Physiksimulation wurden auf Grafikkarten demonstriert. Leider gab es hierfür keine spezielle Bibliothek, mit der man gezielt allgemeine mathematische Operationen auf der Grafikkarte durchführen konnte. Man musste sein wissenschaftliches Problem passend für die Grafikkarte umwandeln. Anstatt einen Datenpuffer zu erzeugen und dort die Eingabedaten hineinzuschreiben, hat man diese als Farbwerte in einer Textur, welche vom Shader gelesen werden kann, abgelegt. Außerdem mussten die Texturen und der Shader auf die Grafikkarte geladen werden, woraufhin das Rendern von Geometrie angestoßen werden konnte. GPGPU war also nur erfahrenen Grafikentwicklern vorbehalten, die fundiertes

Wissen im Umgang mit GPUs hatten. Die Programmiersprache **Brook** vereinfachte diesen Prozess deutlich und ermöglichte einer breiteren Masse, Berechnungen auf GPUs durchzuführen. Mit ihr konnte man seinen Algorithmus in einer leichteren Form beschreiben, ohne die genauen Details von Shaderprogrammierung zu kennen. Zunächst definierte man seine Datenstreams und befüllt diese, woraufhin sie parallel bei einer Kernelausführung verarbeitet werden. Dieses Grundkonzept von Datenstreams und Kernel sind bei heutigem GPGPU erhalten geblieben. Bei der Kompilierung eines Brook Programms werden die Kernel in Shader übersetzt und zusätzlich wird Code eingefügt, der die Shader ausführt.[2]

Man kann wohl sagen, Brook war eine wichtige Etappe auf dem Weg zu GPGPU, wie wir es heute kennen, wird aber nicht mehr verwendet, da die Grafikkartenhersteller inzwischen einfachere Möglichkeiten geschaffen haben, auch nicht grafikbezogene Berechnungen auf ihren GPUs durchzuführen. Dafür gibt es nun einfachere Standards, wie **OpenCL**, **SYCL** oder **CUDA**. Letzterer wird in dieser Arbeit verwendet. Dort schreibt man sogenannte Kernel, welche von verschiedenen Threads auf der Grafikkarte ausgeführt werden. Diese Threads sind organisiert in Warps und Blöcke, was in späteren Kapiteln genauer erklärt wird.

2.2.2 Die Hardware

Um **effiziente** Programme für GPUs zu schreiben, sollte man die zugrunde liegende Hardware kennen, immerhin unterscheidet sich die Graphics Processing Unit trotz ähnlicher Namensgebung doch deutlich von der Central Processing Unit. Die gewöhnlichen Programmierkonzepte verlieren ihre Gültigkeit für Grafikkartenprogrammierung. Heutzutage besitzen so gut wie alle Prozessoren mehr als einen Rechenkern, welcher jeder für sich vollständig autonom arbeiten kann, also eigene Kontrolllogik usw. besitzt. Während CPUs eine einstellige bis niedrig zweistellige Zahl an Kernen besitzen, geht die Kernanzahl bei GPUs bis in die tausende, was nur möglich ist, weil diese simpler aufgebaut sind und weniger Möglichkeiten bieten. Die große Anzahl an GPU Kernen ermöglicht aber deutlich mehr Berechnungen pro Sekunde, ohne dabei viel mehr Energie als eine CPU zu verbrauchen. Die Kennzahlen GFLOPS/s (Milliarden Gleitkommazahloperationen pro Sekunde) und GFLOPS/W sind viel größer als bei CPUs. [1]

In Abbildung 1 sieht man, dass bei CPUs deutlich mehr Transistoren für Caches, und Kontrolllogik verwendet werden, bei GPUs dagegen wird viel Transistorfläche für Recheneinheiten verwendet, um große Mengen an Daten gleichzeitig verarbeiten zu können.

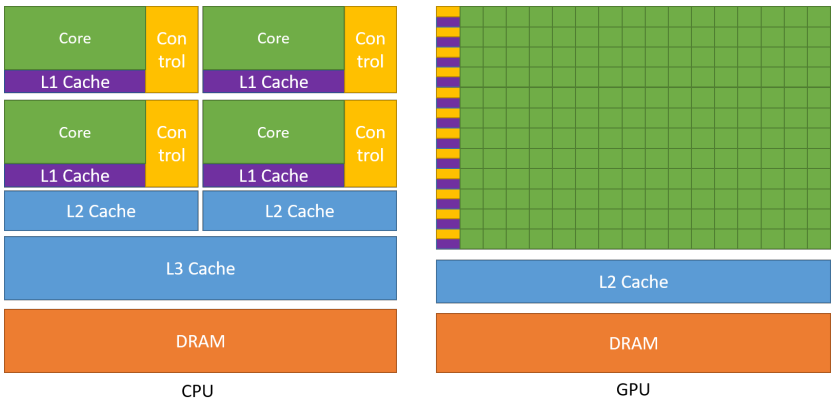

Abbildung 1: Vergleich zwischen CPU und GPU.[11]

Essenziell kann man sagen, die Grafikkarte ist gut für datenparallele und rechenintensive Aufgaben. Durch massiv parallele Ausführung vieler leichtgewichtiger Recheneinheiten kommt die GPU bestens mit großen Datenmengen zurecht. Aufgrund der Art der Problemstellungen ist zumeist kaum Synchronisation unter den Threads nötig, da diese gewöhnlicherweise unterschiedliche Portionen des Gesamtdatensatzes verarbeiten. Sollten doch einmal Reduktionen oder ähnliches durchgeführt werden müssen, besitzen die Threads trotzdem ausreichende Synchronisierungsmöglichkeiten.

Bei Nvidia sind die für uns wichtigsten Einheiten auf einem GPU Chip die **Streaming Multiprocessors**(SMs), welche sogenannte Blöcke, die in **Warps** unterteilt werden, verwalten und

ausführen. Dabei ist ein Warp jeweils eine Gruppe von 32 Threads, welche gleichzeitig ausgeführt werden müssen. Verschiedene Warps werden dagegen nicht zwangsweise parallel ausgeführt. Tatsächlich führt jeder Thread im Warp zur gleichen Zeit die selbe Instruktion aus, sogar wenn gewisse Threads datenabhängig einen anderen Codezweig einschlagen müssen(z.B. wegen einer If-Abfrage), als die anderen Threads. Threads die nicht im gleichen Codezweig gelandet sind, werden bis zur Wiedervereinigung des Zweiges deaktiviert. Für die Performancemaximierung ist es deswegen wichtig, dafür zu sorgen, dass Threads in einem Warp die selben Operationen ausführen. So passiert es selten, dass Threads deaktiviert werden und Rechenleistung unbenutzt bleibt. Außerdem ist es vorteilhaft, die Threads eines Warps nebeneinanderliegende Speicheradressen verwenden zu lassen, da bei einem Datenzugriff nebeneinanderliegende Daten zusammen abgerufen werden. Warps sind ohne große Kosten austauschbar, der SM kann also einen Warp, der gerade auf Daten wartet und deswegen nichts rechnen kann, schnell durch einen anderen austauschen. So wird dafür gesorgt, dass die Recheneinheiten eines SMs stets ausgelastet sind. [11] Für die Performancemessungen in dieser Arbeit wurden zwei unterschiedliche Grafikkarten getestet:

- **GeForce GTX 1070 Ti** - 19 Streaming Multiprocessors mit jeweils 128 CUDA Cores(insgesamt 2432 CUDA Cores)
- **Quadro T1000** - 14 Streaming Multiprocessors mit jeweils 64 CUDA Cores(insgesamt 896 CUDA Cores)

2.2.3 Konzepte

Für die nachfolgenden Kapitel werden hier einige Standardkonzepte erläutert. Prinzipiell unterscheidet man im Kontext von GPU Programmierung zwischen **Hostcode** und **Devicode**. Hostcode meint den Code, welcher auf dem Hauptprozessor ausgeführt wird, Devicecode wird auf der GPU ausgeführt. Die Bezeichnung Host kommt daher, dass eine GPU nicht ohne CPU benutzt werden kann, da sie lediglich einen Koprozessor darstellt. Der Hauptprozessor ist also der "Host", auf Deutsch Gastgeber. Ähnlich bezeichnet man auch den Arbeitsspeicher der Grafikkarte als **Devicespeicher** und den des Hauptprozessors als **Hostspeicher**. Speicheradressen des Hostspeichers bzw. Devicespeichers können nicht einfach sowohl im Hostcode als auch Devicecode verwendet werden, außer beispielsweise mit **Unified Memory** von Nvidia, was aber später noch genauer erklärt wird.[11]

2.3 Expression Templates

2.3.1 C++ Templates

Expression Templates[13] sind eine Programmiertechnik für C++, die stark auf der Verwendung von sogenannten **Templates** basiert.

Sämtliche Variablen, Funktionsparameter und Rückgabewerte von Funktionen müssen in C++ mit spezifischen Typen deklariert werden. Teilweise sind Algorithmen für unterschiedliche Datentypen trotzdem sehr ähnlich, ein Beispiel hierfür ist die verkettete Liste. Ob man in dieser Liste Ganzzahlen(int) oder Gleitkommazahlen(float, double) ablegt, ist für die Art und Weise der Implementation relativ egal, aber nicht für den Compiler. Dieser muss den in der Liste abgelegten Datentyp nämlich explizit wissen, um den Code hierfür zu generieren. Man hat mehrere Möglichkeiten, die verkettete Liste für verschiedene Datentypen zu implementieren. Beispielsweise kann man den Code einer Liste komplett kopieren und lediglich den Namen der Klasse verändern, sowie den verwendeten Datentyp. Diese Lösung bringt viel Codeduplikation mit sich, was im Allgemeinen schlecht ist. Tritt ein Fehler bei der Liste für Ganzzahlen auf, wird er auch bei der Liste für Gleitkommazahlen vorhanden sein und man muss ihn in beiden Codeexemplaren verbessern. Ohne weitere Methoden aufzulisten, die unser Problem ebenfalls nicht besonders schön lösen, kommen wir direkt zur besten Lösung: Templates. Mit Templates kann man Funktionen und Klassen mit einem oder mehreren Templateparameter definieren, wodurch man die Möglichkeit hat, Code für einen noch **unspezifizierten** Datentypen zu schreiben. Der definierte Templateparameter dient also als Platzhalter für einen später konkret verwendeten Typen. Bei der Verwendung einer solchen templatisierten Funktion/Klasse muss in einer spitzen Klammer definiert werden, welcher Typ in diesen Platzhalter eingesetzt werden soll. Der Compiler übersetzt die Klasse/den Algorithmus nur

für den eingesetzten Typen.[7] Dabei kann es durchaus passieren, dass die Kompilation für einen Typen funktioniert, für einen anderen aber nicht.

```
1 #include <iostream>
2 template <class A>
3 void calculateAndPrint(A input){
4   input.Calculate();
5   input.Print();
6 }
7
8 class KonkreterTypOK{
9 public:
10   void Calculate(){
11     val = 5.0;
12   }
13   void Print(){
14     std::cout << val;
15   }
16 private:
17   double val;
18 };
19 class KonkreterTypError{
20 public:
21   void Calculate(){
22     val = 6.0;
23   }
24 private:
25   double val;
26 };
27
28 int main(int argc, char** argv){
29   KonkreterTypOK a = KonkreterTypOK();
30   KonkreterTypError b = KonkreterTypError();
31   calculateAndPrint<KonkreterTypOK>(a);
32   //Ohne diese Zeile kompiliert das Programm
33   calculateAndPrint<KonkreterTypError>(b);
34 }
```

Listing 2: Beispiel zu Templateprogrammierung.

Der Code in Listing 2 wird nicht kompilieren, da für den Templateparameter A(Zeile 2-5) eine Klasse eingesetzt wird, welche keine **Print** Funktion definiert. Ohne die Zeile 33 würde alles problemlos übersetzt werden.

2.3.2 Vorteile von Expression Templates

Um Datenstrukturen miteinander zu verrechnen, bietet sich in C++ das Überladen von Operatoren an. Ein gewöhnlicher Anwendungsfall wäre das Addieren zweier Vektoren. Hier kann man den Plus Operator überladen, sodass dieser, wenn er zwischen zwei Vektorklassen verwendet wird, diese Vektoren addiert und einen neuen Vektor zurückliefert. Auf diese Weise gewinnt man viel Komfort für die Programmierung, verliert aber potenziell einiges an Performance. Vor allem im Bereich der Hochleistungsrechnung benutzt man deswegen kaum Operatorüberladung. Der Performanceverlust wird am Bespiel folgender Berechnung erklärt:
a = b + c + d
Dabei stellen a, b, c und d Objekte einer Klasse Vektor dar. Wir haben für diese Klasse den + Operator überladen, welcher in seinem Coderumpf zwei Vektoren komponentenweise addiert und als Resultat einen neuen Vektor zurückgibt. Es wird also für unser konkretes Beispiel bei der Berechnung von b + c ein temporärer Vektor erzeugt, welcher mit d verrechnet wird. Das Ergebnis ist wiederum ein temporärer Vektor, der a zugewiesen wird. Für die gesamte Berechnung werden also zwei temporäre Vektoren erzeugt und dreimal in einer Schleife sämtliche Komponenten der Vektoren durchlaufen. Wünschenswert wäre es, bei der Berechnung einmal in einer Schleife alle Komponenten durchzulaufen und dem n-ten Komponenten von a direkt die Summe der n-ten Komponenten von b, c und d zuzuweisen. [7] Dieser Ansatz wird unter anderem durch Expression Templates umgesetzt. Mit ihnen kann man immer noch bequem die Vorteile von Operatorüberladung genießen, erhält aber zugleich bessere Performance.

