Friedrich-Alexander-Universität Erlangen-Nürnberg

Technische Fakultät - Department Informatik

Lehrstuhl für Informatik 10 (Systemsimulation)

Expression Templates auf Grafikkarten

Lou Knauer

Bachelorarbeit

Expression Templates auf Grafikkarten

Lou Knauer

Bachelorarbeit

Aufgabensteller und Betreuer:	Prof. Dr. Christoph Pflaum
Bearbeitungszeitraum:	01.05.2019 - 30.09.2019

Der Quelltext für die Implementierung von Expression Templates auf Grafikkarten die mit dieser Arbeit enstanden ist kann unter `https://gitlab.cs.fau.de/ke17fisi/gpu-exprtmpls` eingesehen werden.

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 Forschug und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 18. September 2019 _____

Kurzzusammenfassung

Partielle Differenzialgleichungen und lineare Algebra sind für das Lösen mathematisch beschreibbarer Probleme aus vielen Wissenschaften sehr wichtig. Um Computerprogramme so schreiben zu können, dass sie den mathematischen Modellen und Lösungsformeln ähneln, wurden Expression Templates entwickelt. Diese erlauben es in C++ effizienten Code zu schreiben, der Formeln ähnelt und leicht verständlich ist. Oft wird nicht nur mit Skalaren gerechnet, sondern mit Vektoren, Matrizen oder Gittervariablen (bei Finiten Differenzen).

Ziel dieser Arbeit ist es, zu beschreiben, wie die Auswertung von Expression Templates auf der Grafikkarte parallelisiert werden kann. Grafikkarten eignen sich gut für hoch-parallelisierbare Aufgaben und erreichen oft eine höhere Leistung als normale Prozessoren. Sie werden allerdings grundsätzlich anders programmiert als Hauptprozessoren. Zusätzliche Komplexität durch Auswertung auf einem Beschleuniger soll vor dem Nutzer versteckt werden.

Wie in dieser Arbeit gezeigt wird, wurden die gesetzten Ziele erreicht und mithilfe moderner Übersetzer eine neue Möglichkeit zur Implementierung von Expression Templates für Grafikkarten entwickelt. Die Verwendung dieser hat in gezeigten Fällen keinen Nachteil gegenüber konventioneller Grafikkartenprogrammierung. Die Leistung von Programmen konnte im Vergleich zur Ausführung auf Hauptprozessoren deutlich gesteigert werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Lineare Algebra und partielle Differenzialgleichungen werden bei sehr vielen Problemlösungen der Physik, Chemie oder anderer Wissenschaften gebraucht. Häufig sind die Lösungsalgorithmen durch mathematische Formeln präzise beschrieben. Um also mit dem Computer Probleme wie zum Beispiel die Wärmeausgleichsrechnung zu simulieren oder zu lösen, müssen solche in ein Computerprogramm überführt werden. Bibliotheken wie verschiedene BLAS-Implementierungen bieten bereits Unterprogramme für häufig verwendete Operationen, wie die Matrix-Vektor-Multiplikation, an. Allerdings sind viele dieser Bibliotheken nicht sehr benutzerfreundlich und erlauben es nicht Code zu schreiben, der den mathematischen Formeln, die man implementieren möchte, direkt ähnelt. Das Ziel der bereits seit längerem bekannten und vor allem im Bereich des wissenschaftlichen Rechnens verwendeten Expression Templates[10] ist es, dem Programmierer zu erlauben, mathematische Formeln in ihm gewohnter Schreibweise verwenden zu können.

Eine weitere Entwicklung der letzten Jahre dürfte für Nutzer sehr interessant sein, die Programme schreiben, welche auf großen Datenmengen parallelisierbar arbeiten: Grafikkarten die sich frei programmieren lassen. Diese sind zwar komplizierter und nur eingeschränkter verwendbar als normale CPUs, dafür erreichen sie aber in vielen Anwendungsgebieten eine höhere Leistung.

1.1.1 Vorteile von Expression Templates

Expression Templates wurden ursprünglich entwickelt um optimierten Code in sehr einfacher Weise schreiben zu können. Aufgrund der Möglichkeiten, die Templates und Operatoren-Überladung, welche für Expression Templates unverzichtbar sind, in C++ bieten, wird sich diese Arbeit nur mit dieser Programmiersprache beschäftigen. Expression Templates sind an sich zwar keine Optimierungstechnik, erlauben es dem Benutzer aber sich selbst nicht über die Optimierung von mathematischen Ausdrücken im Code Gedanken machen zu müssen. Als Beispiel soll hier folgende Berechnung mit den drei Vektoren a, b und c dienen: $a = b * 2 + c$. Hier soll jedes Element von b verdoppelt werden und der resultierende Vektor mit c addiert werden. Würde man die Vektor-Klasse einfach um einen Additionsoperator und einen Multiplikationsoperator erweitern, die jeweils als Ergebnis einen neuen Vektoren zurückgeben, entstünde ein sehr unperformantes Programm: Es wird unnötiger Speicher für zwei nur temporär notwendige Vektoren angefordert und mit Zwischenergebnissen gefüllt (Als Ergebnis von $tmp1 = b * 2$ und $tmp2 = tmp1 + c$). Außerdem wird eine Berechnung, die auch in einem einzigen Schritt hätte durchgeführt werden können, in zwei einzelne Schleifen über die Einträge der Vektoren aufgeteilt. Optimierungsstechnicken, die man hier anwenden würde, heißen unter anderem *Loop Fusion*.

Hier setzen Expression Templates an: Sie erlauben es dem Programmierer über spezielle Klassen, Templates und Operatoren trotzdem $a = b * 2 + c$, oder noch beliebige andere Ausdrücke, zu schreiben und sorgen dafür, dass der Code nicht von Hand weiter optimiert werden muss. Die im oberen Absatz beschriebenen Probleme

und möglicherweise noch viele weitere Optimierungen werden von der Implementierung der Expression Templates übernommen. Dies bedeutet für den Entwickler, dass er sich besser auf sein Problem konzentrieren kann und sich weniger Gedanken um die Performance seines Programms machen muss. Außerdem kann so ein leichter verständlicher, schöner, mathematischer Code geschrieben werden, was bei der Verwendung von Bibliotheken wie BLAS nicht immer der Fall ist.

1.1.2 Vorteile von Grafikkarten

Grafikkarten sind grundsätzlich anders aufgebaut als CPUs. Sie waren ursprünglich, und sind zum Teil auch heute noch, Beschleuniger für sehr spezifische Aufgaben. Sie sollen unter anderem Daten, die von der CPU geliefert werden, so aufbereiten, dass diese anschließend als Bild auf einem Ausgabegerät angezeigt werden können. Dafür müssen sie auf jedem Pixel im Bild bzw. den Daten bestimmte Berechnungen und/oder Transformationen anwenden. Moderne Grafikkarten haben diese Fähigkeiten und Aufgaben immer noch, bieten aber noch viele weitere Möglichkeiten. So sind manche von ihnen programmierbar geworden (*General-Purpose Graphics Processing Units*, kurz GPGPUs) und lassen sich grundsätzlich für alle möglichen Berechnungen verwenden. Neben Bilddaten können sie auch effizient mit Matrizen oder Vektoren rechnen. Grafikkarten wurden dafür gebaut, hochgradig parallel auf Daten arbeiten zu können und erreichen durch eine hohe Anzahl an Recheneinheiten bessere Leistungen als CPUs. Einzelne Recheneinheiten sind allerdings weitaus langsamer und auch sonst nicht so performant wie ein einzelner Kern einer CPU.

Es existieren auch speziell für das wissenschaftliche Rechnen entwickelte Grafikkarten die keine Bildschirmausgabe mehr besitzen und nur noch als Koprozessor dienen. Diese werden in vielen der leistungsstärksten Supercomputern der Welt verbaut[1].

1.2 Ziele dieser Arbeit

Das Problem von GPGPUs ist allerdings, dass sie, wie viele andere Beschleuniger auch, schwieriger zu programmieren sind. Speicher auf der Grafikkarte muss vom Programmierer zusätzlich verwaltet werden. Parallelisierung auf der Grafikkarte ist schwieriger als auf dem Hauptprozessor und es stehen viele Synchronisationsmechanismen nicht bereit. Code für die Grafikkarte muss von Anfang an als parallelisierbar konzeptioniert werden. Standards wie *OpenACC*, welche die Grafikkartenprogrammierung einfacher gestalten wollen, sind bei weitem nicht so Verbreitet wie *OpenMP* (Eine Übersetzererweiterung zur einfacheren Parallelisierung auf CPUs). All diese zusätzliche Komplexität soll durch die Entwicklung einer Expression-Templates-Bibliothek, die auf der Grafikkarte ihre Berechnungen ausführt, wegabstrahiert werden. Der Nutzer soll sich bei dem Entwickeln seines Programms nicht mit den Feinheiten und Unterschieden der Grafikkartenprogrammierung beschäftigen müssen und weiterhin mathematischen Code schreiben können. Des Weiteren sollen auch bei dieser Implementierung wieder die Vorteile

[1] Top500-Liste vom Juni 2019: `https://www.top500.org/lists/2019/06/`

von Expression Templates, die auf der CPU ausgewertet werden, erhalten bleiben und das resultierende Programm effizient sein. Viele der existierenden Implementierungen für Hauptprozessoren nehmen dem Nutzer, zum Beispiel durch *OpenMP*, die Parallelisierung ab.

2 Grundlagen

2.1 Iteratives lösen von Laplace-Gleichungen

Expression Templates sind sehr flexibel einsetzbar, können aber auch als *Domain Specific Language*[3] gesehen werden. Die Bibliothek, welche mit dieser Arbeit entstanden ist, lässt sich zwar für verschiedenste Problemstellungen verwenden, allerdings soll hier speziell das Problem der Wärmeausgleichsrechnung behandelt werden. Es dient exemplarisch und zum testen der Implementierung. Die hier beschriebenen Effekte und Gleichungen tauchen auch bei vielen anderen Problemstellungen, wie dem Konzentrationsausgleich oder elektromagnetischen Feldern, auf.

Wenn es in einem System keine inneren Wärmequellen gibt und die Temperaturen an den Grenzen des Systems fest sind, dann kann die homogene Wärmeleitungsgleichung verwendet werden, um zu berechnen, in welchem Zustand sich ein System befindet:

$$\frac{\delta}{\delta t}u(\vec{x},t) - a\triangle u(\vec{x},t) = 0 \tag{1}$$

$u(\vec{x},t)$ ist hier die Temperatur an der Stelle $\vec{x}$ zum Zeitpunkt t. Am interessantesten ist hierbei der Zustand zu dem das System konvergiert, nach welchem sich also nichts mehr verändert. Dieser ist erreicht, wenn $\frac{\delta u}{\delta t}$ null ist. Möchte man diesen Zustand bestimmen, kann man auch folgende, einfachere Laplace-Gleichung lösen:

$$\triangle u = 0 \tag{2}$$

Sowohl diese Gleichung, als auch die homogene Wärmeleitungsgleichung sind partielle Differenzialgleichungen, welche numerisch mithilfe der Finite-Differenzen-Methode[9] gelöst werden können. Numerische Verfahren lösen solche Gleichungen häufig besser und sind weniger fehleranfällig, berechnen dafür aber keine exakte, sondern eine nur näherungsweise Lösung. Bei diesem Verfahren wird ein Gitter über das System gelegt und an jedem Gitterpunkt einzeln die Lösung der Differentialgleichung durch einen Differentialquotienten abgeschätzt. So kann ein Gleichungssystem aus den endlich vielen Gitterpunkten gebaut werden, welches gelöst werden kann.

Hier werden zweidimensionale Räume betrachtet, das Prinzip lässt sich allerdings auch ins Dreidimensionale übertragen. Der Maschenweite genannte Abstand zwischen zwei Punkten im Gitter sei h (alle Punkte sind in beide Dimensionen gleich weit voneinander entfernt). Die zweiten partiellen Ableitungen von u für den Laplace-Operator können durch die Differenzialquotienten abgeschätzt werden:

$$\frac{\partial^2}{\partial x^2}u(x,y,t) \approx \frac{u(x-h,y,t) - 2u(x,y,t) + u(x+h,y,t)}{h^2} \tag{3}$$

$$\frac{\partial^2}{\partial y^2}u(x,y,t) \approx \frac{u(x,y-h,t) - 2u(x,y,t) + u(x,y+h,t)}{h^2} \tag{4}$$

Diskretisiert man diese Gleichung für jeden Punkt im Gitter, baut sich daraus ein lineares Gleichungssystem auf. Die rechte Seite ist hier einfach ein Vektor aus Nullen. Die gesuchte Lösung ist ein Vektor aus allen Gitterpunkten. Die Matrix enthält in der Spalte i die Gewichtung aller anderen Punkte im Gitter für die Gleichung zum Gitterpunkt i. Wie man an den Differenzialquotienten erkennen kann werden nur die unmittelbaren Nachbarn im Gitter und der Punkt selbst berücksichtigt. Es entsteht also eine sehr dünn besetze Matrix, die auf den Diagonalen überall die selben Werte hat. Sie muss nicht komplett aufgestellt werden, sondern kann auch als Stern dargestellt werden, welcher auf jeden Gitterpunkt angewendet wird. Der Stern bildet nicht direkt die Matrix ab, sondern die räumliche Verteilung gewichteter Nachbarpunkte.

$$\frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5}$$

2.1.1 Jacobi-Verfahren

Das aufgestellte Gleichungssystem soll nun iterativ gelöst werden. Dies kann unter anderem durch Jacobi-Verfahren[9] erreicht werden, bei dem sich der Lösung schrittweise genähert wird, indem in jeder Iteration aus den Werten der vorherigen Iteration eine neue, bessere Annäherung berechnet wird. Hier spricht man auch wieder von Sternen, die auf einen Gitterpunkt angewendet werden. Sie beschreiben die Gewichtung der Nachbarn eines Punktes. Eine häufig verwendete Iterationsvorschrift lautet beispielsweise:

$$u_i(x,y) = (u_{i-1}(x+1,y) + u_{i-1}(x-1,y) + u_{i-1}(x,y+1) + u_{i-1}(x,y-1))/\frac{1}{4} \tag{6}$$

Wobei hier u_i die angenäherte Lösung für das Ausgleichsproblem in der aktuellen und u_{i-1} in der vorherigen Iteration darstellt. Anfangs bekannt sind nur die Randpunkte von u_0, welche danach nicht mehr verändert werden. Obige Iterationsvorschrift wird also nur auf die inneren Gitterpunkte angewendet bis u sich nicht mehr verändert. Diese Verfahren konvergieren relativ langsam, sind allerdings einfach zu implementieren und gut parallelisierbar.

2.1.2 Gauß-Seidel-Verfahren

Gauß-Seidel-Verfahren gehen grundsätzlich ähnlich vor, verwenden allerdings bereits berechnete Ergebnisse aus der aktuellen Iteration mit. Eine Iterationsvorschrift könnte also lauten:

$$u_i(x,y) = (u_{i-1}(x+1,y) + u_i(x-1,y) + u_{i-1}(x,y+1) + u_i(x,y-1))/\frac{1}{4} \tag{7}$$

So kann man Ergebnisse direkt in die selbe Datenstruktur zurückschreiben aus der man auch liest. Dies verkompliziert die Parallelisierung, verringert allerdings den Speicherverbrauch.

2.1.3 Verfahren der konjugierten Gradienten

Ein schneller konvergierendes Verfahren ist das der konjugierten Gradienten[7] (kurz CG). Es kann verwendet werden, um beliebige lineare Gleichungssysteme zu lösen, bei denen die Matrix im Gleichungssystem symmetrisch und positiv definit ist. Beides trifft auf die durch das Finite-Differenzen-Verfahren konstruierte Matrix zu. Allerdings muss diese nicht vollständig gespeichert werden, sondern es reicht zu wissen, wie diese auf einen Punkt im Lösungsvektor (hier die Gitterpunkte) angewendet wird: Durch den in 5 beschriebenen Stern.

Dabei wird statt des Gleichungssystems $Ax = b$ zu lösen versucht, die Funktion $E(x) = \frac{1}{2} < Ax, x > - < b, x >$ zu minimieren. Dies passiert ebenfalls iterativ. Dafür wird von einer Startlösung aus, über konjugierte Gradienten, die aktuelle Lösung schrittweise verbessert. Eine mögliche Implementierung wird in der Datei `src-cuda/cg.cu` gezeigt. Genaugenommen ist dieses Verfahren ein exakter Löser, meist konvergiert es allerdings bereits vor den maximal benötigten n Schritten, wobei n die Größe der Matrix A ist.

2.2 Implementierung von Expression Templates auf der CPU

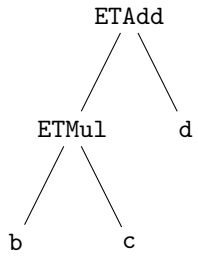
Templates erlauben es in C++ generische Funktionen und Klassen zu erstellen, wobei anders als bei Java nicht auf dynamische Polymorphie gesetzt wird, sondern durch verschiedene Template-Parameter (meist Typen) neue Typen entstehen. Ein Klassen-Template kann also als Vorlage für verschiedene tatsächliche Klassen gesehen werden, alle mit unterschiedlichen Template-Parametern, welche zur Übersetzungszeit bekannt sein müssen. Deshalb kann der Code auch besser Optimiert werden als beispielsweise bei Java-Generics. Nachteile sind größere Binärdateien, da pro Template mehrere Klassen oder Funktionen instantiiert werden.

Um zu verstehen, wie Expression Templates funktionieren, die auf der Grafikkarte ausgewertet werden, sollte man sich erst zumindest darüber bewusst sein, wie sie normalerweise funktionieren. Die zugrundeliegende Idee besteht darin, Operationen, wie die Addition oder Multiplikation von zwei Objekten nicht direkt auszuführen, sondern stattdessen in einer extra Klasse zu abstrahieren, welche die Operation zwar beschreibt, aber nicht direkt ausführt. Für jede Operation wird ein Klassentemplate angelegt, welches für diese steht. Anschließend werden alle Operatoren, die in Expression Templates vorkommen dürfen, so überladen, dass sie die Instanz eines solchen Klassentemplates zurückgeben, bei dem die beiden Operanden jeweils auch wieder zusammengesetzte Ausdrücke oder Terminale wie Vektoren oder Matrizen sind. Da in dieser Arbeit Wärmeleitungsprobleme gelöst werden sollen, sind die verwendeten Datenstrukturen, welche tatsächliche Werte speichern, Skalare und Gittervariablen. Mit Gittervariablen kann gerechnet werden wie mit Vektoren, sie speichern aber für jeden Punkt im hier zweidimensionalen Gitter einen diskretisierten Wert.

Als Beispiel soll der Ausdruck $a = b * c + d$ dienen. Hier kommen die Gittervariablen (oder Vektoren) a, b, c und d vor, welche alle die selbe Anzahl an Elementen speichern. Alle Operationen sollen wieder elementweise ausgeführt werden. In diesem Ausdruck werden zwei zusätzliche Klassentemplates benötigt, um die Multiplikation und die Addition darzustellen:

```
template<typename T>
struct ET {
  operator const T&() const {
    return *static_cast<const T*>(this);
  }
};

struct GridVariable: public ET<GridVariable> { /* ... */ };

template<typename LHS, typename RHS>
struct ETAdd: public ET<ETAdd<LHS, RHS>> {
  const LHS &lhs;
  const RHS &rhs;

  // ...
};

template<typename LHS, typename RHS>
struct ETMul: public ET<ETMul<LHS, RHS>> {
  const LHS &lhs;
  const RHS &rhs;

  // ...
};
```

Die zusätzlich eingeführte Struktur `ET` dient als Basisklasse für alle Ausdrücke die in Expression Templates vorkommen dürfen. Dies vereinfacht bei der Implementierung unter anderem das Überladen der Operatoren wie `+` und `*`. `ET` ist ein Template welches selbst den Typ der Struktur, die von ihr erbt, als Templateparameter erhält. Dabei handelt es sich um das *Curiously recurring template pattern* (es dient hier hauptsächlich der statischen Polymorphie), welches für das Verständnis von Expression Templates allerdings nicht unbedingt bekannt sein muss. Hier wichtig zu erkennen ist, dass die beiden Attribute `lhs` und `rhs` selbst wieder zusammengesetzte Strukturen vom Typ `ETAdd` oder `ETMul` sein können. So entsteht aus $b * c + d$ eine Struktur vom Typ `ETAdd<ETMul<GridVariable, GridVariable>, GridVariable>`. Es wird also bereits zur Übersetzungszeit aus jedem Ausdruck ein für diesen spezifischer Typ erstellt. Dieser spiegelt den abstrakten Syntaxbaum wieder, der aus dem Ausdruck gebildet wird und der hier graphisch dargestellt ist:



Um den Ausdruck auswerten zu können, bekommt jeder Knoten in diesem Baum, repräsentiert durch eine von `ET` erbende Struktur oder Klasse, eine Methode, die das Ergebnis für sich ausrechnet. Dies passiert allerdings vorerst nur an einer Stelle im Gitter. Handelt es sich um einen Knoten mit Nachfolgern, werden wie bei einer rekursiven Tiefensuche, erst diese ausgewertet und die Ergebnisse verrechnet zurückgegeben. Für den Additionsknoten könnte dies zum Beispiel so aussehen: `lhs.get(x, y) + rhs.get(x, y)`, wobei `x` und `y` die Stelle im Gitter sein soll, an der ausgewertet wird und `get` der Name der Methode ist, die jedes Klassentemplate zur Auswertung hat. Für eine Gittervariable selbst wird der Wert an dieser Stelle im Gitter zurückgegeben.