2.3.3 Implementation

Um das gewünschte Verhalten zu erhalten, werden die auf der Datenstruktur durchgeführten mathematischen Operationen in einem Ausdrucksbaum abgelegt, welcher zu einem späteren Zeitpunkt ausgewertet werden kann. Solche Bäume können durch die Verwendung von abstrakten Klassen und virtuellen Funktionen konstruiert werden, bieten dann aber nicht die gewünschte Performance, da der Compiler die Funktionsaufrufe in diesem Baum schlecht optimieren kann. Der Typ eines Elements in solch einem Baum steht nämlich zur Übersetzungszeit noch nicht fest, wodurch das für die Performance wichtige inlining der Funktionen nicht möglich ist. Bei der Verwendung von **Expression Templates** stehen die Typen im Baum jedoch schon zur Übersetzungszeit fest, weshalb der Compiler effizienten Maschinencode erstellen kann. Im Listing 3 findet sich der von Härdtlein entwickelte Code einer einfachen Expression Template Variante[7].

```
1  template <class E>
2  struct Expr{
3    operator const E& () const {
4      return *static_cast<const E*>(this);
5    }
6    double give(int i) const {
7      return static_cast<const E&>(*this).give(i);
8    }
9  };
10
11 template <class L, class R>
12 class Addition : public Expr<Addition<L,R>>{
13   const L& l;
14   const R& r;
15 public:
16   Addition (const L& l_, const R& r_) : l(l_), r(r_){}
17   double give(int i) const { return l.give(i) + r.give(i); }
18 };
19
20 template <typename L, class R>
21 class ScalarMultiplication
22   : public Expr<ScalarMultiplication<L,R>>{
23   const L l;
24   const R& r;
25 public:
26   ScalarMultiplication (const L& l_, const R& r_) : l(l_), r(r_) {}
27   double give(int i) const {return l * r.give(i); }
28 };
29
30 template <class L, class R>
31  inline Addition<L,R>
32   operator+(const Expr<L>& l, const Expr<R>& r){
33     return Addition<L,R>(l,r);
34   }
35
36 template <class R>
37  inline ScalarMultiplication<double, R>
38   operator*(double l, const Expr<R>& r){
39     return ScalarMultiplication<double,R>(l,r);
40   }
41
42 class Vector : public Expr<Vector>{
43   ...
44   template<class E>
45   void operator=(const Expr<E>& e){
46     for(int i=0; i<size; ++i)
47       data[i] = e.give(i);
48   }
49 };
```

Listing 3: Einfache Implementation von Expression Templates.[7]

Diese Implementation wurde am Beispiel einer Vektor Klasse durchgeführt und teilt sich nach Härdtlein in vier prinzipielle Einheiten auf:

- Eine oder mehrere **Wrapperklassen**(**struct Expr**).

- **Grunddatenstrukturen** welche Blätter(Terminale) im Ausdrucksbaum bilden. Diese müssen entsprechende Zugriffs-(**Vector::give**, nicht im Code dargestellt) und Zuweisungsmethoden enthalten.
- **Operationsklassen**(**Addition** und **ScalarMultiplication**), die ihre Operanden(**L** und **R**) speichern und die Operation durchführen können(**Addition::give** und **ScalarMultiplication::give**). Die Typen der Operanden sind als Templateparameter definiert.
- **Creator Funktionen** und überladene Operatoren(**+ und * Operator**), die dem Benutzer die einfache Erzeugung von Ausdrucksbäumen erlauben.

Zunächst einmal sollte die Notwendigkeit der **Wrapperklasse** erklärt werden. Die überladenen Operatoren akzeptieren nur Klassen die in diesem Wrapper gekapselt sind als Parameter. So wird verhindert, dass die definierten Operationen auf jegliche Typen angewandt werden können. Erwähnenswert ist auch, dass man mehrere Wrapper verwenden kann, um zum Beispiel unterschiedliche Datenstrukturen zu gruppieren. Man könnte eine Klasse für Matrixartige und eine für Vektorartige Strukturen definieren. Dadurch kann man genauer festlegen, welche Ergebnisse die Operationsklassen zurückliefern. Eine Operationsklasse, die Beispielsweise eine Matrix mit einem Vektor multipliziert, bekommt einen Vektorwrapper, da das Ergebnis der Operation ein Vektor ist. Die Grundlegende Datenstruktur(im Beispielcode die Klasse Vector) muss nicht zwingend vom Wrapper erben, dadurch wird aber zusätzlicher Codeaufwand verhindert. Ohne die Wrapperklasse müsste man die Operatoren jeweils vier mal überladen und zwar für die Operanden Vector-Vector, Vector-Expr<R>, Expr<L>-Vector und Expr<L>-Expr<R>. Die Auswertung eines Ausdrucksbaumes sollte in einer Grunddatenstruktur implementiert werden. Im Beispiel wurde das mit dem überladen des operator= erreicht.

Die Operationsklassen beinhalten ihre Operanden typischerweise als Referenz, um viele aufwändige Kopieoperationen zu vermeiden. In den Methoden der Operationsklasse, die den Wert an einem bestimmten Index im Gitter zurückgeben(hier: give), befindet sich der Code, welcher die Berechnung durchführt.

```
1  Vector a, b, c, result;
2  ... //initialize
3
4  //Auswertung mit Expression Templates
5  //Expr<Addition<Addition<Vector, Vector>, Vektor>>
6  result = a + b + c;
7
8  //Manuelle Auswertung
9  for(int i = 0; i < result.size(); i++)
10   result.data[i] = a.data[i] + b.data[i] + c.data[i];
```

Listing 4: Auswertung mit Expression Templates und manuelle Auswertung.

In Listing 4. ist beispielhaft die Auswertung einer Vektoraddition mit Expression Templates und mit einer for-Schleife dargestellt. Abbildung 2 enthält die Baumartig aufgebaute Struktur der Expression, sowie eine Skizze der konkret Instanziierten Klassen mit den in die Templateparameter eingesetzten Typen. Bei der Auswertung des Ausdrucks wird die give Funktion der Baumwurzel aufgerufen, welche selbst wiederum die give Funktion ihrer beiden Äste aufruft und die Werte addiert zurückgibt. Die Aufrufkette endet bei der Grunddatenstruktur Vector, dieser wertet keinen Ausdruck mehr aus, sondern gibt einfach den Wert seines i-ten Komponenten zurück. Aufgrund der zur Laufzeit festgelegten Typen kann der Compiler die Auswertung des Baumes so optimieren, dass sich der erzeugte Maschinencode wenig vom Maschinencode der manuellen Auswertung unterscheidet.

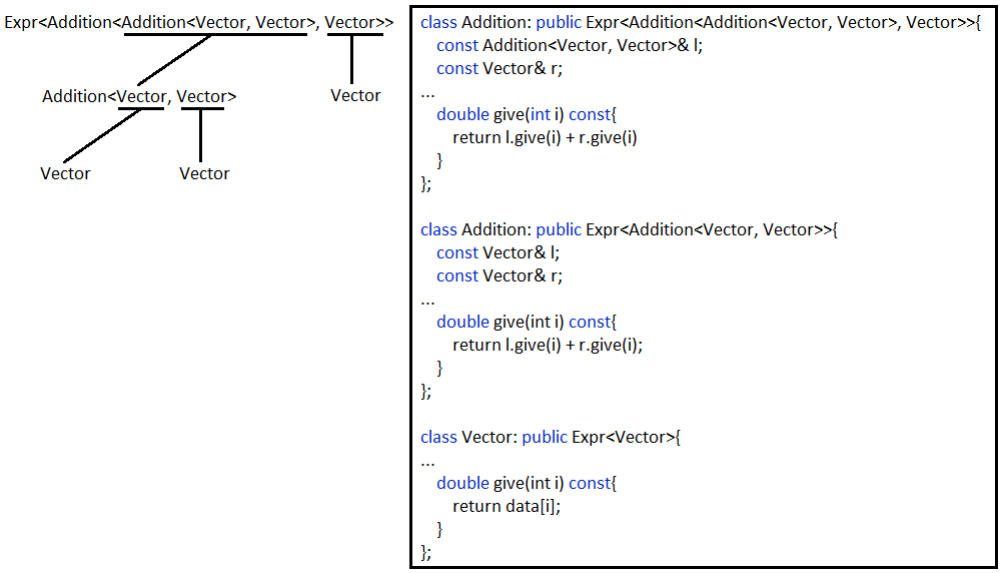

Abbildung 2: Ausdrucksbaum mit konkreten Implementationen der Klassen.

3 CUDA Programmierung

3.1 Allgemein

Bei CUDA werden im C++ Code diejenigen Funktionen, welche auf der Grafikkarte ausgeführt werden, mit dem Präfix **__device__** gekennzeichnet. Funktionen für die CPU werden mit **__host__** und sogenannte Kernel mit **__global__** gekennzeichnet. Ein Kernel ist der Einstiegspunkt von der CPU in die GPU, da Kernel die einzigen auf der Grafikkarte ausgeführten Funktionen sind, die von der CPU aus gestartet werden. Um also eine Berechnung auf der Grafikkarte anzustoßen, wird ein Kernel parametrisiert mit einer **Griddimension** und **Blockdimension** gestartet. Diese Dimensionen sind vom Programmierer wählbar und richten sich üblicherweise nach der Problemstellung. Dabei beschreibt die Blockdimension die Anzahl an Threads pro Block und die Griddimension die Gesamtzahl an Blöcken. Ein Grid besteht also aus mehreren Blöcken und ein Block besteht aus mehreren Threads. Um die Gesamtzahl an Threads auszurechnen multipliziert man die Threads pro Block(Blockdimension) mit der Anzahl an Blocks pro Grid(Griddimension). In Abbildung 3 sieht man ein Grid mit sechs Blöcken und zwölf Threads pro Block, wobei die Blöcke und Threads sogar zwei- statt eindimensional organisiert sind. Bei CUDA lassen sich die Block- und Gridgröße in bis zu 3 Dimension aufteilen.[3]

Der Code eines Kernel wird für jeden Thread genau einmal ausgeführt, wobei jeder Thread seine eigene Id innerhalb des Blockes und jeder Block seine eigene Id innerhalb des Grids erhält. Wenn das Grid oder der Block zweidimensional aufgebaut ist, ist diese Id auch zweidimensional. Durch den Zugriff auf diese Id innerhalb des Devicecodes(mit **threadIdx** und **blockIdx**) kann man steuern, welchen Teil der Daten der jeweilige Thread bearbeitet. Um dieses Prinzip zu verdeutlichen, findet sich im folgenden ein extrem simpler Kernel, der zwei Matrizen addiert.

```
1 //Starten des Kernels:
2 addMatrix<<<Griddimension, Blockdimension>>>(a, b, buffer, n, m);
3 __global__ addMatrix(double** matA, double** matB, double** result, int n, int m)
4 {
5   //Verhindert unzulaessigen Speicherzugriff
6   if(threadIdx.x >= n || threadIdx.y >= m) return;
7   int index = threadIdx.x + n * threadIdx.y;
8   result[index] = matrixA[index] + matrixB[index];
9 }
```

Dieser Kernel ist nicht wirklich praktikabel, da er nur mit einem einzigen Block gestartet werden kann/sollte, was man daran sieht, dass **blockIdx** nicht verwendet wird. Würde man den Kernel also mit zwei Blöcken starten, rechnen beide mit den selben Daten. Da lediglich ein Block benutzt werden kann, ist es auch nicht möglich, Matrizen zu addieren, die mehr Zellen haben, als maximal mögliche