Der Zuweisungsoperator einer Gittervariable wird so implementiert, dass er ein beliebiges Expression Template entgegennimmt und für jeden Punkt im Gitter den Ausdruck auswertet und speichert. Da der Typ des Ausdrucks bereits zur Übersetzungszeit bekannt ist, ist es dem Übersetzer problemlos möglich die Funktionsaufrufe an die einzelnen `get`-Methoden durch *inlining* zu eliminieren und gegebenenfalls noch weiter zu optimieren[3]. Es werden auch keine temporären Strukturen angelegt oder die Berechnung in mehrere Schleifen aufgeteilt, weil jeweils eine Position im Gitter für sich vollständig ausgewertet wird.

```
template<Expr>
void GridVariable::operator = (const ET<Expr> &expr) {
  for (int x = 0; x < width; ++x)
    for (int y = 0; y < height; ++y)
      this->values[x][y] = expr.get(x, y);
}
```

2.3 Grafikkartenprogrammierung

Grafikkarten sind in ihrer Bauweise zwar meist ähnlich, aber nicht alle gleich. Auch wenn die Konzepte auf die meisten Architekturen und sogar auf einige andere Beschleuniger zutreffen, wird hier hauptsächlich auf Nvidia-Architekturen eingegangen, da diese auch verwendet wurden, um die Implementierungen zu testen. Über einen Bus wie *PCIe* wird die Grafikkarte an den Hauptprozessor angebunden. Ein Programm, das auf dem Hauptprozessor läuft, kann über Treiber und den verwendeten Bus mit ihr kommunizieren. Die Grafikkarte selbst ist aufgeteilt in mehrere Multiprozessoren (Von Nvidia *streaming multiprocessors* (kurz SM) genannt), welche unabhängig voneinander Blöcke ausführen können. Ein Block wiederum besteht aus mehreren Threads (vergleichbar mit den Threads auf der CPU, allerdings

deutlich leichtgewichtiger). Diese werden innerhalb eines Multiprozessoren immer in Gruppen von 32 (eine solche Gruppe wird *Warp* genannt) gleichzeitig ausgeführt, wobei mehrere Gruppen parallel abgearbeitet werden können. Grundsätzlich sind Grafikkarten deutlich langsamer getaktet als CPUs, haben aber viel mehr Rechenkerne. Diese sind aber einfacher aufgebaut und haben weniger Optimierungsmechanismen. So kann man sich auch vorstellen, dass ein *Warp* auf einem einzigen Kern ausgeführt wird, welcher auf 32 Einträgen breiten *SIMD*-Registern operiert. Deutlich wird dies unter anderem dadurch, dass alle Threads in einem *Warp* immer die selbe Instruktion gemeinsam ausführen müssen. Sollte sich der Kontrollfluss von Threads unterscheiden, wird der *Warp* aufgeteilt. Dies führt dazu, dass in einem *Warp* einige Threads warten müssen, was die Leistung verringert. Durch diese Einschränkungen können sich aber Instuktionscache, Programmzähler und andere Bauteile geteilt oder stark vereinfacht werden. Außerdem verzichten Grafikkarten-Kerne auf Techniken wie mit der CPU vergleichbares *Pipelining* oder spekulative Ausführung. Eine Synchronisation zwischen allen Threads in einem Block ist möglich, nicht aber mit verschiedenen Blöcken[2]. Eine Operation oder Berechnung, die man vom Hauptprozessor aus auf der Grafikkarte startet, nennt man Kernel.

Da Threads innerhalb eines Blockes und auch die Blöcke selbst nicht nur linear durchnummeriert werden können, spricht man bei der Aufteilung der Threads auf die Grafikkarte auch von einem Gitter. Bei Nvidia-Grafikkarten kann man sie üblicherweise in bis zu drei Dimensionen organisieren.

Der Speicher einer Grafikkarte ist ebenfalls unterteilt und anders organisiert als der des Hauptprozessors. Er ist häufig kleiner als der Hauptspeicher, allerdings dafür auch schneller, unter anderem weil er meist in den Chip der Grafikkarte mit eingebaut ist. Häufig wird, wie bei *GDDR5 SDRAM*, auch eine andere Technologie verwendet als für den Hauptspeicher. Der Speicherbereich, den sich alle Threads, egal in welchen Blöcken, teilen, nennt man global. Dieser kann beliebig gelesen und geschrieben werden, ist allerdings auch am langsamsten. Blöcke teilen sich, was man geteilten oder auch lokalen Speicher nennt. Dieser Speicherbereich kann nur von Threads in einem Block gelesen und geschrieben werden, wobei auf ihn schneller zugegriffen werden kann als auf globalen Speicher. Er ist allerdings auch deutlich kleiner. Er kann als expliziter Cache gesehen werden und liegt noch näher an den Ausführungseinheiten als der globale Speicher. Jeder Thread verfügt außerdem noch über Register, die frei verwendet werden können. Es gibt noch weitere Speicherbereiche, wie solche die von der Grafikkarte nur gelesen werden können (konstant genannt) oder die speziell für Texturen bei der Bildverarbeitung gedacht sind, diese werden hier allerdings nicht benötigt und deshalb außen vor gelassen.

Der Programmierer muss die Aufteilung der Threads in Blöcke selbst vornehmen und auch entscheiden, welche Daten in globalen, konstanten oder geteilten Speicher hinterlegt werden sollen und diesen allozieren. Den Transfer von Daten aus dem Hauptspeicher der CPU über einen Bus in den Speicher der Grafikkarte (und zurück) muss ebenfalls der Programmierer organisieren. Wie dies genau abläuft hängt vom Hersteller und der Architektur ab, es gibt allerdings Bibliotheken die Schnittstellen dafür bereitstellen. Für die hier vorgestellten Implementierungen von

[2]Wenn auch mit hohen Kosten bietet Cuda 9 so etwas nun doch an: Siehe *C.3. Grid Synchronization* in [6]

Expression Templates auf der Grafikkarte wurden drei verschiedene Technologien ausprobiert und diese werden im folgenden kurz vorgestellt.

2.3.1 Cuda

Cuda[6] ist eine von Nvidia entwickelte proprietäre Schnittstelle und Erweiterung der Programmiersprache C++ zum Entwickeln von Software für Nvidia-Grafikkarten. Sie führt neue Annotationen für Funktionen oder Methoden ein. Mit `__device__` annotierte Funktionen können nur auf der Grafikkarte ausgeführt werden und dürfen auch nur von Funktionen aufgerufen werden, die dort bereits laufen. Die Annotation `__global__` bedeutet, dass diese Funktion auf der Grafikkarte ausgeführt wird, allerdings über eine ebenfalls von Cuda eingeführte spezielle Syntax vom Hauptprozessor aus aufgerufen wird. Das sind die Kernels welche bereits erwähnt wurden. Sie dürfen keinen Rückgabewert haben und erhalten beim Aufruf zusätzliche Startparameter. Darunter müssen sich die Anzahl der Blöcke und die der Threads pro Block befinden, die gestartet werden sollen. Die Aufteilung kann in einem bis zu dreidimensionalen Gitter stattfinden. Weitere Parameter können die Menge an benötigtem geteiltem Speicher und der Stream sein. Alle Kernel-Aufrufe im selben Stream werden hintereinander ausgeführt. Das Starten eines Kernels verläuft asynchron, kehrt die Funktion also für den Aufrufer zurück, bedeutet dies nicht, dass die Berechnung bereits beendet ist. Sie wurde lediglich über die Treiberschnittstelle bei der Grafikkarte zur Ausführung eingereiht. Mit `__host__` annotierte Funktionen entsprechen dem Standardfall: Solche dürfen nur von Code aufgerufen werden, der auf dem Hauptprozessor läuft und werden auch dort ausgeführt. Es ist möglich eine Funktion sowohl mit `__device__` als auch mit `__host__` zu markieren, dann wird sie getrennt zweimal übersetzt (Für Hauptprozessor und Grafikkarte). Sie können dann von beiden Prozessoren aus verwendet werden. Über Präprozessormakros kann unterschieden werden für welchen Prozessoren übersetzt wird und unterschiedliches Verhalten herbeigeführt werden.

Die Variablen `threadIdx` und `blockIdx` erlauben es einem Thread seine Position im Gitter an gestarteten Threads zu ermitteln. Diese sind in jeder Funktion definiert, die auf der Grafikkarte ausgeführt wird. Sie haben jeweils eine x, y und z-Koordinate, wobei je nach Aufteilung y und z auch immer Null sein können.

Über Bibliotheksfunktionen wird es ermöglicht globalen Speicher auf der Grafikkarte anzufordern und Daten sowohl aus dem Hauptspeicher in den Grafikkartenspeicher als auch zurück zu kopieren. Speicheroperationen sind in Cuda standardmäßig blockierend. Wird also von der Grafikkarte Speicher zurück in den Hauptspeicher kopiert, wartet der Hauptprozessorthread solange, bis alle laufenden Kernels auf der Grafikkarte beendet worden sind und setzt sich erst fort, wenn der Speicher vollständig kopiert wurde. Die Bibliotheksfunktion blockiert. Dies gilt für alle Vorgänge die auf dem selben Stream agieren. Auf verschiedenen Streams kann unabhängig voneinander gearbeitet werden. Um mehrere Vorgänge überlappen zu lassen existieren auch asynchrone, nicht blockierende Bibliotheksfunktion.

Übersetzt man Cuda wird der Quelltext, welcher später auf der Grafikkarte ausgeführt werden soll, zu *PTX*-Binärcode, der im Standardfall in der selben Objektdatei wie der von einem normalen C- oder C++-Kompilierer übersetzte Quelltext

für den Hauptprozessor gespeichert wird. In früheren Versionen von Cuda wurde für Code, der auf der Grafikkarte läuft, nur C unterstützt. Mit fortschreitender Entwicklung von Cuda wurde allerdings immer mehr C++-Programmierung möglich. Sowohl Funktionen, die auf dem Hauptprozessor ausgewertet werden, als auch Programmcode für die Grafikkarte, können in die selbe Quelltextdatei geschrieben und durch Annotationen unterschieden werden.

2.3.2 OpenCL

OpenCL[4] ist ein Offener und von der *Khronos Group* entwickelter Standard für die Programmierung von verschiedenen Beschleunigern und Koprozessoren. Es existieren Implementierungen von den Herstellern von Grafikkarten, unter anderem Intel, AMD und Nvidia. OpenCL beschreibt eine Schnittstelle für C oder C++, mit der Programme, welche auf dem Hauptprozessor laufen, den Koprozessor ansteuern können. Es existieren zusätzlich Anbindungen der Schnittstelle für Programmiersprachen wie Python, Rust, Haskell und viele andere. Für den Koprozessor, hier eine Grafikkarte, existiert eine gesonderte Programmiersprache namens *OpenCL C* welche auf C99 aufbaut. Mit dem Schlüsselwort `kernel` markierte Funktionen in *OpenCL C* können über die dazugehörige Schnittstelle vom Hauptprozessor aus gestartet und auf dem Beschleuniger ausgeführt werden. Wie bei Cuda wird die Verwaltung von Speicher und Datentransfer durch den Hauptprozessor vorgenommen. Da verschiedene Programmiersprachen zum Einsatz kommen, kann *OpenCL C* Code, der auf der Grafikkarte ausgeführt wird, nicht direkt in die selbe Quelltextdatei geschrieben werden wie der Rest des Programms. Dafür ist ein Übersetzer für diese Sprache Teil der Laufzeitumgebung von OpenCL. Zum Programmstart, oder auch später, kann ein als Zeichenkette beschriebener Kernel in auf der Grafikkarte ausführbaren Binärcode übersetzt werden. Es existieren allerdings auch Zwischenrepräsentationen für vorkompiliertes *OpenCL C*. Aufgrund der gewünschten Plattformunabhängigkeit wird allerdings zur Übersetzungszeit des Hauptprogramms kein direkt ausführbarer Binärcode für den Beschleuniger erzeugt.

Beim Starten eines Kernels wird, wie bei Cuda, die gewünschte Aufteilung der Threads in Blöcke angegeben und, falls benötigt, die Menge an geteiltem Speicher. Anstatt von Streams spricht OpenCL von verschiedenen Warteschlangen, welche die selbe Funktionalität übernehmen.

2.3.3 SYCL

Einer der großen Nachteile von OpenCL im Vergleich zu Cuda ist, dass in der selben Quelltextdatei nicht Ko- und Hauptprozessorcode gemeinsam entwickelt werden können. Außerdem ist keine Grafikkartenprogrammierung in C++ möglich, sondern nur im einschränkenden *OpenCL C*. Nachteile von Cuda sind, dass es sich um einen herstellerspezifischen Ansatz handelt und nicht ohne weiteres das geschriebene Programm auf anderen Prozessoren (auch dem Hauptprozessor) ausgeführt werden kann, beispielsweise weil dort besser Fehler gefunden werden können. Außerdem sind zusätzliche Annotationen notwendig. SYCL[5] ist der Versuch diese Probleme zu lösen. Es ist die neueste der hier verwendeten Techniken und ebenfalls ein offener, von der *Khronos Group* entwickelter, Standard. SYCL erlaubt das

Programmieren in C++14, wobei ohne zusätzliche Annotationen Programmcode für den Haupt- und den Koprozessor in die selbe Quelltextdatei geschrieben werden können. Um SYCL zu übersetzen wird ein spezieller Kompilierer benötigt. Es existieren bereits mehrere Implementierungen eines solchen, der dann getrennten Code als Zwischenschritt für beide Prozessoren erzeugt. Dieser kann getrennt und für die jeweilige Plattform weiter übersetzt werden. Ursprünglich war gedacht SYCL auf OpenCL aufbauen zu lassen, es existieren allerdings auch Implementierungen die direkt Nvidia-*PTX*-Code erzeugen. Da SYCL speziell für modernes C++ entwickelt wurde verwendet es viele moderne Eigenschaften und Paradigmen wie Lambda-Ausdrücke.

Die Speicherverwaltung wird komplett von der SYCL-Laufzeitumgebung und den Bibliotheksfunktionen übernommen, der Speicher muss nicht speziell für die Grafikkarte angefordert werden. Dies ist möglich, weil SYCL keinen direkten Speicherzugriff erlaubt, sondern von der Klasse `cl::sycl::buffer` verwalteter Speicher nur über sogenannte Akzessoren verwendet werden kann. Bevor ein Kernel auf der Grafikkarte ausgeführt wird, muss für jeden Puffer, dessen Speicher verwendet werden soll, ein Akzessor angelegt werden. Dabei wird auch angegeben, wie auf den Speicher zugegriffen wird (lesend und/oder schreibend). Ein Puffer kann sowohl geteilten, als auch globalen Speicher verwalten. SYCL verwendet hier die Terminologie von OpenCL und nennt geteilten auch lokalen Speicher. Über den Akzessor kann die Laufzeitumgebung erkennen, ob der Speicher von dem Beschleuniger oder vom Hauptprozessor aus verwendet wird und kopiert ihn automatisch an die entsprechende Stelle. Nutzt man mehrere Beschleuniger in einem Programm werden auch deren getrennte Speicher automatisch verwaltet.

Die Grafikkarte, es ist auch möglich mehrere zu verwenden, wird über Geräte-Warteschlangen des Typs `cl::sycl::queue` angesprochen. Diese Klasse übernimmt die Synchronisierung und Verwaltung von verschiedenen Aufträgen an einen Beschleuniger. Um einen Kernel zu starten übergibt man der Laufzeitumgebung über eine Geräte-Warteschlange einen Lambda-Ausdruck. Dieser erhält ein spezielles Argument und wird auf dem Hauptprozessor ausgeführt. Mithilfe des Argumentes ist es möglich, sich Akzessoren für alle benötigten Daten anzulegen und ein weiteres Lambda an eine Methode wie `parallel_for` zu übergeben. Dieses zweite Lambda wird dann auf dem Koprozessor ausgeführt und läuft potenziell hoch-parallel in vielen Threads. Die Anzahl der Threads und optional auch die Aufteilung in Blöcke, können beim Aufruf von `parallel_for` angegeben werden. Über ein Argument wird den Threads ihre Position angegeben. Da dieser Code potenziell in einem anderen Adressraum ausgeführt wird, ist es nicht möglich von diesem Lambda aus noch Referenzen oder Zeiger auf Objekte außerhalb davon zu verwenden.

SYCL bietet also mehr Komfort als OpenCL oder Cuda: Vollständiges C++, einheitliche Programmierung, keine zusätzlichen Annotationen und weniger komplexe Speicherverwaltung. Zum gegenwärtigen Zeitpunkt existieren allerdings nur wenige Implementierungen, welche zudem meist noch in der Testphase oder nicht für den produktiven Einsatz gedacht sind.

3 Implementierungen

Insgesamt wurden für diese Arbeit drei Implementierungen für Expression Templates auf Grafikkarten entwickelt. Um zu verstehen, warum die OpenCL-Variante sich grundsätzlich von den anderen beiden unterscheidet, sollte man sich darüber bewusst sein, welche Eigenschaften eine Sprache für Expression Templates haben muss. Es ist notwendig Klassen- und auch Methodentemplates verwenden zu können, um die einzelnen Operationen und den Zuweisungsoperator, wie bereits beschrieben, implementieren zu können. Außerdem müssen der Code für das Expression Template und Methoden die es auswerten in die selbe Quelltextdatei geschrieben werden können, da in C++ Templates nicht deklariert werden dürfen, ohne auch alle Methoden und Funktionen in der selben Übersetzungseinheit zu definieren. Externe Template-Definitionen sind nicht praktikabel, da die Typen eines Expression Templates sehr unterschiedlich und vielfältig sein können.

Unter `samples/` finden sich minimalistische und dokumentiere Implementierungen der hier beschriebenen Herangehensweisen.

3.1 Implementierungsmöglichkeit unter OpenCL und alten Cuda-Versionen

Da bei OpenCL die Programmierung der Kernels in einer anderen Programmiersprache und in einer anderen Quelltextdatei geschieht, ist es hier nicht möglich Templates, die Code für den Haupt- und den Koprozessor mischen, zu entwickeln. Cuda, anfangs eine Erweiterung von C[3], hat die Unterstützung für C++-Klassen und Templates erst später Stückweise bekommen. In frühen Versionen waren zwar bereits einfache Template-Kernel möglich, allerdings noch keine Lambda-Ausdrücke, richtige Template-Metaprogrammierung oder vollständige Unterstützung für Klassentemplates. Insbesondere für den auf Grafikkarten auszuführenden Code haben diese Merkmale gefehlt.

Um diese Probleme zu lösen wurde sich folgendes überlegt: Die Expression Templates werden wie auf der CPU gebildet und instanziiert. Dieser Ausdruck kann allerdings nicht direkt auf einer Grafikkarte ausgewertet werden. Stattdessen wird aus dem Expression Template zur Laufzeit des Programms erst der Quelltext in Cuda oder OpenCL C generiert und dieser dann für die verwendete Plattform übersetzt. So kann dann mit den ebenfalls im Expression Template eingetragenen Werten und Datensätzen dieser frisch übersetzte Kernel vom Zuweisungsoperator gestartet werden.

Sämtliche vom Autor bereits gefundenen Implementierungen funktionieren nach dem hier im Folgenden beschriebenen Prinzip. Die erste ist ein sehr minimalistisches und eingeschränktes *Proof of Concept* der TU Braunschweig namens *cuda-Vec*[11], welches nur Cuda verwendet. Mit einer ähnlichen Herangehensweise wurde auch schon eine Technik für OpenCL vorgestellt[1]. Des Weiteren existiert noch die Bibliothek *VexCL*, welche aktiv entwickelt wird und mit öffentlich einsehbarem

[3]Siehe `http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`

Quelltext zur Verfügung steht. Diese wurde allerdings nicht verwendet, da eine eigene, möglichst neue Implementierungstechnik gefunden werden sollte.