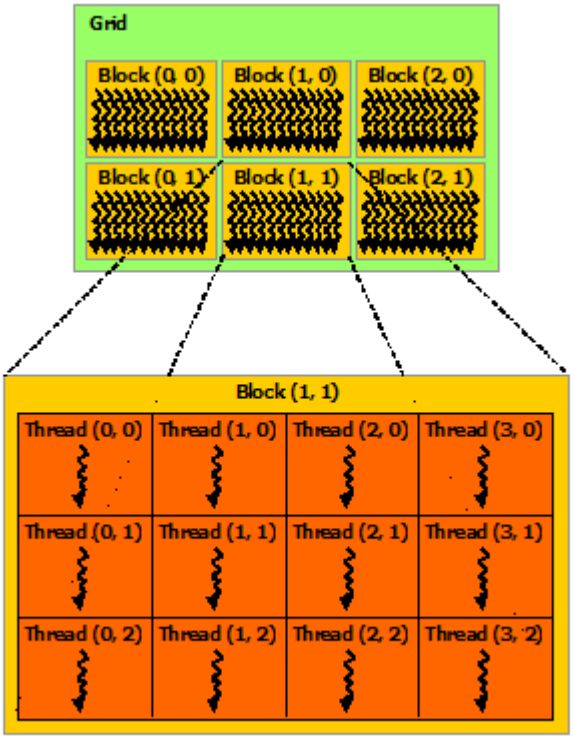

Abbildung 3: Schemenhafter Aufbau eines Grids.[11]

Anzahl an Threads pro Block. Diese Zahl liegt für aktuelle Grafikkarten bei 1024 Threads pro Block, kann und sollte aber mit der **CUDA API** im Code abgefragt werden. Aufgrund der Abfrage in Zeile vier kann es nicht passieren, dass im Kernel auf Speicherbereich außerhalb der Matrizen zugegriffen wird und das Programm abstürzt. In unserem Fall ist die Abfrage nicht zwingend notwendig, wenn der Kernel genau mit der Dimension der Matrizen aufgerufen wird. Ein weiterer großer Nachteil des obigen Kernels ist die ausschließliche Erzeugung eines einzigen Blockes, weil somit weder die maximale Anzahl an Threads pro SM erreicht wird, noch jeder Streaming Multiprocessor einen Block zur Bearbeitung erhält.[11] Das hat einen niedrigen Occupancy Wert und eine schlechte Auslastung der zur Verfügung stehenden Rechenleistung zur Folge. Die Bedeutung von Occupancy wird in einem späteren Kapitel erklärt.

Im Allgemeinen ist der CUDA Anteil eines Programmes so strukturiert:

- 1. Reservieren des GPU Speichers
- 2. Daten aus dem CPU Arbeitsspeicher in den GPU Arbeitsspeicher kopieren
- 3. Starten des Cuda Kernels
- 4. Daten von GPU zurück auf CPU Arbeitsspeicher kopieren
- 5. GPU Speicher freigeben

[3]

3.2 nvcc

Für CUDA wurde der Compiler nvcc entwickelt. Dieser unterteilt Quelldateien nach Devicecode und Hostcode. Der Devicecode wird durch nvcc entweder in PTX Code oder direkt in Binärcode für die GPU übersetzt. Die im Hostcode verwendete Syntax, um einen Kernel zu starten(<<< ... >>>), stellt lediglich eine Vereinfachung für den Programmierer da. Um letztendlich einen Kernel zu starten, muss dessen Binärcode auf die Grafikkarte geladen und eine Ausführung mit entsprechenden Parametern angestoßen werden. Deswegen wird diese Syntax im Hintergrund von nvcc in Aufrufe

der CUDA Runtime umgewandelt. Tatsächlich könnte man diese selbst auch verwenden und auf die spezielle Syntax verzichten. Letztendlich wird der verbliebene Hostcode von nvcc an den Standardcompiler weitergereicht und kompiliert. Am Ende werden sämtliche erzeugten Objektdateien miteinander verlinkt.[11]

3.3 Speicherverwaltung

Ein großes Thema bei der Grafikkartenprogrammierung ist die Speicherverwaltung. Prinzipiell gibt es verschiedene Arten von Speicher:

- **Lokaler Speicher**, auf den nur der Thread selbst zugreifen kann.
- **Shared Memory**(geteilter Speicher), der von allen Threads eines Blockes verwendet werden kann.
- **Globaler Speicher** auf den sämtliche Threads aller Blöcke zugreifen können.

Bei CUDA kann man Speicher auf der Grafikkarte mit **cudaMalloc** und **cudaFree** erzeugen bzw. freigeben. Ein Kernel kann prinzipiell nur auf Speicheradressen zugreifen, die auf Grafikkartenspeicher verweisen. Das übliche Prozedere ist es, die benötigten Daten mit **cudaMemcpy** vom Hostspeicher auf den Devicespeicher zu kopieren, die Berechnungen durchzuführen, das Ergebnis in Devicespeicher abzulegen und dieses danach mit **cudaMemcpy** vom Devicespeicher auf den Hostspeicher zu kopieren. Um die maximale Performance zu erreichen, muss der Programmierer hier sehr genau aufpassen und unnötige Kopien vom Host- zum Devicespeicher bzw. vom Device- zum Hostspeicher vermeiden. Eventuell wird beispielsweise das Ergebnis eines Kernels nicht direkt danach von der CPU benötigt, sondern in weiteren Kernelaufrufen verarbeitet, weswegen man mit der Speichermigration noch warten sollte. Es ist am sinnvollsten, die Kopie erst durchzuführen, wenn keine weiteren Berechnungen mehr erfolgen. Besonders effizient kann man den Datentransfer mit **asynchronen Kopieraufrufen** gestalten: Der gewöhnliche cudaMemcpy Aufruf ist **blocking**, was bedeutet, der gesamte Speichertransfer wird direkt durchgeführt und danach wird erst der nächste Befehl auf der CPU ausgeführt. Dagegen wird beim Aufruf von cudaMemcpyAsync die Kontrolle direkt an die CPU zurückgegeben und der Datentransfer im Hintergrund ausgeführt, was sich also besonders eignet, wenn das Ergebnis fertig im Devicespeicher liegt, man es aber noch nicht direkt braucht, sondern zunächst noch andere unrelatierte Operationen auf der CPU durchführt. Während die CPU weiter rechnet, wird im Hintergrund der Datentransfer durchgeführt und sobald man die Daten tatsächlich benötigt, ist der Transfer teilweise oder ganz fertig. Essentiell kann man sich also die Zeit für einen kompletten Datentransfer sparen.[4]

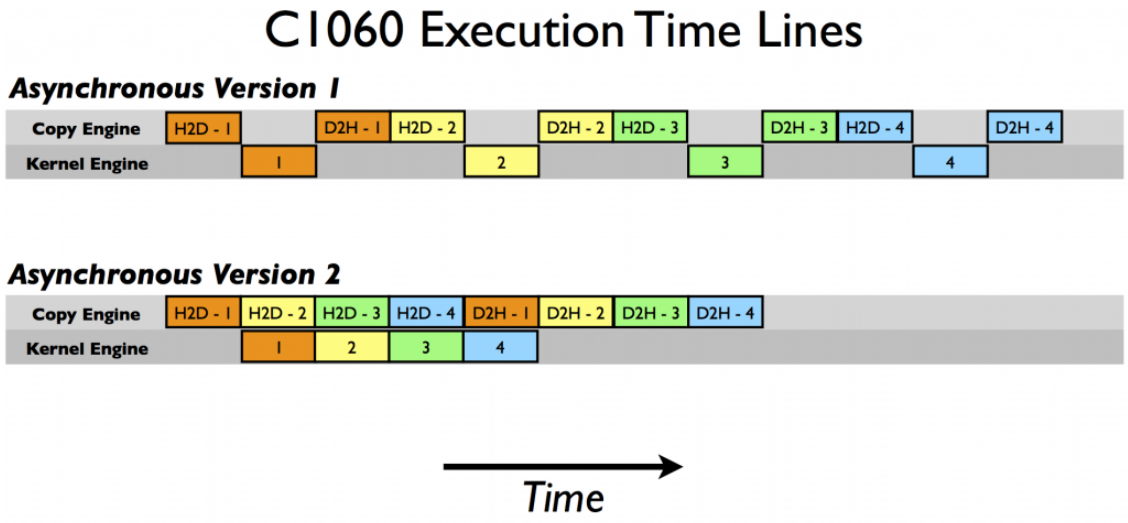

Abbildung 4: Ausführung bei einzelner Copyengine nach [4].

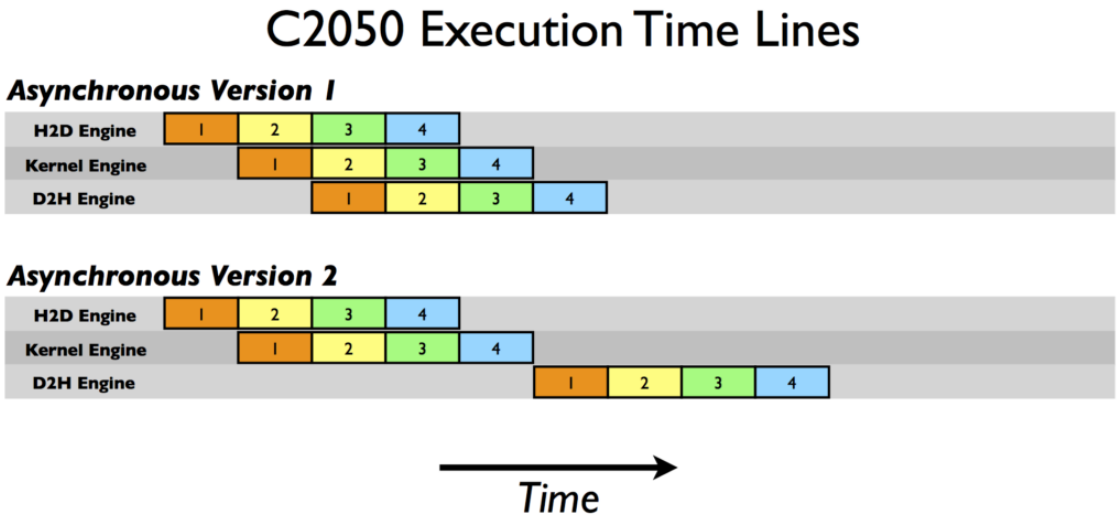

Abbildung 5: Ausführung bei zwei Copyengines nach [4].

In Abbildung 4 und 5 sieht man anschaulich, warum asynchrones Kopieren effizienter sein kann. Es wurden Zeitlinien für unterschiedliche Implementationen der gleichen Berechnung dargestellt. Der relevante Code ist in Listing 5 dargestellt.

```
1  //Asynchronous Version 1
2  for (int i = 0; i < nStreams; ++i) {
3    int offset = i * streamSize;
4    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice,
       stream[i]);
5    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
6    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost,
       stream[i]);
7  }
8
9  //Asynchronous Version 2
10 for (int i = 0; i < nStreams; ++i) {
11   int offset = i * streamSize;
12   cudaMemcpyAsync(&d_a[offset], &a[offset],
13                   streamBytes, cudaMemcpyHostToDevice, cudaMemcpyHostToDevice,
       stream[i]);
14 }
15 for (int i = 0; i < nStreams; ++i) {
16   int offset = i * streamSize;
17   kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
18 }
19 for (int i = 0; i < nStreams; ++i) {
20   int offset = i * streamSize;
21   cudaMemcpyAsync(&a[offset], &d_a[offset],
22                   streamBytes, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToHost,
       stream[i]);
23 }
```

Listing 5: Code der asynchronen Versionen von [4].

Der Datenpuffer d_a wird zur Verarbeitung in kleinere Portionen unterteilt. Hierfür werden Streams benutzt, die davor sorgen, dass die 4 Kernel unabhängig voneinander arbeiten können. Ein Kernel wird nämlich nur ausgeführt, wenn alle zuvor angestoßenen Kopieaufrufe abgearbeitet sind. Durch Streams kann man verschiedene Kernel entkoppeln. Der Kernel im Stream 2 muss nur auf die cudaMemcpyAsync Aufrufe warten, welche im Stream 2 stattfinden. Der große Unterschied zwischen den beiden Versionen ist die Reihenfolge der ausgeführten Befehle. In der Version 1 wird für die Kernel jeweils der H2D(Host to Device) Transfer eingereiht, der Kernel gestartet und der D2H Transfer angestoßen. Dagegen werden in der Version 2 zuerst sämtliche H2D Transfers angestoßen, dann die Kernel aufgerufen und dann die D2H Transfers eingereiht.

Bei der Ausführung zeigen sich deutliche Unterschiede, wobei zuerst auf Abbildung 4 eingegan-

gen wird. Die verwendete Grafikkarte hat nur eine einzige Copy Engine, was bedeutet, es kann entweder nur ein H2D oder ein D2H Transfer durchgeführt werden. Wie erwähnt, muss ein Kernel in einem Stream darauf warten, dass Speichermigrationen abgeschlossen worden sind. Deswegen ergibt sich für die erste Version das exakt gleiche Bild, als würden sämtliche Aufrufe blockierend geschehen. Es wird der H2D - 1 Transfer angeregt, darauf hin wird der Kernel 1 gestartet, welcher aber auf den H2D - 1 Transfer warten muss. Da direkt nach dem Start des Kernels wiederum der D2H - 1 Transfer eingereiht wurde, muss bis zur Ausführung dieser Kopie gewartet werden, bis Kernel 1 fertig ist. Nun kann im selben Schema mit dem zweiten Stream forgefahren werden.
Die zweite Version ist schon deutlich effizienter, denn dort kann nach der ersten Speichermigration(H2D - 1) direkt mit H2D - 2 weitergemacht werden, während der Kernel 1 rechnet. Wenn alle Host to Device Kopien fertig sind, werden die Daten wieder zurück in den Host Speicher kopiert. Mit Beginn von D2H - 1 ist der Kernel 1 schon lange fertig und es kommt somit zu keiner weiteren Verzögerung.

Bei der Ausführung in Abbildung 5 ist tatsächlich die Version 1 schneller, was sich durch die zwei Copy Engines erklären lässt. Hier können gleichzeitig Daten vom Device zum Host und vom Host zum Device kopiert werden. Dadurch wird in der ersten Version auch nicht die Ausführung der Kernels verzögert, da der D2H Datentransfer im Stream 1 nicht den H2D Datentransfer im Stream 2 blockiert. Für die zweite Version würde man das selbe Bild wie bei der ersten Version erwarten, bei der verwendeten Grafikkarte wird aber das Signal, dass die Erledigung eines Kernels anzeigt, verzögert, wenn mehrere Kernel direkt hintereinander gestartet werden. Erst wenn alle Kernel fertig sind, kann der D2H Transfer ausgeführt werden.[4]

3.4 Unified Memory

Von NVIDIA wurde das Konzept **Unified Memory** entwickelt, bei welchem dem Programmierer vorgespielt wird, er hätte bestimmte Speicheradressen, die für die CPU und für die GPU gültig sind. Tatsächlich wird der Speicher beim Anlegen zunächst nur auf der GPU reserviert und für die CPU wird lediglich die **Seitentabelle** so angepasst, dass beim Zugriff von der CPU aus ein **Seitenfehler** auftritt. Der Grafikkartentreiber fängt diesen ab und migriert die Daten vom Devicespeicher zum Hostspeicher. Greift ein Kernel auf Unified Memory zu, gibt es seit der Pascal Architektur(ca. 2016) zwei Lösungsstrategien: Vor Pascal(bspw. Kepler) wurden mit dem Aufruf eines Kernels **alle** Seiten, die zuvor von Devicespeicher zu Hostspeicher migriert worden sind, zurück auf den Grafikkartenspeicher migriert. Seit der Pascal Architektur unterstützen Nvidia Grafikkarten Seitentabellen, wodurch **On-Demand Page Migration** möglich wurde. Nun müssen vor dem Starten eines neuen Kernels nicht mehr sämtliche auf der CPU befindlichen Seiten direkt auf die GPU kopiert werden, sondern werden während der Kernelausführung kopiert, sobald auf die jeweilige Seite zugegriffen wird. Ein weiterer Vorteil seit Pascal ist die Möglichkeit aufgrund der Seitentabelle, mehr Speicher auf der GPU reservieren zu können, als tatsächlich physisch vorhanden ist. Immerhin müssen nicht mehr sämtliche Daten im Devicespeicher liegen, sondern können an anderen Orten abgelegt sein. Bei Benutzung von aktuell nicht vorhandenen Speicher wird eine andere Seite ausgelagert und die angeforderte Seite im Devicespeicher eingelagert.

Unified Memory vereinfacht die Programmierung erheblich und kann ähnlich gute Performance wie manuelle Speicherverwaltung erreichen. Unter anderem wurde diese Technik auch im Rahmen dieser Arbeit verwendet. [5]

3.5 Occupancy

Bei der Grafikkartenprogrammierung werden Kernel geschrieben, die dann letztendlich von sehr vielen Threads gleichzeitig auf der Grafikkarte ausgeführt werden. Beim Start eines Kernels teilt man der Grafikkarte allerdings nicht mit, wie viele Threads man insgesamt ausführen möchte, sondern definiert eine Anzahl an **Blöcken** und **Threads pro Block**. Zunächst einmal ist zu verdeutlichen, dass nicht sämtliche Blöcke gleichzeitig verarbeitet werden. In UGBlocks werden für große Gitter teilweise mehr als 3000 Blöcke mit je 128 Threads gestartet. Es ist klar, dass die Grafikkarte nicht jeden dieser Threads gleichzeitig ausführen kann, also werden die Blöcke nacheinander abgearbeitet. Einem **Streaming Multiprocessor** werden mehrere Blöcke zur Ausführung zugewiesen, wobei die genaue Anzahl der Blöcke von mehreren Hardwarelimits beschränkt werden. Tabelle 1 beinhaltet einige dieser Limits für eine **GTX 1070 Ti**.

Tabelle 1: Limits einer GTX 1070 Ti.

Beschreibung	**Wert**
Maximum an Warps pro SM	64
Maximum an Blöcke pro SM	32
Maximum an Threads pro SM	2048
Maximum an Threads pro Block	1024
Maximum an verwendeten Registern pro SM	65536
Maximum an verwendeten Registern pro Block	65536
Maximum an verwendeten Registern pro Thread	255
Maximum an Shared Memory pro SM	65536 Bytes
Maximum an Shared Memory pro Block	49152 Bytes

Im Grundlagenkapitel wurde erwähnt, dass ein SM schnell zwischen der Ausführung verschiedener Warps wechseln kann, und so die relativ große Latenz des Arbeitsspeichers verstecken kann, indem einfach ein anderer Warp ausgeführt wird, wenn der gerade in Abarbeitung befindliche Warp auf Daten warten muss. Damit das möglichst gut funktioniert, sollte die Blockgröße so gewählt werden, dass das Limit von Warps pro SM erreicht wird. In diesem Falle hätte man eine Occupancy von 100%.

$$Occupancy = \frac{WarpsProSM}{maximaleWarpsProSM} \tag{3}$$

Um die Anzahl an Warps zu erhalten, muss man den kleinsten von drei berechneten Werten nehmen. Zunächst benötigt man immer die Anzahl an Warps pro Block, welche man errechnet, indem man die Blockgröße durch 32 teilt und aufrundet. Wir nehmen exemplarisch einmal 67 Threads pro Block an.

Warps pro Block = $\lceil \frac{67}{32} \rceil = \lceil 2,093 \rceil = 3$

1. Unter Beachtung der maximalen Warp- und Blockzahl pro SM kann man errechnen, wie viele Blocks pro SM aktiv sein können. Da maximal 64 aktive Warps pro SM möglich sind, können wir $\lfloor \frac{64}{3} \rfloor = 21$ Blöcke erstellen, was also 63 aktive Warps auf einem Streaming Multiprocessor bedeuten würde.

2. Je nach Kernel kann es sich unterscheiden, wie viele Register ein Thread verwendet, insgesamt müssen wir hier wiederum die Limits an Register pro Block und SM einhalten. Wir stellen uns vor, unser Kernel bräuchte 127 Register pro Thread. Demnach würde ein Warp 32 * 127 = 4064 Register und ein Block 4064 * 3 = 12192 Register benötigen. In diesem Fall können wir also bloß $\lfloor \frac{65536}{12192} \rfloor = 5$ Blöcke pro SM aktiv halten, was dann 15 aktiven Warps entspricht.

3. Die letzt Zahl errechnet sich ähnlich zur vorherigen: Nicht nur die Registeranzahl ist beschränkt, sondern auch die Verwendung des Shared Memory pro SM. Wenn wir pro Block 22000 Bytes Shared Memory benötigen, können wir je SM nur $\lfloor \frac{65536}{22000} \rfloor = 2$ Blöcke aktiv halten, was 6 Warps pro SM entspräche.

Von diesen drei Zahlen ist die kleinste, also **6**, die Anzahl an aktiven Warps pro SM. Im konkreten Fall hätten wir also eine **Occupancy von 9%**. In unserem Beispiel wird man die Occupancy nicht viel weiter steigern können, da sie durch den Kernel an sich limitiert ist. Man könnte den Kernel umschreiben und so die verwendete Registerzahl bzw. die größe des Shared Memory verringern. Trotzdem bedeutet ein größerer Occupancy Wert nicht zwangsweise eine höhere Performance, wie man auch im Kapitel 6 anhand der Messungen sehen kann. Dort fällt die Performance, beim Übergang von 512 zu 514 Threads pro Block, was eine um 20% niedrigere Occupancy bewirkt hat, stark ab. Allerdings wird bei der Blockgröße von 672 wieder eine Occupancy von 98% erreicht, die Rechengeschwindigkeit verbessert sich aber nicht merklich. Insofern sollte man im Einzelfall

abwägen, ob man unbedingt höhere Occupancy Werte erzielen möchte. Bei weiterem Interesse an Occupancy Berechnungen, lohnt sich ein Blick in den CUDA Occupancy Calculator[12], von dem auch die Limits für Tabelle 1 entnommen wurden.

3.6 Cooperative Groups

Ursprünglich war es bei der CUDA Programmierung nur möglich, mit der Funktion **__syncthreads** einen gesamten Threadblock zu synchronisieren. Das bedeutet, der Code aller Threads eines Blockes musste bis zur **syncthreads** Anweisung ausgeführt werden und erst danach konnten Threads ihre Codeausführung fortsetzen. Dieser Mechanismus ist zum Beispiel wichtig für eine Reduktion und wird auch bei UGBlocks verwendet, um die Summation bei einem Skalarprodukt durchzuführen. Seit CUDA 9 gibt es sogenannte **Cooperative Groups**, mit welchem dem Programmierer die Möglichkeit gegeben wird, seine Threads feiner oder grober zu Gruppieren und untereinander zu synchronisieren.[6] Beispielsweise kann man die Threads eines Blockes in Gruppen von 32 aufteilen, was besonders häufig verwendet wird, da man somit spezielle Funktionen verwenden kann, welche die Notwendigkeit von **Shared Memory** eliminieren. Ein Beispiel findet sich in Listing 6.

```
1 __device__ int reduce_sum_tile_shfl(thread_block_tile<32> g, int val)
2 {
3     // Each iteration halves the number of active threads
4     // Each thread adds its partial sum[i] to sum[lane+i]
5     for (int i = g.size() / 2; i > 0; i /= 2) {
6         val += g.shfl_down(val, i);
7     }
8     return val; // note: only thread 0 will return full sum
9 }
```

Listing 6: shfl_down Beispielcode von [6].

Ohne die verwendete **shfl_down** Funktion würde man sämtliche Threads ihren Wert **val** in Shared Memory schreiben lassen, eine Synchronisierung durchführen und danach die Werte wieder aus dem Shared Memory lesen. Die **shfl** Funktionen ermöglichen es, direkt den Wert einer lokalen Variable aus einem anderen Thread zu lesen, der im selben Warp existiert. In der for Schleife von Listing 6 nimmt **i** die folgenden Werte an: 16, 8, 4, 2, 1. Im ersten Schleifendurchlauf wird also g.**shfl_down**(val,16) ausgeführt, was den Wert von val aus dem Thread zurückgibt, dessen Id im Warp um 16 größer ist. Dadurch halten nach einem Durchlauf die ersten 16 Threads im Warp die relevanten Teilsummen. In den nächsten Durchläufen sind es nur noch die ersten 8, 4,... Am Ende der Schleife hat der erste Thread im Warp die komplette Summe. Die Reduktion für einen Warp ist extrem effizient, allerdings müssen die Werte pro Warp letztendlich auch wieder summiert werden. Dafür gibt es viele verschiedene Techniken, welche nicht weiter erläutert werden. Nvidia veröffentlicht sehr viele Informationen zu diesem Thema. Um noch auf den Aspekt zurückzukommen, seine Threads grober als in Blöcken zu synchronisieren, sei noch kurz erwähnt, dass man mit Hilfe von Cooperative Groups auch mehrere Blöcke oder sogar verschiedene GPUs untereinander synchronisieren kann.

4 GPU Auswertung der Expressions in UGBlocks

4.1 Allgemeines

UGBlocks ist eine vom Lehrstuhl für Informatik 10 (Systemsimulation) entwickelte Bibliothek zum Lösen von Partiellen Differentialgleichungen mit Hilfe von unstrukturierten Blockgittern. Durch die Verwendung von Expression Templates wird eine hohe Benutzerfreundlichkeit und gute Rechenleistung erzielt.

Um die Geschwindigkeit der Rechnungen noch weiter zu verbessern, wurden im Rahmen dieser Arbeit Teile des Codes so verändert, dass Berechnungen auf der Grafikkarte durchgeführt werden können. Dabei war es wichtig, den Code so zu implementieren, dass auf den Hauptprozessor zurückgegriffen wird, wenn keine kompatible Grafikhardware erkannt wurde.