3.1.1 Programmcode generieren

Der erste Schritt bei dieser Variante von Expression Templates für Grafikkarten ist es, aus dem Ausdruck den Quelltext zu generieren. Das Expression Template stellt den Ausdruck bereits ähnlich wie einen abstrakten Syntaxbaum dar. Als Beispiel soll hier wieder der Ausdruck $a = b * c + d$ dienen, welcher wie bereits erläutert den Typ `ETAdd<ETMul<GridVariable, GridVariable>, GridVariable>` hat. Um daraus einen lauffähigen und übersetzbaren Kernel zu generieren muss festgestellt werden, wie viele Terminale (Gittervariablen oder Skalare) in dem Ausdruck vorkommen, da diese dem Kernel später als Argumente übergeben werden müssen. Zusätzlich müssen gegebenenfalls noch Informationen wie die Größe übergeben werden. Sobald man die Signatur des Kernels kennt, kann man den Körper der Funktion erzeugen. Zuallererst muss dort die Position des Threads in dem Gitter an Blöcken ermittelt werden. Hierfür stehen unter Cuda wie bereits erwähnt `threadIdx` und `blockIdx` zur Verfügung, OpenCL bietet dafür die Funktion `get_global_id` an. Jeder Thread braucht diese Information um zu wissen, auf welche Werte im Gitter an diskreditierten Punkten für die Wärmeleitungsgleichungen er zugreifen muss. Da sich die Punkte, auf die zugegriffen werden muss, auch relativ verschoben zur eigenen Position befinden können, beispielsweise wenn auf den Punkt oberhalb des aktuellen Gitterpunktes zugegriffen werden soll, muss es möglich sein, auch neue Positionen zu berechnen.

Damit jeder Knoten im abstrakten Syntaxbaum in Programmcode überführt werden kann müssen alle seine Unterknoten bereits überführt worden sein. Der Name der Variable, in die das Ergebnis eines Unterknotens geschrieben wird, muss zurückgegeben werden. Mithilfe dieses Namens kann anschließend der Code generiert werden, der die Auswertung dieses Knotens darstellt. Handelt es sich um Knoten ohne Nachfolger, wie einen Vektor, wird nur der Wert an der entsprechenden Position in eine neue Variable gespeichert. Der Name des Index an dem gerade gerechnet werden soll wird von Knoten mit Nachfolgern weitergereicht. Die Abfolge in der ein Expression-Template-Ausdruck abgearbeitet werden muss ist die Selbe wie bei einer Tiefensuche. Im Folgenden ist ein generierter Kernel gezeigt, bei dem in Kommentaren am Ende einer Zeile steht, für welchen Knoten im Syntaxbaum diese Zeile das (Zwischen-)Ergebnis berechnet. Die Gittervariablen hier sind der Einfachheit wegen nur eindimensional.

```
kernel void et_kernel(/* ... */) {
  size_t i = get_global_id(0);

  double tmp0 = b[i];          // b
  double tmp1 = c[i];          // c
  double tmp2 = tmp0 * tmp1;   // b * c
  double tmp3 = d[i];          // d
  double tmp4 = tmp2 + tmp3;   // b * c + d
```

```
  a[i] = tmp4;
}
```

Eine mögliche Implementierung, die den Ausdruck für die Addition in Programmcode überführt, ist in 3.1.1 gezeigt. In der Klasse `Builder` befindet sich noch weiterer Zustand, wie Namen der Variablen für die Position im Gitter oder eine Abbildung der Vektoren auf Argumentnamen des Kernels, gespeichert. Jedes Klassentemplate und jede Struktur, die in einem Expression Template vorkommt, muss über eine solche Funktion verfügen. Diese erzeugt dann jeweils für diese Operation spezifischen Code.

```
template<typename LHS, typename RHS>
std::string ETAdd<LHS, RHS>::build(Builder &builder) {
  std::string lhs_res = this->lhs.build(builder);
  std::string rhs_res = this->rhs.build(builder);

  std::string res = builder.unique_id();
  builder.code << "double " << res << " = "
    << lhs_res << " + " << rhs_res << ";\n"
  return res;
}
```

Die vielen temporären Variablen, die hierbei entstehen, können später von dem *OpenCL C* oder Cuda-Übersetzer problemlos wieder eliminiert werden. Es existieren sogar Zwischenrepräsentationen, wie die von LLVM[4], welche jeden Ausdruck auf eine solche *static single assignment* Form bringen. Wie genau das Generieren von Quellcode abläuft kann unter `src-opencl/cl-grid.h` in der Struktur `GridAssignment` betrachtet werden.

3.1.2 Kernel ausführen

Um ein Expression Template auf der Grafikkarte ausführen zu können muss der erzeugte Quellcode in einen ausführbaren Kernel übersetzt werden und dieser mit den richtigen Argumenten gestartet werden. Ein Übersetzer für Kernelcode ist Teil der OpenCL-Laufzeitbibliothek. Dieser kann damit in ein Objekt der Klasse `cl::Program` übersetzt werden und in einem solchen Programm enthaltene Kernel können von der Laufzeitumgebung aufgerufen werden. Dafür müssen die Argumente angegeben werden, was auch wieder durch ein rekursives Ablaufen des Ausdruckes möglich ist. Für jeden Blattknoten in dem durch das Expression Template beschriebenen Syntaxbaum wird der darin enthaltene Wert (oder ein Zeiger auf die Werte, welcher auf der Grafikkarte gültig ist) als Argument eingetragen. Läuft man dabei in der selben Reihenfolge über den Ausdruck wie beim generieren des Kernel-Codes, so wird jedem Wert der richtige Parametername zugeordnet.

Da es bei Cuda vorgesehen ist den Code für den Haupt- und den Koprozessor gemeinsam zu übersetzen, ist ein Kompilierer kein Teil der normalen Laufzeitumge-

[4] *Low Level Virtual Machine*, eine Sammlung von Kompilerwerkzeugen

bung. Aktuelle Versionen von Cuda beinhalten allerdings eine zusätzliche *Runtime Compilation Library*, die es wie unter OpenCL ermöglicht, Kernel zur Laufzeit zu übersetzen und hinzuzuladen. Vor Cuda 7 gab es diese Möglichkeit allerdings noch nicht. Die Kernel müssen in diesem Fall mit dem, auch über die Kommandozeile verwendbaren *nvcc*-Übersetzer kompiliert werden. Der dabei entstehende *PTX*-Code kann dann über Treiberschnittstellen geladen werden. Die Bibliothek für das Ansprechen der Grafikkartentreiber, die *CUDA Driver API*, ist auch heute noch der Unterbau für die Laufzeitumgebung.

3.1.3 Durch *static initialization* Kernel nur einmal übersetzen

Der offensichtlichste Nachteil einer solchen Implementierung ist, dass vor der Ausführung eines Expression Templates erst zur Laufzeit ein Kompilierer den Kernel generieren muss. Es ist allerdings möglich diesen zusätzlichen Übersetzungsschritt auf ein einziges Mal pro Programmstart zu beschränken. Dies ist immens nützlich, da ein Expression Template beispielsweise bei iterativen Lösern sehr oft ausgewertet werden muss. Müsste man in jeder Schleifeniteration erst immer selben Quelltext generieren und diesen Übersetzen wären das sehr hohe zusätzliche Kosten.

Der Typ eines Expression Templates liegt schon bei dessen Übersetzungszeit fest. Der Kernelquelltext hängt nicht von den im Ausdruck vorkommenden Gittervariablen oder Skalaren ab, sondern nur von dessen Struktur. Der Typ alleine reicht schon um diese Struktur festzustellen. Die Programmiersprache C++ verfügt über eine Technik namens *static initialization*, mit deren Hilfe es möglich ist, statische Variablen einmalig vor Beginn der Hauptroutine eines Programms zu initialisieren. Wie der Name bereits verrät wird diese Technik meist benutzt um die Konstruktoren von globalen oder statischen Objekten auszuführen. Sie erlaubt es außerdem Klassenmethoden als `static` markierte Klassenattribute zu initialisieren. Diese Technik kann verwendet werden um allein mithilfe des Typs des Expression Templates einmalig dieses zu übersetzen und zu laden. Dann muss zur eigentlichen Auswertung des Ausdrucks nur noch der Kernel gestartet werden.

Um dies zu ermöglichen wird ein Klassentemplate eingeführt, welches im Zuweisungsoperator instanziiert wird. Als Templateparameter erhält es den Typ des Expression Templates. Dieser ist bekannt, weil er der implizite Templateparameter des Zuweisungsoperators selbst ist. Ein beliebiges Klassenattribut des Templates kann durch eine statische Methode initialisiert werden. Diese Methode kann das als Templateparameter angegebene Expression Template zu einem ausführbaren Kernel übersetzen. Damit passiert dies bereits vor der ersten Auswertung des Ausdrucks und auch nur ein einziges mal.

3.2 Implementierung für neuere Cuda-Versionen

Neue Cuda-Versionen unterstützen, insbesondere auch für den Code der auf der Grafikkarte läuft, vollständige C++-Templates. Dies ermöglicht eine neue Implementierungsstrategie für Expression Templates auf der Grafikkarte. Insbesondere interessant an Cuda ist, dass es ermöglicht, einzelne Methoden in einer Struktur, Klasse oder einem Klassentemplate mit `__device__` zu annotieren. Dies wird sich

zunutze gemacht um die Auswertung eines Expression Templates auf der Grafikkarte zu vollziehen, Konstruktoren und andere Methoden eines Ausdrucks laufen aber trotzdem auf dem Hauptprozessor.

3.2.1 Anpassungen in den wertespeichernden Strukturen

Da die Auswertung der Expression Templates nun auf der Grafikkarte erfolgen soll, müssen von den Gittervariablen auch der Speicher auf der Grafikkarte verwaltet werden. Im Konstruktor der Gittervariablen wird nicht nur der Platz für die Gitterpunkte im Hauptspeicher, sondern auch im Grafikkartenspeicher, angefordert. Die Methode zur Auswertung eines Gitterpunktes soll die Signatur `__device__ double get(int x, int y)` haben. Hier wird der zweidimensionale Fall betrachtet und jede in einem Expression Template vorkommende Struktur muss die Methode `get` besitzen. Wie man an der Signatur erkennt, kann diese Funktion nur auf der Grafikkarte ausgeführt und aufgerufen werden. Sie muss also den Zeiger auf die Daten im Grafikkartenspeicher an der entsprechenden Position dereferenzieren.