UGBlocks benutzt QT Funktionalität und wurde mit der Hilfe von QMake kompiliert. Bei der Kompilation von CUDA spezifischem Code muss aber der **nvcc** Compiler verwendet werden, was einige Schwierigkeiten bei der Konfiguration der QMake Projekt Datei mit sich brachte. Man kann

in QMake einen sogenannten Extra Compiler definieren und so komplett selbst die Kommandozeile definieren, mit welcher die Quelldateien in Objektdateien übersetzt werden. Normalerweise würde man QMake also so konfigurieren, dass sämtliche Quelldateien mit CUDA Code vom nvcc Compiler und die restlichen Dateien vom Standardcompiler verarbeitet werden. Problematisch ist hierfür aber der Aufbau der UGBlocks Bibliothek und die Verwendung von Expression Templates. Sehr viel Code wird nämlich in den Header Dateien definiert, was letztendlich doppelte Code Kompilation zur Folge hat. Die main.cc Datei wird vom Standardcompiler bearbeitet. Dieser sieht die inkludierten Headerdateien und verarbeitet auch diese. Da dort Funktionen aufgrund der Verwendung von Templates direkt definiert ist, und nicht wie üblich in der .cc Datei, wird dieser Code direkt vom Compiler übersetzt. Der nvcc Compiler wird in gewissen Fällen die selben Headerdateien wie der Standardcompiler betrachten und übersetzt diesen Code dann ein weiteres mal. Beim Linken der Objektdateien wird nun der Fehler auftreten, dass die selben Funktionen in unterschiedlichen Objektdateien vorhanden sind. Zusätzlich gibt es ein weiteres Problem durch den Kompilationsprozess in Bezug auf C++ Templates: Der Code für Templates wird erst in Maschinencode übersetzt, wenn der Compiler auf eine konkrete Verwendung dieses Codes stößt.

```
1  //templateClass.h
2  template<typename A>
3  class Test{
4  public:
5    A givePayload();
6    void setPayload(A p);
7  private:
8    A payload;
9  };
10
11 //templateClass.cpp
12 #include "templateClass.h"
13 A Test::givePayload(){
14   return payload;
15 }
16 void Test:.setPayload(A p)
17 {
18   payload = p;
19 }
20
21 //main.cpp
22 #include templateClass.h
23
24 int main(int argc, char** argv)
25 {
26   Test<double> usage;
27   usage.setPayload(2.333);
28
29   return 0;
30 }
```

Listing 7: Kompilationsprobleme mit Templates.

Der in Listing 7. dargestellte Code würde nicht kompilieren, da der Compiler versucht, die Test Klasse für A=double zu übersetzen, sobald er auf die Benutzung in der main-Methode stößt. Der Methodenrumpf von **getPayload** und **setPayload** ist aber in der **templateClass.cpp** Datei definiert, von deren Existenz der Compiler nichts weiß, weswegen er an dieser Stelle einen Fehler ausgeben wird. Nehmen wir an, der nvcc Compiler versucht die templateClass.cpp zu übersetzen, ruft dabei die templateClass.h Datei auf und erkennt, dass es sich bei der Klasse um eine Templateklasse handelt. Da er in seinem gesamten Kompilationsprozess auf keine Verwendung der Test Klasse getroffen ist, wird er letztendlich keinen Code für templateClass.cpp generieren.

Die gefundene Lösung für diese Probleme war es, in der QMake Projekt Datei ausschließlich den nvcc Compiler zu verwenden, sodass dieser sämtliche Quelldateien sichtet und die entsprechenden Templates generieren kann.

4.2 Expression Templates auf der GPU durch dynamische Kernelgenerierung

Die Evaluation von Expression Templates mit CUDA wurde 2011 wurde von Wiemann u.a. demonstriert.[14] Dort wurden die Kernel, welche die Berechnung durchführten, zur Laufzeit aus den Expressions generiert. In ihrem Paper erklären die Autoren, dass der CUDA Compiler(nvcc) keine generelle Templateprogrammierung unterstützt, weshalb die Expression Templates nicht direkt im CUDA Code verwendet werden können, sondern für jede Expression zur Laufzeit ein CUDA Kernel erzeugt wird. Hierfür wurde ein Kernel Prototyp definiert, dessen Parameterliste und Evaluationsausdruck abhängig vom konkreten Ausdrucksbaum verändert wird. Da die Kernelgeneration zur Laufzeit geschieht, verlängert sich die Gesamtausführungszeit, was bei großen Berechnungen aber nicht mehr so stark ins Gewicht fällt, weil der Performancegewinn enorm ist. Insofern kann man sagen, dass dies ein erfolgreicher Ansatz war, Expression Templates auf der Grafikkarte möglich zu machen.

4.3 Implementation nach Knauer

Da neuere CUDA Versionen vollständige C++ Templates ermöglichen, kann man Expression Templates nun auch mit einer anderen Strategie auf Grafikkarten lauffähig machen. Diese Strategie wurde von Knauer im Rahmen seiner Bachelorarbeit[8] erörtert. Ausgehend von seinen Ergebnissen wurde die Bibliothek UGBlocks erweitert.

CUDA bietet inzwischen die Möglichkeit, einzelne Methoden einer Klasse mit __device__ kennzuzeichnen, wodurch diese Methode für die Grafikkarte übersetzt wird. Die Auswertung einer Expression geschieht bei Knauer in einem Lambda Ausdruck, der als Kernel gestartet wird. Dort wird lediglich die **get** Funktion der Wurzel des Ausdrucks aufgerufen.

```
1  template <class DTyp>
2  template <class A>
3  void Cell_variable<DTyp>::operator=(const Expr<A>& a) {
4    ...
5    A ao(a);
6    ao.SyncData(false);
7
8    auto data = dataContainer.GetDataTotalGPUPtr();
9    auto lambda = [totalNumberData, data, ao]__device__() -> void {
10     int i = blockIdx.x * blockDim.x + threadIdx.x;
11     if (i >= totalNumberData)
12       return;
13
14       data[i] = ao.Give_fromTotal(i);
15     };
16
17   int threads = CudaOptimizer::GetInstance().GetNumberOfThreads();
18   int blocks = (totalNumberData + (threads - 1)) / threads;
19   lambda_as_kernel<decltype(lambda)><<<blocks, threads>>>(lambda);
20   dataContainer.SetDataFlags(false, true);
21
22   ...
23 }
```

Listing 8: Auswertung der Expression mit CUDA.

In Listing 8. sieht man den für UGBlocks geschriebenen Code, welcher sich nur leicht von Knauers Code unterscheidet. Zunächst wird eine lokale Kopie der übergebenen **Expression a** erstellt(Zeile 6) und deren **SyncData** Funktion aufgerufen. Mit dieser Funktion wird die gesamte Struktur des Ausdrucksbaumes traversiert und dafür gesorgt, dass die Daten sämtlicher Terminale(Cell_variablen) in den Devicespeicher geladen werden, falls sie dort noch nicht vorhanden sind. Das Vorhandensein wird anhand von Wahrheitswerten geprüft, die bei einer Veränderung der Daten stets aktualisiert werden müssen. Ein Beispiel hierfür findet sich in Zeile 21. Hier werden Wahrheitswerte gesetzt, anhand derer man ableiten kann, wo sich die aktuellen Daten gerade befinden. In diesem Fall befinden sie sich nach der Ausführung des Kernels im Grafikkartenspeicher. Wollte man direkt nach der Kernelausführung die Werte der Cell_variable mit Hilfe von Hostcode ausgeben, müssten diese wiederum vom Devicespeicher in den Hostspeicher migriert werden. Im Kernelcode(Zeile 10 bis 15) wird lediglich geprüft, dass kein Zugriff außerhalb des gültigen Datenbereichs geschieht(Zeile

11) und der Ausdrucksbaum wird mit der Funktion **Give_fromTotal** traversiert. Da Expressions immer noch auf der CPU auswertbar sein sollen, müss diese Funktion sowohl für die Grafikkarte, als auch für den Hauptprozessor übersetzt werden, was durch die Annotation mit __device__ und __host__ möglich ist. Für die Operationsklassen ist der Host- und Devicecode gleich,aber für die Terminale(Zellvariablen) des Ausdruckbaumes muss im Devicecode auf den Zeiger zugegriffen werden, welcher auf der Grafikkarte gültig ist, und entsprechend muss im Hostcode auf den Zeiger zugegriffen werden, der für den Hauptprozessor gültig ist. Listing 9. zeigt, wie das erreicht werden kann.

```
1  template <class DTyp>
2  class Cell_variable : public Expr<Cell_variable<DTyp>> {
3    ...
4    __host__ __device__ inline DTyp Give_fromTotal(int i) const {
5  #ifdef __CUDA_ARCH__
6      //Zugriff ueber den Zeiger fuer Devicespeicher
7      return dataContainer.GetDataTotalGPUPtr()[i];
8  #else
9      //Zugriff ueber den Zeiger fuer Hostspeicher
10     return dataContainer.GetDataTotalPtr()[i];
11 #endif
12   }
13   ...
14 };
15
16 template <class A, class B>
17 class Add : public Expr<Add<A, B> > {
18   A a_;
19   B b_;
20
21 public:
22   bool CalculateOnCPU() const { return a_.CalculateOnCPU() || b_.CalculateOnCPU(); }
23   void SyncData(bool toCPU) { a_.SyncData(toCPU); b_.SyncData(toCPU); }
24   __host__ __device__ inline Result Give_fromTotal(int i) const {
25     //Gleicher Code fuer GPU und CPU
26     return a_.Give_fromTotal(i) +
27       b_.Give_fromTotal(i);
28   }
29   ...
30 };
```

Listing 9: Implementation von GivefromTotal.

Mit Hilfe der Präpozessoranweisung kann die Methode Give_fromTotal unterschiedlichen Quelltext für die GPU und CPU Version beinhalten. Bei der Definition von **__CUDA_ARCH__** wird der Devicecode übersetzt, ansonsten der Hostcode. In der **Add Expression** sind dagegen keine unterschiedlichen Methodenrümpfe erforderlich. Wird Give_fromTotal im Devicecode aufgerufen, wird auch die Devicecodevariante für Give_fromTotal von a_ und b_ aufgerufen. Falls diese Operanden Cell_variablen sind, wird der Wert korrekterweise aus dem Devicespeicher ausgelesen.

Ein wichtiger Unterschied zu gewöhnlichen Expression Templates ist bei den Operanden der Add Expression zu sehen. Diese **können nicht mehr als Referenzen gespeichert werden**. Referenzen werden prinzipiell vom Compiler wie Zeiger übersetzt, was bedeutet, diese Zeiger verlieren ihre Gültigkeit im Devicecode, da sie auf Speicheradressen des Hosts zeigen. Da keine Referenzen verwendet werden können, müssen die verwendeten Objekte(Operationsklassen und Cellvariablen) häufig kopiert werden. Es wäre unpraktisch, stets den kompletten verwalteten Speicher einer Zellvariable zu kopieren, stattdessen werden nur die Zeiger auf die Datenpuffer kopiert. Um sogenannte Memoryleaks - nicht mehr benutzter Speicher, der nicht freigegeben wurde - zu verhindern, wird bei jeder Kopie ein Referenzzähler um eins erhöht. Bei jeder Destruktion einer Zellvariable wird dieser Referenzzähler wieder um eins verringert. Sobald der Referenzzähler die Zahl null erreicht, ist die letzte Kopie einer Zellvariable zerstört und der zuvor angeforderte Speicher nun freigegeben.

4.4 Rückgriff auf CPU

Wie bereits erwähnt, sollen Berechnungen auf der CPU durchgeführt werden, wenn keine GPU verfügbar ist. Um das zu erreichen wird in sämtlichen Klassen, die in einer Expression auftauchen können, ein Wahrheitswert eingefügt, ob die Berechnung auf der CPU durchgeführt werden soll. Im

Zuweisungsoperator, also der Operator, der die Evaluation einer Expression durchführt(siehe Listing 8.), wird von einer Expression abgefragt, ob diese auf der CPU berechnet werden soll. Dabei wird der gesamte Baum traversiert und die Rückgabewerte werden mit einem logischem Oder verknüpft, wodurch der Gesamtrückgabewert wahr wird, wenn ein einziges Element im Baum auf der CPU berechnet werden muss. Diese Abfrage wird in der Funktion CalculateOnCPU realisiert und ist in Listing 9. mit dargestellt.

4.5 Speichersynchronisation

4.5.1 Manuelle Synchronisation

Bei der manuellen Synchronisation musste in der Bibliothek an vielen Stellen zusätzlicher Code eingefügt werden, der dafür sorgte, dass die Daten dorthin migriert werden, wo sie gebraucht werden. Die Zellvariable besitzt verschiedene Funktionen, die ihre Daten als VTK-Datei ausgeben können. Dort musste bedarfsweise eine Synchronisation vom Device- zum Hostspeicher angestoßen werden. Bei der Auswertung einer Expression, sei es auf der GPU oder CPU, müssen für sämtliche Terminale im Ausdruck ebenfalls Synchronisationen durchgeführt werden. Das geschieht über die Funktion **SyncData**, welche von jeder Klasse, die in einer Expression vorkommen kann, implementiert werden muss. In UGBlocks gibt es aktuell noch Klassen, die nicht auf der GPU berechnet werden können, aber trotzdem in Expressions vorkommen. Auch diese müssen die Funktion SyncData definieren, können den Methodenrumpf aber leer lassen, da in diesem Fall keine Daten migriert werden müssen. Die Implementation von SyncData für eine Operationsklasse findet sich in Listing 9. Für Zellvariablen sieht diese Funktion schon komplizierter aus, da hier tatsächlich Daten migriert werden müssen. Abhängig von den Wahrheitswerten, die angeben, wo sich die Daten befinden und vom Parameter der SyncData Funktion wird letztlich nur cudaMemcpy aufgerufen und die Wahrheitswerte werden aktualisiert. Als Parameter der SyncData Funktion wird mitgegeben, ob die Daten von der CPU oder von der GPU benötigt werden, der Grund dafür ist klar: Die Daten werden nur migriert, wenn sie dort gebraucht werden, wo sie nicht aktuell sind. In anderen Fällen wird nichts kopiert, da ständiger Speichertransfer das Programm stark ausbremsen würde.

4.5.2 Unified Memory

Durch die Verwendung von Unified Memory fällt die Notwendigkeit weg, sich um die Speichersynchronisation und das Setzen der Flags zu kümmern. Außerdem benötigt man in der Klasse Cell_variable keine unterschiedlichen Versionen von Give_fromTotal mehr implementieren. Allgemein wird der Code um einiges lesbarer und auch die relativ aufwändige Struktur zur Datenspeicherung entfällt. Um den CPU Fallback zu realisieren, werden die Zellvariablen ausschließlich mit Builderfunktionen erzeugt, was durch das Privatsetzen ihres Konstruktors sichergestellt wird. Ein Builder erzeugt eine Zellvariable für CPU Berechnungen, der andere eine Zellvariable, die sowohl auf der GPU als auch CPU rechnen kann. Letztere erzeugt den Speicher mit Unified Memory, sodass die Speicheradressen, wie wir aus Kapitel 3 wissen, sowohl im Devicecode als auch Hostcode verwendet werden können. Ursprünglich wurde versucht, die gesamte Cell_variable mit Unified Memory zu erzeugen, wobei sich ein entscheidender Nachteil aufzeigt: Da der Wahrheitswert cudaCalculation, der anzeigt, ob die Berechnung auf der GPU oder CPU durchgeführt werden soll, somit auch mit Unified Memory erzeugt wurde, geschieht bei jedem Zugriff auf diesen eine Seitenmigration von Devicespeicher zu Hostspeicher. Ein Teil der Daten, welche in den Expressions verarbeitet werden, liegen in der selben Seite, wie cudaCalculation, weswegen bei der Auswertung der Expression genau diese Seite wieder in den Devicespeicher migriert wird. Nach der Auswertung wird wieder auf cudaCalculation zugegriffen und die gesamte Seite unnötig in den Hostspeicher kopiert. Dieser Mechanismus hat die Performance des Programms extrem ausgebremst.

5 Optimierung

Der Auswertungsmechanismus der Expressions lässt sich nicht mehr optimieren, da dieser, wie Knauer[8] gezeigt hat, für CUDA die selbe Rechenleistung wie handgeschriebene Kernel erreicht. Optimierungsmöglichkeiten bestehen bei der Wahl möglichst guter Block- und Griddimensionen

und der effizienten Implementation des Skalarprodukts, was bei UGBlocks erfolgreich umgesetzt wurde. Die genauen Informationen finden sich in den nachfolgenden Kapiteln.

5.1 Effiziente Kernelaufrufparameter

Wie bereits ausführlich beschrieben worden ist, sollte man bei der Wahl der Blockgröße auf jedenfall ein vielfaches von 32 verwenden, und eine Occupancy von 100% anstreben. Letzteres war sehr einfach realisierbar, da die verwendeten Kernel nicht besonders viele Register und kein Shared Memory, verwenden. Weil sich in Messungen gezeigt hat, dass große Blockgrößen weniger GFLOPS/s bringen, als kleine Blockgrößen, wurde die Threadzahl pro Block auf ein achtel der maximalen Threads pro Block gesetzt. Dies wird in UGBlocks über die Singleton Klasse **CudaOptimizer** realisiert, welche eine Funktion definiert, die einen Vorschlag für die zu verwendende Blockgröße liefert.

```
1 int GetNumberOfThreads() {
2     return (int)((double)DeviceProperties.warpSize * floor((double)DeviceProperties.
      maxThreadsPerBlock / 8.0 / (double)DeviceProperties.warpSize));
3 }
```

Listing 10: GetNumberOfThreads Funktion.

Das Objekt **DeviceProperties** beinhaltet verschiedene Eigenschaften der eingebauten Grafikkarte, welche zuvor mit **cudaGetDeviceProperties** abgefragt wurden. Im Code wurde darauf geachtet, dass die Blockgröße ein vielfaches der Warp Größe ist. Diese wurde nicht mit 32 festgelegt, sondern wird auch durch den CUDA API Aufruf abgefragt, was den Vorteil hat, dass diese Funktion nicht verändert werden muss, sollte Nvidia irgendwann die Warpgröße verändern.

5.2 Effizientes Skalarprodukt

Das Skalarprodukt lässt sich auf verschiedene Wege implementieren. Ein einfacher Ansatz, der auch zunächst für UGBlocks verwendet wurde, wäre beispielsweise, jeden Thread genau vier Gitterpunkte miteinander multiplizieren und aufaddieren zu lassen. Die Ergebnisse werden in einen Buffer zurückgeschrieben, der nach der kompletten Ausführung auf die CPU kopiert wird. Die darin verbliebenen Werte werden nun von der CPU aufsummiert. Durch diese Methode müssen für das Skalarprodukt keine Multiplikationen mehr durchgeführt werden, aber immer noch sehr viele Additionen. Dementsprechend lässt sich zwar bereits eine Leistungssteigerung gegenüber der reinen CPU Berechnung feststellen, es wäre aber wünschenswert die komplette Berechnung auf der Grafikkarte auszuführen und nur das Endergebnis zurückzugeben.

Um die Funktionsweiße des Skalarproduktes zu erklären wird zunächst ein unpraktikabler Ansatz erklärt. Man lässt jeden Thread nach wie vor vier Gitterpunkte verarbeiten, addiert aber das Ergebnis zum endgültigen double Wert dazu. Da die Threads parallel ausgeführt werden, würden Nebenläufigkeitsprobleme auftreten, verhindern lässt sich das mit der Funktion **atomicAdd**. Mit dieser kann ein Thread in einem atomaren Schritt den double Wert lesen, seinen Wert dazu addieren und den neuen Wert zurückschreiben, wodurch alle Threads parallel ihren Code ausführen können. Tatsächlich ist die atomicAdd Funktion sehr aufwändig und sollte so wenig wie möglich verwendet werden, weshalb man gewöhnlich einen anderen Ansatz verwendet, welcher auch von Knauer verwendet wurde. Jeder Block summiert für sich eine Teilsumme, die vom ersten Thread im Block mit Hilfe von atomicAdd zum Endresultat dazu addiert wird. Die Reduktion im Block geschieht in $\log_2 b$ Schritten. Zunächst schreibt jeder Thread seine eigene Zwischensumme in einen Shared Memory Block. Nun ist nur noch die Hälfte aller Threads aktiv. Jeder aktive Thread addiert die Teilsumme eines inaktiven Threads zu seiner eigenen Summe und schreibt das Ergebnis wieder in Shared Memory hinein. Nun halbiert sich immer wieder die Menge der aktiven Threads, bis nur noch ein Thread im Block aktiv ist. Dieser verfügt nun über die Gesamtsumme des Blocks und führt die atomicAdd Instruktion aus. Der genaue Algorithmus ist in Listing 11 zu sehen. Die Methode __syncthreads stellt hier einen Synchronisierungspunkt dar, den alle Threads in einem Block erreichen müssen, bevor weiter gerechnet wird. In diesem Fall wird so sichergestellt, dass alle Threads ihren Wert in den Shared Memory Block geschrieben haben, bevor im nächsten Schleifendurchlauf darauf zugegriffen wird.

```
1 __global__ void EfficientDotproduct(double* Result, double* InputA, double* InputB,
      int n)
2 {
```

```
3   extern __shared__ double data[];
4   data[threadIdx.x] = 0.0; //Initially set the value to 0
5   for (int index = blockIdx.x * blockDim.x + threadIdx.x; index < n; index += (
      gridDim.x * blockDim.x))
6   {
7     data[threadIdx.x] += InputA[index] * InputB[index];
8   }
9   __syncthreads();
10  for (int activeThreads = blockDim.x / 2; activeThreads > 0; activeThreads /= 2)
11  {
12    if (threadIdx.x < activeThreads)
13    {
14      data[threadIdx.x] += data[threadIdx.x + activeThreads];
15    }
16    __syncthreads();
17  }
18
19  if (threadIdx.x == 0)
20  {
21    atomicAdd(Result, data[0]);
22  }
23 }
```

Listing 11: Blockweise Reduktion.

Für UGBlocks wurde noch eine andere Methode getestet, bei der nicht für einen ganzen Block die Teilsumme gebildet wird, sondern für jeden einzelnen Warp im Grid. Realisiert wird das mit den Cooperative Groups und einer **shfl** Funktion, welche in Kapitel 3 erwähnt wurden. Prinzipiell fällt Synchronisierungsaufwand weg, es kommen aber mehr atomicAdd Instruktionen hinzu. Trotzdem war die Warpweise Reduktion in teilen des Performancetests sogar besser als die Blockweise Reduktion.

```
1  __global__ void EfficientDotproduct(double* Result, double* InputA, double* InputB,
     int n)
2  {
3    auto block = cg::this_thread_block();
4    auto warp = cg::tiled_partition<32>(block);
5    auto warpLane = warp.thread_rank();
6
7    double myValue = 0.0;
8    for (int index = blockIdx.x * blockDim.x + threadIdx.x; index < n; index += (
       gridDim.x * blockDim.x))
9    {
10     myValue += InputA[index] * InputB[index];
11   }
12
13   myValue += __shfl_down_sync(0xFFFFFFFF, myValue, 1);
14   myValue += __shfl_down_sync(0xFFFFFFFF, myValue, 2);
15   myValue += __shfl_down_sync(0xFFFFFFFF, myValue, 4);
16   myValue += __shfl_down_sync(0xFFFFFFFF, myValue, 8);
17   myValue += __shfl_down_sync(0xFFFFFFFF, myValue, 16);
18   if (warpLane == 0)
19   {
20     atomicAdd(Result, myValue);
21   }
22 }
```

Listing 12: Warpweise Reduktion.

Die Funktion shfl_down_sync holt sich den Wert eines anderen Threads im selben Warp, dabei gibt der letzte Parameter die relative Position des Zielthreads zum ausführenden Thread an. Die erste shfl Instruktion holt also für jeden Thread den myValue Wert des Threads mit der nächst höheren Id. Da Anweisungen für jeden Thread im Warp synchron ausgeführt werden, fällt die aufwändige Synchronisierung durch **syncthreads** weg. Am Ende hat wieder der erste Thread im Warp die Gesamtsumme des Warps und addiert diese mit atomicAdd zum Endergebnis dazu.

6 Performance

In diesen Kapitel werden sämtliche durchgeführte Performancemessungen vorgestellt. Diese Messungen wurden einmal mit Hilfe einer Kombination aus **Intel i5-8600k und GeForce GTX 1070 Ti** und einmal mit einem **Intel i7-9750H und einer Quadro T1000** berechnet. Zuerst wird eine ausführliche Messung mit verschiedenen Blockgrößen, bei gleichbleibender Problemgröße(Gitter und Iterationszahl bleiben gleich) vorgestellt. Danach wurde die Leistung für eine Version mit ineffizienter Skalarproduktberechnung gemessen und schlussendlich finden sich Diagramme zur Leistung der finalen Version von UGBlocks.

6.1 Variierende Blockgrößen

Die Berechnungen wurden ohne Unified Memory mit manueller Synchronisation durchgeführt.