Weil ein Expression-Template-Ausdruck für die Auswertung später aus dem Hauptspeicher in den Grafikkartenspeicher kopiert werden muss, dürfen in einem solchen Ausdruck, anders als bei der klassischen Implementierung, keine Referenzen auf die Unterausdrücke mehr enthalten sein. Das liegt an dem getrennten Adressraum: Die Referenzen würden ungültig werden. Die einzige praktikable Lösung ist es, Kopien zu erzeugen und diese zu speichern. Dafür muss allerdings der Kopierkonstruktor der Gittervariablen angepasst werden. Dieser darf nicht den kompletten von ihm verwalteten Speicher, sondern nur die Zeiger und zusätzliche Metainformationen wie Größe kopieren. Im Destruktor einer Gittervariablen muss dann, ähnlich wie bei der Referenzenzählung, überprüft werden, ob keine Kopie dieser Gittervariablen mehr existiert. Nur dann kann der Speicher wieder freigegeben werden. Da sich der Kopierkonstruktor damit anders verhält als der für viele Klassen der C++-Standardbibliothek, bietet es sich an, diesen als privat zu markieren. Er muss allerdings von allen Klassentemplates, die für die einzelnen Operationen stehen, aufgerufen werden dürfen. Gezeigt wird die beschriebene Art Speicher zu verwalten unter `src-cuda/exprtmpl.h` in der Klasse `HoldingData`.

3.2.2 Anpassungen in den Klassentemplates für die Operationen

Die für Operationen wie Addition oder Multiplikation stehenden Klassentemplates können zu großen Teilen von einer klassischen Implementierung von Expression Templates übernommen werden. Wichtig ist allerdings, dass die `get`-Funktionen als auf der Grafikkarte ausführbar annotiert sind. Folglich dürfen sie auch nur dort verfügbare Operationen ausführen, alle benötigten sollten allerdings auch vorhanden sein. Mathematische Funktionen wie der Sinus, Kosinus oder der natürliche Logarithmus sind alle auch auf modernen Grafikkarten verwendbar, wobei sie allerdings häufig nicht so stark parallelisierbar sind wie gewöhnliche arithmetische Operationen und einen Kernel stark verlangsamen. Die Unterausdrücke, welche üblicherweise mit konstanten Referenzen angesprochen werden, müssen aber aus den bereits beschriebenen Gründen als vollständige Objekte im Klassentemplate enthalten sein. Der Konstruktor dieser Klassentemplates kann weiterhin konstante

Referenzen erhalten, weshalb aber auch die Kopierkonstruktoren vorhanden sein müssen. Der Übersetzer wird viele der benötigten Kopieroperationen eliminieren können, weil er das Expression Template schon vor der Laufzeit analysieren kann. Für die Typprüfung und Übersetzung ohne Optimierungen sind sie aber dennoch notwendig.

3.2.3 Auswertung und der Zuweisungsoperator

Wie auch bei einer klassischen Implementierung muss der Zuweisungsoperator der Gittervariablen das zugewiesene Expression Template auswerten. Dieses enthält allerdings keine Referenzen und die Zeiger auf die darin verwendeten Daten beziehen sich auf Grafikkartenspeicher. Es kann direkt in den Grafikkartenspeicher kopiert werden und dort, dank der mit `__device__` annotierten `get`-Methoden, an jeder Stelle im Punktegitter ausgewertet werden. Da sowieso ein Kernel gestartet werden muss, der als Argumente neben der Größe des Punktegitters auch einen Zeiger auf den Speicherbereich, in den geschrieben werden soll, erhält, bekommt dieser das Expression Template als zusätzlichen Parameter. Da dieses ein Template ist, muss auch dieser Kernel eines sein, welches als Templateparameter den Typ des Expression Templates akzeptiert. Auch wenn dies für sich eigentlich schon reichen würde, wurde in der entwickelten Implementieren stattdessen ein C++11-Lambda erzeugt, welches alle notwendigen Argumente in seinen Sichtbarkeitsbereich kopiert. Solche auf der Grafikkarte ausführbaren Lambdas sind ebenfalls Teil neuerer Cuda-Versionen. Gestartet werden muss ein solches Lambda trotzdem durch einen Template-Kernel, da jedes Lambda in C++ einen einzigartigen Typ hat. Funktionszeiger oder Klassen wie `std::function` können aufgrund der verschiedenen Adressräume nicht verwendet werden.

Da Grafikkarten für starke Parallelisierung gebaut sind, soll für jede Stelle im Punktegitter, welche neue Werte zugewiesen bekommt, auch ein eigener Thread gestartet werden, der diesen einen Wert ausrechnet. Bei der mit dieser Arbeit entwickelten Implementierung wurde die Anzahl an Threads in einem Block konstant festgelegt. Die Anzahl der Blöcke ergibt sich damit als Quotient der Threads pro Block durch die Größe des Gitters. Vor der Auswertung der Punkte im Gitter muss für die Position jedes Threads geprüft werden, ob auch ein dazugehöriger Gitterpunkt existiert, der aktualisiert werden muss. Es ist möglich, dass mehr Threads als benötigt gestartet werden, weil die Ränder des Punktegitters bei Gittervariablen unverändert bleiben sollen und die Anzahl an Threads pro Block fest ist. Es wäre unsinnig einen perfekten Teiler zu suchen, selbst wenn es einen gäbe, weil Threads auf den hier betrachteten Grafikkarten sowieso immer in Gruppen von 32 ausgeführt werden. Die Anzahl an Threads pro Block sollte deshalb auch ein vielfaches von 32 sein. Diesen Wert gut zu wählen ist schwierig, weil das Optimum unter anderem von der Grafikkartenarchitektur und der Anzahl an Speicherzugriffen abhängt.

Der Kernel, bzw. das Lambda, werden also mit der ermittelten Aufteilung gestartet und führen das Expression Template in einem Thread pro Gitterpunkt aus. Eine mögliche Implementierung des Zuweisungsoperators ist hier gezeigt:

```
template<typename Expr>
void GridVariable::operator = (const ET<Expr> &expr) {
  size_t n = rows(), m = cols();
  double *values = this->dev_values;

  auto lambda = [n, m, values, expr] __device__() -> void {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= n - 1 || y >= m - 1 || x == 0 || y == 0)
      return;

    values[x * m + y] = expr.get(x, y);
  };

  dim3 blockDim(THREADS_X, THREADS_Y, 1);
  dim3 gridDim(
    (n + THREADS_X - 1) / THREADS_X,
    (m + THREADS_Y - 1) / THREADS_Y, 1);

  run_lambda<decltype(lambda)><<<gridDim, blockDim>>>(lambda);
}
```

3.2.4 Automatische Speichersynchronisation

Einer der entscheidenden Vorteile von Expression Templates ist, dass sie einfacher zu verwenden sind als viele andere Bibliotheken. Diese Eigenschaften sollen auch für die Expression Templates für Grafikkarten erhalten bleiben. Sie sollen dem Benutzer das Synchronisieren von Speicher auf der Grafikkarte und im Hauptspeicher abnehmen oder vor ihm verstecken. Die Initialisierung oder Aktualisierung der Daten im Hauptspeicher soll automatisch in den Grafikkartenspeicher übermittelt werden. Die Ergebnisse von Berechnungen auf der Grafikkarte sollen des weiteren in den Hauptspeicher übertragen werden, wenn vom Hauptprozessor aus auf die Ergebnisse zugegriffen wird. Da die Verbindung zwischen Grafikkarte und Hauptprozessor der Flaschenhals vieler Datentransfers ist und auch viel langsamer ist, als Speichertransfer innerhalb der Grafikkarte, sollte ein unnötiger Datenaustausch vermieden werden.

Dazu speichert sich jede Gittervariable über eine Bitmaske oder Wahrheitswerte in welchem Adressraum die Daten zuletzt aktualisiert wurden. Vor jeder Auswertung eines Expression Templates auf der Grafikkarte wird überprüft, ob Änderungen im Hauptspeicher gemacht wurden. Ist dies der Fall werden die Daten blockierend aus dem Hauptspeicher in den Grafikkartenspeicher übertragen. Dafür muss jede im Ausdruck vorkommende Gittervariable überprüft werden. Dies geschieht am einfachsten durch eine weitere, rekursiv arbeitende Methode, die jede Klasse in einem Expression Template hat. Sie überprüft bei Gittervariablen wo die aktuellsten Daten liegen und kopiert diese falls notwendig. Am Ende einer Aus-

wertung wird markiert, dass die Daten auf der Grafikkarte für diese Gittervariable verändert worden sind. Die Synchronisation ist unter `src-cuda/exprtmpl.cu` in der Methode `sync` der Klasse `HoldingData` implementiert.

Soll vom Hauptprozessor aus auf Gitterpunktwerte zugegriffen werden, so darf dies nur über dafür bereitgestellte Methoden der Gittervariable passieren. Diese Methoden müssen, falls notwendig, vor dem Zugriff die Daten synchronisieren. Da dies, wie auch alle anderen Datentransfers, blockierend stattfindet, sollte es nicht möglich sein auf Daten zu stoßen, die sowohl von der Grafikkarte, als auch vom Hauptprozessor verändert worden sind. Einzig ein anderer Thread auf dem Hauptprozessor könnte dies verursacht haben.

3.3 Implementierung mittels SYCL

SYCL war von Anfang an als Erweiterung der Programmiersprache C++14 gedacht. Folglich ist es nicht verwunderlich, dass diese Technologie es ermöglicht, Expression Templates zu implementieren. Anders als Cuda setzt es nicht auf Annotationen um Grafikkarten- von Hauptprozessorcode zu unterscheiden, was es ermöglicht noch mehr Code einer ursprünglich für Hauptprozessoren geschriebenen Implementierung unverändert zu übernehmen. An zentralen Stellen müssen allerdings trotzdem wichtige Änderungen vorgenommen werden.

3.3.1 Getrenntes Klassentemplate für Auswertung auf der GPGPU

Die Speicherverwaltung wird von SYCL komplett durch C++-Klassen übernommen. Es ist nicht möglich Zeiger in den Hauptspeicher auf der Grafikkarte zu verwenden, weil SYCL-Übersetzer es nicht erlauben Variablen vom Typ Zeiger oder Referenz in auf der Grafikkarte ausgeführte Lambdas zu übernehmen. Auf der Grafikkarte liegender Speicher wird von SYCL vom Hauptprozessor aus über Objekte der Klasse `cl::sycl::buffer` angefordert und verwaltet. Wie bereits beschrieben, kann auf diese Puffer nur über Akzessoren zugegriffen werden. Wird ein Akzessor angelegt, so kopiert die SYCL-Laufzeitumgebung automatisch die Daten in den Speicher, in dem sie gebraucht werden. In das Lambda, welches auf der Grafikkarte parallel ausgeführt wird (z.B. durch `parallel_for`), können keine Puffer direkt übernommen werden, sondern nur die Akzessoren. Da für das Anlegen ein `cl::sycl::handler` notwendig ist, welchen man erst erhält, nachdem man in einer Gerätewarteschlange den Kernel zur Ausführung einreiht, kann der Akzessor nicht bereits zur Konstruktionszeit des Expression Templates angelegt werden.