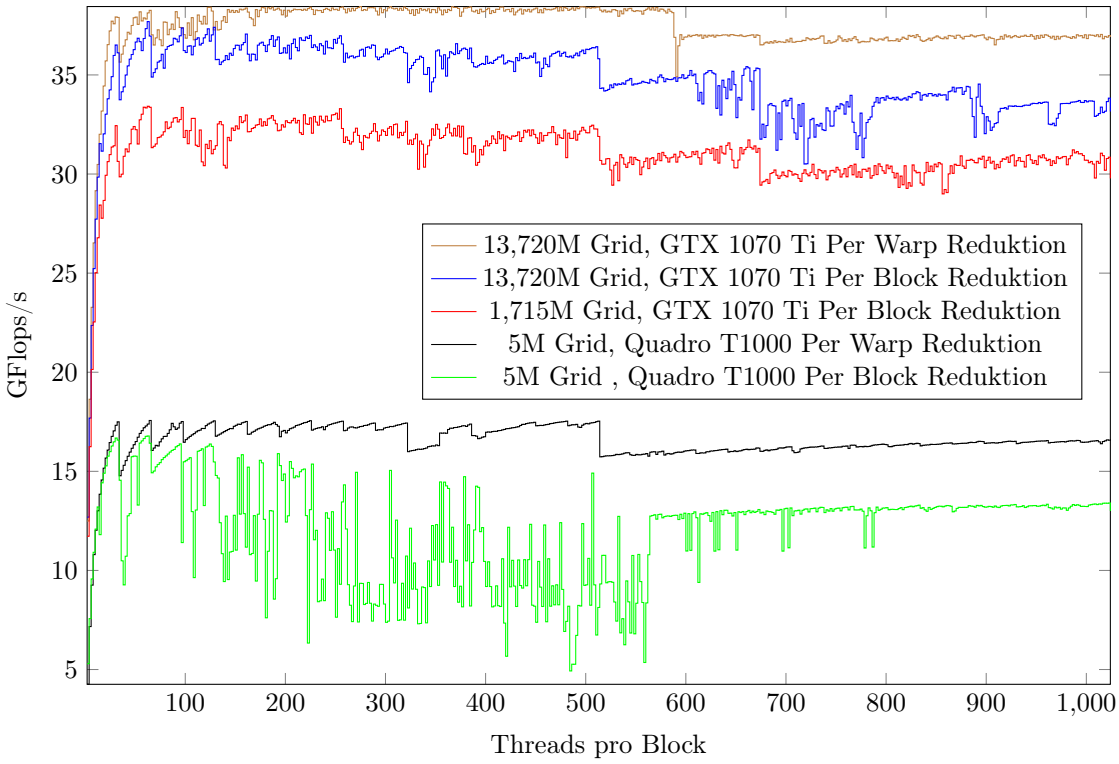

Abbildung 6: GFLOPS/s in Abhängigkeit von verschiedenen Blockgrößen(zwei bis 1024).

Aufgrund der allgemein niedrigeren Performance und der dementsprechend längeren Rechenzeit, wurde die Messreihe bei der Quadro T1000 nur für ein Gitter mit 5 Millionen Punkten durchgeführt. Von den Messungen gibt es jeweils zwei Typen, einerseits die Blockweise Bildung von Teilsummen im Skalarprodukt, andererseits die Bildung von Teilsummen je Warp. Informationen zu den unterschiedlichen Implementationen des Skalarprodukts finden sich im Kapitel **Optimierung**.

Die beiden Diagramme basieren auf derselben Datengrundlage, für das zweite Diagramm ist einzig und allein der X-Achsenbereich eingeschränkt worden, um den Verlauf der Graphen in diesem Bereich besser nachvollziehen zu können. Die Graphen für die Warpsummenbildung sind deutlich markanter, als für die Blocksummenbildung. Der Sägezahnartige Verlauf des Graphen, welcher sich mit Erkenntnissen aus Kapitel 3 deuten lassen, lässt sich dort besser erkennen. Die Performancemaxima befinden sich fast immer bei Blockgrößen, die ein Vielfaches von 32 sind. Der Grund ist folgender: Ein SM verarbeitet Threads immer in Einheiten von Warps, welche genau 32 Threads groß sind. Um genauer zu verdeutlichen, warum deswegen bei falscher Wahl der Blockgröße ein großer Performanceverlust entstehen kann, wird eine theoretische Überlegung durchgerechnet.

Abbildung 7: GFLOPS/s in Abhängigkeit von verschiedenen Blockgrößen(zwei bis 220).

- Wir rechnen auf einem Gitter mit p=5M Punkten.
- Blockgrößen(Threads pro Block) $b_1 = 32, b_2 = 33$

Daraus ergeben sich, um das ganze Gitter abzudecken, folgende Griddimensionen:

- $g_1 = \lfloor \frac{p}{b_1} \rfloor = \lfloor \frac{5000000}{32} \rfloor = 156250$ Blöcke
- $g_2 = \lfloor \frac{p}{b_2} \rfloor = \lfloor \frac{5000000}{33} \rfloor = 151515$ Blöcke

Wir nehmen an, ein Warp braucht für die Abarbeitung des Kernels in etwa o Operationen. Da wir für b_1 pro Block nur einen Warp abarbeiten müssen, haben wir pro Block einen Aufwand von o Operationen. Für b_2 werden pro Block zwei Warps ausgeführt, wir haben also pro Block einen Aufwand von $2 * o$ Operationen. Der eine überschüssige Thread belegt die Rechenleistung eines ganzen Warps. Letztendlich haben wir für b_2 zwar weniger Blocks, es müssen aber trotzdem mehr Berechnungen durchgeführt werden. Hiermit kann man auch den langsamen Wiederanstieg der Performance nach einem Einbruch erklären. Im Bereich der Blockgröße von 33 bis 64 werden pro Block immer noch $2 * o$ Operationen durchgeführt, die Anzahl der Blöcke sinkt aber mit steigender Blockgröße, also verringert sich die nötige Gesamtzahl an Berechnungen. Nach diesen Überlegungen könnte eine Allgemeine Formel für die nötigen Berechnungen etwa so aussehen:

$$y = \frac{p}{b} * o * \lfloor \frac{b}{32} + 1 \rfloor \tag{4}$$

Mit dieser Formel könnte man die insgesamt durchzuführende Anzahl an Rechenoperationen abschätzen. Hier wäre eine größere Zahl schlechter, bei der GFLOPS/s Kennzahl ist eine höhere Zahl besser. Um einen Vergleich möglich zu machen, nehmen wir den Kehrwert der Gleichung (4) und tragen das Ergebnis in ein Diagramm ein.

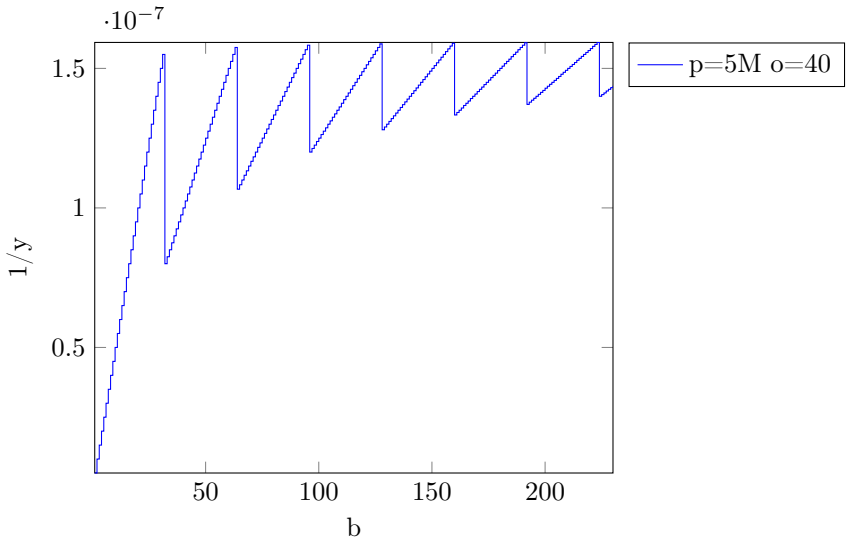

Abbildung 8: Kehrwert der Gleichung (4) als Graph.

Man erkennt, dass das Aufgestellte Berechnungsmodell Ähnlichkeiten zur Realität aufweist und man damit den Graphenverlauf aus Abbildung 7 teilweise nachvollziehen kann. Unterschiedliche Anzahl an Operationen pro Warps, und andere Einflussfaktoren bei der Verarbeitung von Kernel, wie z.B. Caching oder Warp Scheduling werden in diesem Modell nicht abgebildet. Ein weiterer Effekt der in der Performancemessung ersichtlich ist, aber nicht durch das Modell erklärt werden kann, ist folgender: Bei Blockgrößen von 321 und 513 ist ein starker Performanceverlust sichtbar, der sich im Bereich von 513 bis 1024 Threads pro Block nicht mehr erholt und das Sägezahnmuster ist nur noch deutlich schwächer ausgeprägt. Erklären lässt sich das Teilweise mit einem stark gesunkenen Occupancy Wert. Für die Blockgröße 512 liegt der Occupancy Wert bei 100%, für die Blockgröße 513 wird nur noch eine Occupancy von 80% erreicht und zusätzlich ist 513 auch kein Vielfaches von der Warpgröße 32 mehr. Allerdings wird bei einer Blockgröße von 672 eine Occupancy von 98% erreicht und die Blockgröße ist wieder ein Vielfaches von 32, aber die Performance steigt in diesem Bereich nicht merklich an. Hieran sieht man, dass eine hohe Occupancy nicht immer auch bessere GFLOPS/s Werte bedeutet. Bei der per Block Reduktion wird pro Block einmal eine atomicAdd Instruktion ausgeführt, bei der per Warp Reduktion, werden pro Block so viele atomicAdds ausgeführt, wie Warps in diesem Block vorhanden sind. Da atomicAdd eine relativ aufwändige Instruktion ist, könnte dies auch ein Grund sein, warum die Performance bei großen Blockdimensionen allgemein schlechter ist.

6.2 Ineffizientes Skalarprodukt

In der gelösten Differentialgleichung kommt ein Skalarprodukt vor, welches für das ganze Gitter ausgeführt wurde. Um zunächst einmal nur die Funktionstüchtigkeit der CUDA Berechnung zu testen, wurde dieses ohne Reduktion berechnet. Jeder einzelne Thread nahm sich vier Punktepaare im Gitter, hat diese miteinander multipliziert und aufaddiert. Die restliche Summation der Zwischenergbnisse wurde auf der CPU durchgeführt. Diese Version ist zwar bereits schneller als zuvor, aber immer noch langsamer als die Endversion, was sich im nächsten Kapitel zeigen wird.

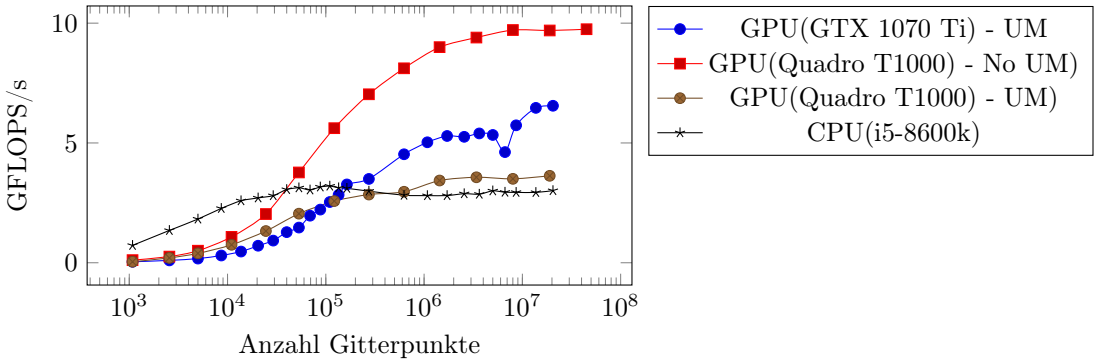

Abbildung 9: Messung bei ineffizienter Skalarprodukt Implementation.

6.3 Finales Ergebnis

6.3.1 Unified Memory

Die maximal erreichte Leistung liegt bei 23,35 GFLOPS/s für die Warpweise Reduktion durchgeführt auf der GTX 1070 Ti. Wie man im Diagramm sieht, ist bei den GPU Berechnungen bei etwa eine Millionen Gitterpunkte ein Plateau erreicht, ab dem die Leistung nicht mehr merklich ansteigt. Der Starke Leistungszuwachs beginnt bei etwa 40 Tausend Gitterpunkten, das ist auch in etwa der Wert, ab dem die Grafikkartenberechnung schneller wird, als die CPU Berechnung.