Für die Implementierung der Expression Templates bedeutet dies, dass die Gittervariablen alle einen SYCL-Puffer als Attribut besitzen, welcher den Speicher für die Gitterpunkte verwaltet. Soll der Ausdruck ausgewertet werden, muss ein neues Objekt konstruiert werden, welches nicht mehr den Puffer, sondern einen Akzessor beinhaltet. Es ist also ein Zwischenschritt notwendig: Vor der Auswertung, während der Einreihung des Kernels zur Ausführung auf der Grafikkarte, muss jede in einem Expression Template auftauchende Gittervariable durch ein Objekt einer anderen Klasse ersetzt werden. Diese Vorbereitung kann rekursiv durchgeführt werden. Da sich der Typ der Blattknoten ändert, müssen auch alle anderen Knoten im Syntaxbaum ausgetauscht werden. Weil von jedem Ausdruck allerdings klar ist, in

welchen anderen Typ er umgebaut werden muss, stellt dies kein großes Hindernis da. Während dieses Umbauens ist noch etwas anderes zu beachten: Wie bereits bei der Cuda-Variante darf der Ausdruck, welcher von einem Adressraum in den anderen kopiert wird, keine Referenzen oder Zeiger mehr enthalten. Die Unterausdrücke in einem Expression Template dürfen nicht, wie bei einer klassischen Implementierung, über konstante Referenzen angesprochen werden, sondern müssen kopiert werden. Im Fall von SYCL warnt der Kompilierer den Programmierer vor solchen Fehlern. Wie dies Aussehen könnte ist hier gezeigt:

```
struct GridVariable: public ET<GridVariable> {
  cl::sycl::buffer<double, 1> data;

  struct DevET {
    cl::sycl::accessor</*...*/> accessor;

    inline double get(int x, int y) const {
      return accessor[x][y];
    }

    /*...*/
  };

  DevET prepare(cl::sycl::handler &h) {
    return DevET(data.get_access</*...*/>(h));
  }

  /*...*/
};
```

Die Methode, welche jede Operation und jede Struktur in einem Expression Template für die Auswertung umbaut, heißt hier `prepare`. Bei Operations-Klassentemplates wie denen für die Addition oder Subtraktion muss sie nur eine neue Struktur zurückgeben, die als Operanden jeweils die umgebaute Variante enthält. Die Methoden zur Auswertung werden nur für die Strukturen benötigt, welche in den Grafikkartenspeicher verschoben werden sollen.

3.3.2 Auswertung auf der Grafikkarte

Die neu erzeugte und umgebaute Kopie des Expression Templates kann von dem Lambda eingefangen werden, welches durch `parallel_for` auf der Grafikkarte ausgeführt wird. Wie schon bei Cuda kann die Anzahl der Threads anhand der Punktegitter ermittelt werden. Die Aufteilung in Blöcke kann selbst vorgenommen werden, oder auch der SYCL-Implementierung überlassen werden.

```
template<typename Expr>
void operator = (const ET<Expr> &expr) {
  size_t n = rows(), m = cols();
```

```
  deviceQueue->submit([&](sycl::handler& h) {
    auto accessor = buff
      .get_access<sycl::access::mode::write>(h);
    auto devexpr = expr.prepare(h);
    h.parallel_for<Expr>(
      cl::sycl::range<2>(n, m),
      [=](sycl::item<2> it){
        auto i = it.get_id(0);
        auto j = it.get_id(1);

        if (i == 0 || i >= n - 1) return;
        if (j == 0 || j >= m - 1) return;

        accessor[i * m + j] = devexpr.get(i, j);
    });
  });
}
```

3.3.3 Automatische Speichersynchronisation

Dank der Speicherverwaltung durch die SYCL-Klassen wie `cl::sycl::buffer` ist hier nicht viel zu beachten. Man kann dem Konstruktor solcher Puffer einen Zeiger auf Daten im Hauptspeicher übergeben, die verwendet werden sollen, um den Grafikkartenspeicher zu initialisieren. Sobald der Destruktor aufgerufen wird, schreibt SYCL die Daten aus dem Grafikkartenspeicher zurück in den Hauptspeicher. So kann, während die Gittervariable noch existiert, ein Akzessor verwendet werden, um vom Hauptprozessor aus auf die Daten zuzugreifen. Sobald die Gittervariable destruiert wird kann der ursprüngliche Zeiger auf die Daten im Hauptspeicher verwendet werden.

3.4 Funktionsweise von Reduktionen und Skalaren

Die Multiplikation der Werte eines Vektors oder einer Gittervariablen mit einem Skalar wird in vielen Fällen benötigt. Handelt es sich um einen auf dem Hauptprozessor ermittelten oder bekannten Wert (also beispielsweise ein Literal oder eine Variable vom Typ `double` oder `float`), so kann dieser Wert direkt in das Klassentemplate für diese besondere Form der Multiplikation aufgenommen werden und gemeinsam mit dem Rest des Expression Templates als Argument eines Kernels auf die Grafikkarte kopiert werden.

Es gibt aber auch Fälle, in denen ein Skalar erst auf der Grafikkarte berechnet wird. Bei Reduktionen von Vektoren oder Gittervariablen auf eine Norm passiert genau dies. Effiziente parallele Reduktionen auf Grafikkarten sind komplizierter als auf Hauptprozessoren, für die Implementierung wurde sich an einem Vortrag[2] von Nvidia dazu orientiert. Für Skalare, die das Ergebnis einer Berechnung auf dem Beschleuniger sind, muss eine neue Klasse eingeführt werden. Diese verwaltet den Speicher auf der Grafikkarte für mindestens einen Wert. Soll die Reduktion in

mehrere Kernel-Starts aufgeteilt werden, müssen Zwischenergebnisse in ebenfalls von dieser Klasse angelegten Speicher passen. Warum dies notwendig sein kann wird ebenfalls von dem Vortrag erläutert.

Für jede Reduktion wird ein Klassentemplate angelegt, welches die beteiligten Operanden speichert und ein Expression Template Ausdruck ist. Als Beispiel soll hier das Skalarprodukt dienen. Dabei wird aus zwei Vektor-Ausdrücken ein Skalar. Die Vektor-Ausdrücke können zusammengesetzte Ausdrücke sein, deren Typen wie gewohnt als Templateparameter übergeben wurden. Der Zuweisungsoperator eines Skalars wird für Reduktionen spezialisiert. Wie auch bei Gittervariablen wird durch ihn auf der Grafikkarte die gewünschte Berechnung gestartet und das Ergebnis im Grafikkartenspeicher festgehalten. Bei Zugriff über den Hauptprozessor muss das Datum zurück in den Hauptspeicher. Kommt ein solcher Skalar in einem Expression Template Ausdruck vor, wird er vom Kernel aus dem Grafikkartenspeicher geladen, weil als Teil des Ausdrucks lediglich seine Adresse bekannt ist.

Wie im Ausblick noch beschrieben wird bietet der Umgang mit Skalaren noch Optimierungspotenzial, welches noch nicht wahrgenommen wurde. Die beschriebene Implementierung hat den Nachteil, dass mit dem Ergebnis einer Reduktion nicht direkt gerechnet werden kann, sondern es erst einem Skalar zugewiesen werden muss. Man könnte aber durch eine einfache vorherige Analyse des Ausdrucks erst nach Reduktionen suchen. Der Rest des Ausdrucks könnte dann mit diesem vorher berechneten Wert arbeiten.

3.5 Hinzufügen von eigenen Funktionen

Es kann notwendig sein Funktionen auf die Punkte im Gitter anwenden zu können, damit zum Beispiel jedem Gitterpunkt ein Wert abhängig von seiner X und Y-Position gegeben werden kann. Dafür wird eine Klasse eingeführt, welche die Raumgeometrie des Gitters beschreibt. Mithilfe eines Objektes dieser Klasse können dann Koordinaten-Objekte angelegt werden, welche, wenn sie in einem Expression Template auftauchen, für die räumliche Position von jedem einzelnen Gitterpunkt stehen.

Wenn eine exakte Lösung für ein Problem bekannt ist, oder wenn die Randpunkte der Gittervariablen als Werte einer Funktion gesetzt werden sollen, werden solche selbst-definierbaren Funktionen gebraucht.

```
// 2d-Gitter mit 10000 Punkten,
// welches von (0.0, 0.0) zu (1.0, 1.0) reicht:
Grid grid(100, 100, 0., 0., 1., 1.);

/* ...Definieren von Sin und Sinh... */

X x(grid);
Y y(grid);

GridVariable A(grid);

A = Sin(X * M_PI) * Sinh(Y * M_PI);
```

Hier sollen `Sin` und `Sinh` für die beiden trigonometrischen Funktionen Sinus und Sinus hyperbolicus stehen. Diese werden direkt von Cuda als auch von OpenCL angeboten. Nach dem selben Prinzip sollen allerdings auch beliebige andere Funktionen aufgestellt werden können. Die Implementierung verläuft ähnlich wie bei Expression Templates für Hauptprozessoren: Über ein Lambda oder eine Struktur mit Aufruf-Operator wird eine vom Nutzer definierte Funktion in eine Klasse der Expression-Template-Bibliothek gepackt. Der Aufruf-Operator dieser Klasse gibt eine Struktur zurück, welche genau wie z.B. `ETAdd` Teil eines Expression Templates ist und genau wie jeder andere Unterausdruck ausgewertet werden kann. Bei der Auswertung wird die vom Nutzer festgelegte Funktion auf das Ergebnis des Unterausdruckes angewandt und zurückgegeben. Wie bei allen anderen Ausdrücken wird auch hier der Typ des Unterausdruck als Template-Parameter übergeben. In `src-cuda/grid.h` in den Strukturen `Function` und `FunctionCall` sieht man dies.