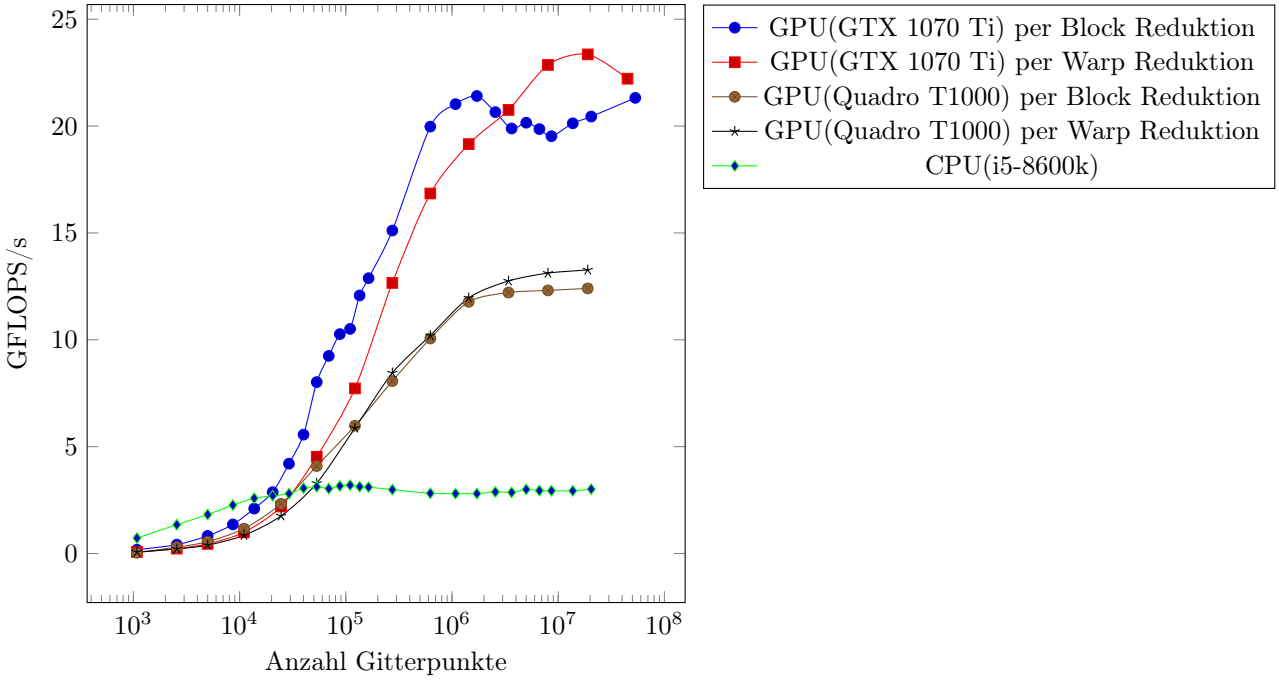

Abbildung 10: Messung der finalen Version mit Unified Memory.

6.3.2 Manuelle Speicherverwaltung

Der Graph weißt große Ähnlichkeiten zum Unified Memory Graph auf, vorallem hinsichtlich Plateau und Leistungsanstieg. Was besonders auffällt ist der Unterschied zwischen der GTX 1070 Ti und der Quadro T1000. Die beste Leistung der Quadro T1000 bei manueller Speicherverwaltung ist diese nur etwa 30% besser, als bei Unified Memory. Dagegen beträgt dieser Unterschied bei der GTX 1070 Ti etwa 70%. Die sinnvollste Erklärung findet sich beim Alter der Grafikkarten, so hat die Quadro T1000 eine Compute Capability von 7.5 und besitzt einen Chip der Turing Architektur, wohingegen die GTX 1070 Ti nur eine Compute Capability von 6.1 hat und einen Chip der älteren Pascal Architektur verbaut hat. Es ist anzunehmen, dass das Unified Memory Konzept für neuere Grafikkarten noch einmal stark verbessert wurde.

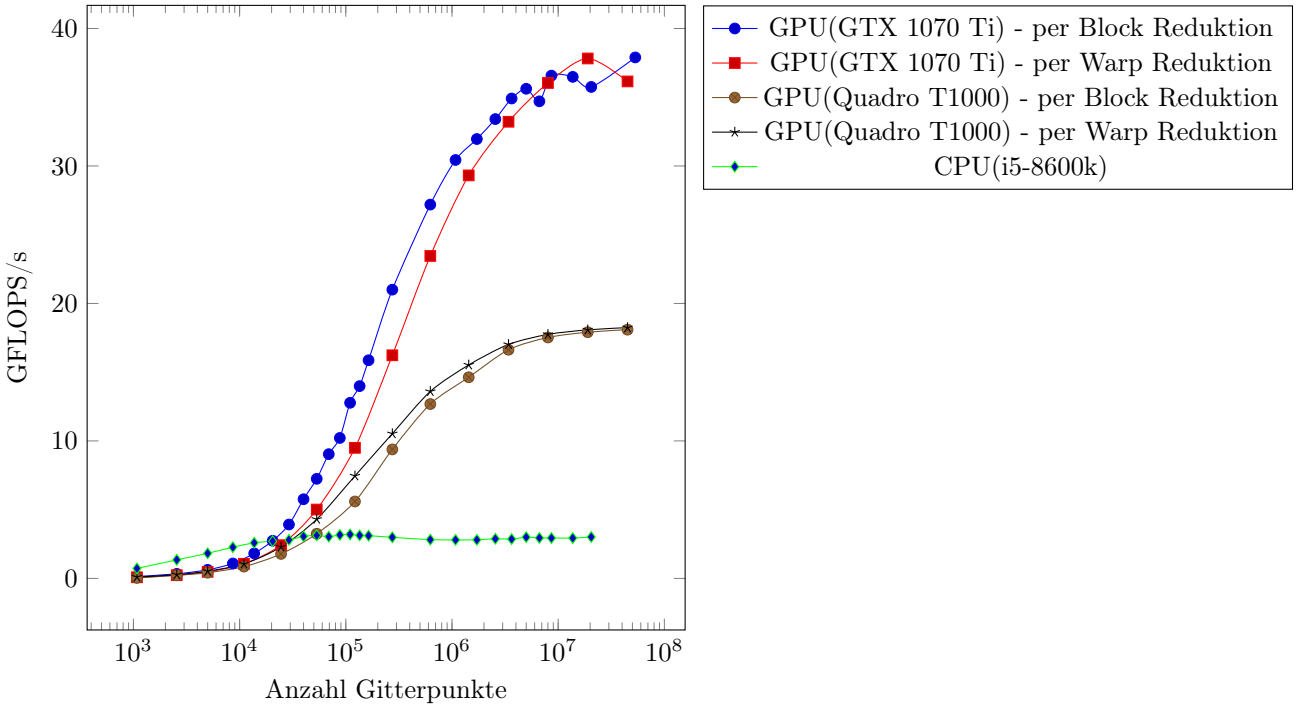

Abbildung 11: Messung der finalen Version mit manueller Speicherverwaltung.

7 Zusammenfassung

Der Leser wurde eingeführt in die Grundlagen der Grafikkartenprogrammierung und Expression Templates. Ausgehend von einer bestehenden Implementation der Expression Templates mit CUDA, wurde das Ziel die Bibliothek UGBlocks zu erweitern, erfolgreich umgesetzt. Hierdurch konnte eine deutliche Leistungssteigerung erreicht werden. Bei der Programmierung wurden verschiedenste Ideen umgesetzt, teilweise aber auch wieder verworfen, wenn das Ergebnis nicht zufriedenstellen war. Ein überraschender negativer Effekt wurde bei der Erzeugung einer kompletten Zellvariable mit Unified Memory bemerkt. Durch dieses Vorgehen sollte es wie ursprünglich wieder möglich werden, die Operanden in den Operationsklassen der Expressions mit Referenzen zu speichern. Dafür wurden die Operationsklassen und Zellvariablen mit Unified Memory erzeugt. Letztendlich konnte eine Expression also auf der GPU verwendet werden und die in ihr enthaltenen Referenzen waren immer noch gültig. Nach der Implementation stellte sich aber heraus, dass die Implementation sogar langsamer war, als die reine CPU Berechnung. Das Problem lag in der Verwendung der Variable cudaCalculation in einer Zellvariable. Auf diese wurde in jedem Iterationsschritt zugegriffen, was unnötige Speichermigration verursacht hat. Doch selbst wenn das funktioniert hätte, wäre die stetige Erzeugung von Objekten mit Unified Memory ein großes Problem geworden. Um nämlich Referenzen in den Operationsklassen zu verwenden, müssten die Operationsklassen selbst

auch mit Unified Memory erzeugt werden, da sie selbst ja auch in einer Expression vorkommen und demnach als Referenz gespeichert werden würden. Nicht nur die Umsetzung gestaltete sich schwierig, auch die Erzeugung einer ganzen Expression in jedem Iterationsschritt verlangsamte das Programm extrem. Die Verlagerung der Expressionerzeugung außerhalb der Iterationsschleife bot sich als eventuelle Lösung an, was bei genauerer Betrachtung aber auch verworfen werden musste. Die Berechnung ging zwar nun schnell vonstatten, lieferte aber falsche Werte. Tatsächlich verändern sich ja die Photonenanzahl in einer Expression, weswegen diese in jedem Iterationsschritt verändert werden müsste. Prinzipiell wäre das möglich mit einem Zugriff wie in Listing 13.

```
1 auto expr = (Npopinv + tau * pumping) / (1.0 + tau * (1.0 / tau_f +
      sigma_by_roundTrip * Phi * mode + pumping / NpopinvMax));
2 for(...){
3   expr.b_.b_.b_.a_.b_.a_.b_ = newPhi; //Change Phi
4
5   Npopinv = expr; //Evaluate Expression
6   ...
7 }
```

Listing 13: Anpassung von double Werten in einer Expression.

Dieser Code hat aber mit der angestrebten Anwenderfreundlichkeit nichts mehr zu tun und insofern wurde diese Idee auch verworfen. Letztendlich wurden nur noch die Gitterwerte mit Unified Memory erzeugt. Hierdurch ergaben sich weitaus weniger nötige Codeänderungen, als bei manueller Speicherverwaltung, leider aber war die Lösung nicht so Leistungsstark, wie erwartet. Der vorliegende Anwendungsfall hat eigentlich gleichbleibende Leistung suggeriert, immerhin werden die Daten der Cell_variablen während der Iterationsberechnung kein einziges mal auf der CPU benötigt und demnach sollte durch Unified Memory keine übermäßige Zeitverzögerung durch Speichermigration entstehen. Die Messungen haben aber gezeigt, dass die manuelle Speicherverwaltung trotzdem um einiges schneller ist. Ein kleiner Lichtblick ergibt sich aber durch die stetige Verbesserung von Nvidia, denn die neuere Quadro T1000 war mit manueller Speicherverwaltung nicht so viel schneller als mit Unified Memory. Es könnte sich deswegen durchaus lohnen, den Unified Memory Ansatz noch nicht zu verwerfen.

Literatur

[1] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, 2015.

[2] Ian Buck u. a. *Brook for GPUs: Stream Computing on Graphics Hardware*. 2004.

[3] John Cheng, Grossman Max und McKercher Ty. *Professional CUDA C Programming*. Wiley, 2014.

[4] Mark Harris. *How to Overlap Data Transfers in CUDA C/C++*. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/. Zugriff: 12.09.2020.

[5] Mark Harris. *Unified Memory for CUDA Beginners*. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Zugriff: 12.09.2020.

[6] Mark Harris und Kyrylo Perelygin. *Cooperative Groups: Flexible CUDA Thread Programming*. https://developer.nvidia.com/blog/cooperative-groups/. Zugriff: 12.09.2020.

[7] Jochen Härdtlein. *Moderne Expression Templates Programmierung*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg. 2007.

[8] Lou Knauer. *Expression Templates auf Grafikkarten*. Bachelorarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg. 2019.

[9] Annette Kunow. *Finite Elemente Methode, Anwendungen und Lösungen*. Hüthig Verlag, 1998.

[10] Eichhorn Marc. *Laserphysik*. Springer Spektrum, 2013.

[11] Nvidia. *CUDA C Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Zugriff: 12.09.2020.

[12] Nvidia. *CUDA Occupancy Calculator*. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html Zugriff: 12.09.2020.

[13] T. Veldhuizen. *Expression templates*. https://www.semanticscholar.org/paper/Expression-templates-Veldhuizen/8f41dfd51e4da3543ae27263fc1f469a05096730?p2df Zugriff: 12.09.2020. 1996.

[14] Paul Wiemann, Stephan Wenger und Marcus Magnor. "CUDA Expression Templates". In: *WSCG Communication Papers Proceedings*. ISBN 978-80-86943-82-4. 2011, S. 185–192.

[15] M. Wohlmuth u. a. "Dynamic multimode analysis of Q-switched solid state laser cavities". In: *Opt. Express* 17.20 (2009), S. 17303–17316.