Die Besonderheit bei der Auswertung auf der Grafikkarte: Die vom Nutzer der Expression Templates festgelegte Funktion muss auf dieser ausgeführt werden können. Dazu muss sie unter Cuda annotiert werden und es muss auf die getrennten Adressräume geachtet werden. Bei SYCL gelten nur zusätzliche Einschränkungen welche der Übersetzer überprüft. Eine Implementierung für den Produktiveinsatz müsste also alle gebrauchten Funktionen selbst bereitstellen, wenn der Nutzer wirklich nichts von der Ausführung auf der Grafikkarte merken soll. Das folgende Codebeispiel zeigt, wie die Definition von `Sin` und `Sinh` unter Cuda aussehen könnte:

```
/* ... */

auto sinLambda = [] __device__ (double x) -> double {
  return sin(x);
};

ETFunction<decltype(sinLambda)> Sin(sinLambda);

// oder ohne C++11-Lambdas:

struct SinhStruct {
  __device__ double operator()(double x) {
    return sinh(x);
  }
} sinhCallable;

ETFunction<sinhStruct> Sinh(sinhCallable);

/* ... */
```

Einziger Unterschied zu einer gewöhnlichen Implementierung sind die zusätzlichen Annotationen mit `__device__`.

3.6 Auswertung für Jacobi-Verfahren

Hier soll gezeigt werden, wie die vorgestellten Expression Templates verwendet werden können, um ein Wärmeausgleichsproblem zu lösen. Dazu kann ein Jacobi-Verfahren verwendet werden. Die Aktualisierungsvorschrift für das iterative Lösen ist in Gleichung 6 gezeigt. Zunächst muss eine Gittervariable angelegt und die Randpunkte alle auf die bekannten Werte gesetzt werden. Um dies zu erreichen, können entweder vorinitialisierte Daten aus dem Haupt- in den Grafikkartenspeicher kopiert werden oder die Randpunkte werden auf der Grafikkarte zugewiesen, indem das Ergebnis einer mathematischen Formel auf allen Randpunkten ausgewertet wird.

Eine zweite Gittervariable wird gebraucht, um die Ergebnisse der Aktualisierung der Ersten zu speichern. Da dabei nur die inneren Gitterpunkte neu berechnet werden, müssen die äußeren Gitterpunkte einmalig von der anderen Gittervariable übernommen werden. Nach jeder Iteration können beide vertauscht werden damit in die Lösung sich schrittweise verbessert.

```
Grid grid(1000, 1000, 0., 0., 1., 1.);
GridVariable A(grid), B(grid);

/* Randpunkte von A und B setzen... */

for (int i = 0; i < ITERS; ++i) {
  B = (N(A) + E(A) + S(A) + W(A)) * 0.25;
  swap(A, B);
}
```

`N`, `E`, `S` und `W` stehen hier für die vier Himmelsrichtungen und lesen jeweils einen Punkt, neben dem der aktuell ausgewertet wird, aus. Die Schleifen über die Gitterpunkte werden nicht hingeschrieben, da der Zuweisungsoperator sich um die Parallelisierung dieser kümmert. `swap` vertauscht die beiden Variablen `A` und `B`. Diese Hilfsfunktion verhindert unnötiges kopieren der ganzen Daten in den Speicherbereichen, die von `A` und `B` verwaltet werden, sondern setzt nur die Zeiger auf diese um.

3.7 Auswertung für Gauß-Seidel-Verfahren

Bei Verfahren vom Gauß-Seidel-Typ werden die Ergebnisse in die selbe Gittervariable zurückgeschrieben, aus der sie in dieser Iteration auch gelesen wurden. Dies erschwert die Parallelisierung: Auf eine Speicherstelle sollte niemals unkoordiniert lesend und schreibend gleichzeitig zugegriffen werden, weswegen eine Aufteilung in Punkteklassen gefunden werden muss, bei der alle Punkte in der selben Klasse parallel verarbeitet werden können. Punkte aus verschiedenen Klassen sollen im Gegenzug hintereinander aktualisiert werden. Diese Aufteilung bei einem Gitter dieser Art nennt man auch Einfärbung. Wendet man den selben Ausdruck wie im vorher gezeigten Jacobi-Löser an (`(N(A) + E(A) + S(A) + W(A)) * 0.25`), so

kann eine Färbung in nur zwei Farben vorgenommen werden. Jeder Punkt im zweidimensionalen Gitter liest nur seine direkten Nachbarn. Es reicht also alle Punkte so einzuteilen, dass die Nachbarn alle eine andere Farbe haben. Würde man für die Aktualisierung eines Gitterpunktes auch die Punkte schräg über- und unterhalb von diesem einbeziehen, also insgesamt acht Werte, müsste man das Gitter in vier Farben aufteilen, da auch die schräg nebeneinander liegenden Punkte unterschiedliche Farben haben müssten.

Es ist möglich das Expression Template, welches der Gittervariable zugewiesen wird, auf seinen Typ (Jacobi oder Gauß-Seidel) und den Radius um den auf andere Punkte zugegriffen wird, zu analysieren. Dann kann der Zuweisungsoperator selbst entscheiden, wie er die Färbung vornimmt oder ob überhaupt eine notwendig ist. In der entwickelten Implementierung wurde darauf vorerst verzichtet, stattdessen bietet die Gittervariablen-Klasse zusätzliche Methoden an, welche das übergebene Expression Template schrittweise ausführen. Alle Punkte einer Farbe werden parallel wie bereits beschrieben ausgewertet. Für die verschiedenen Farben werden nacheinander eigene Kernel gestartet. Alle Threads in einem Kernel laufen zwar parallel, aber obwohl das starten eines Kernels asynchron passiert, werden diese hintereinander abgearbeitet. So können Wettlaufsituationen oder undefinierte Werte an einzelnen Gitterpunkten verhindert werden.

```
Grid grid(1000, 1000, 0., 0., 1., 1.);
GridVariable A(grid);

/* Randpunkte von A setzen... */

for (int i = 0; i < ITERS; ++i) {
  auto expr = (N(A) + E(A) + S(A) + W(A)) * 0.25;
  A.twoColorEvaluation(expr);
}
```

Im gezeigten Code ist zu beachten, dass der Ausdruck erst ausgewertet wird, wenn er der Methode `twoColorMode` übergeben wird. Bei einer Einfärbung in zwei Farben hat jeder Kernel nur halb so viele Threads wie es innere Gitterpunkte gibt.

3.8 Aufteilung der Threads in Blöcke

Beide gezeigten Verfahren greifen oft auf Nachbarzellen zu. Da hier auf zweidimensionalen Gittern gerechnet werden soll und sowohl SYCL als auch Cuda eine Aufteilung der Threads in zwei Dimensionen erlauben, ist man geneigt die Threads so aufzuteilen, dass ihre x und y-Positionen im Gitter an Threads auf der Grafikkarte auch direkt der Position im Punktegitter korrespondiert.

Es zeigt sich allerdings, dass diese Aufteilung nicht die höchste Leistung erreicht. Nummeriert man alle Threads in nur einer Dimension linear auf und errechnet aus dieser Postionen in dem 2D-Punktegitter erreicht man bessere Laufzeiten. Die Umrechnung könnte wie folgt aussehen: $x = i/m, y = i\%m$. Dabei sei i der Index des Threads und m die Anzahl an Punkten in der Y-Dimension des Gitters. Der Grund für dieses Verhalten erklärt sich dadurch, dass auf Grafikkarten

automatisch effizienter auf Speicher zugegriffen wird, wenn nebeneinander liegende Threads in einem Block auf nebeneinanderliegende Speicheradressen zugreifen. Dieses Phänomen nennt sich *Memory coalescing*. Da das Gitter linear Zeile für Zeile im Speicher liegt und auf beim Zugriff auf Nachbarpunkte von verschiedenen Threads auch diese wieder häufiger nebeneinanderliegen als bei einer zweidimensionalen Aufteilung der Threads, ist dieser Effekt bei den gezeigten Codes deutlich erkennbar.

3.9 Optionale Ausführung auf dem Hauptprozessor

Da SYCL-Code für die Aufteilung in Grafikkarten- und Hauptprozessorcode vorübersetzt wird, kann während dieses Schrittes auch entschieden werden, die eigentlich für Beschleuniger gedachten Kernels auf dem Hauptprozessor auszuführen. Genauso kann auch für verschiedene Plattformen und Beschleuniger ganz anderer Art, zum Beispiel spezielle FPGAs, übersetzt werden. Dies vereinfacht die Fehlersuche und ermöglicht es, ein SYCL-Programm auch auf Computern auszuführen, die keine Grafikkarte besitzen.

Auch die Cuda-Implementierung kann so geschrieben werden, dass es, falls gewünscht, möglich ist, das Programm auf dem Hauptprozessor auszuführen. Dies kann auch genutzt werden, um die Leistung der Grafikkarte für den selben Algorithmus mit der des Hauptprozessors zu vergleichen. Annotiert man sämtliche `get`-Methoden aller Strukturen in einem Expression Template sowohl mit `__device__`, als auch mit `__host__`, können diese Methoden auf beiden Prozessoren ausgewertet werden. Für jede Gittervariable und jedes Skalar wird über einen Wahrheitswert oder ein Flag in einer Bitmaske gespeichert, ob Berechnungen die diesem zugewiesen werden auf dem Beschleuniger ausgeführt werden sollen. Ist dies nicht der Fall verhält sich der Zuweisungsoperator anders: Er läuft wie bei einer normalen Implementierung einfach über alle Punkte im Gitter und wertet den übergebenen Ausdruck auf dem Hauptprozessor aus. Über einfache `OpenMP`-Pragmas kann diese Schleife über mehrere Kerne verteilt werden. Auch hier muss bei Bedarf Speicher übertragen werden, und zwar vom Beschleuniger in den Hauptspeicher. Dies ist der Fall, wenn eine der im zugewiesenen Ausdruck vorkommenden Gittervariablen zuletzt auf der Grafikkarte aktualisiert wurde.

Dieser Ansatz funktioniert nur, weil die Gittervariablen auch den Speicher auf dem Hauptprozessor verwalten. Es wäre genauso eine Implementierung denkbar, in der die Gittervariablen nur Speicher auf dem Beschleuniger anlegen und jede Berechnung dort ausführen. Getrennte Klassen könnten für Auswertung auf dem Hauptprozessor verwendet werden und ein Datentransfer würde nur stattfinden, wenn einer Hauptprozessor-Gittervariablen eine Beschleuniger-Gittervariable zugewiesen wird (und anders herum). Einen solchen Ansatz verwendet beispielsweise die von Nvidia bereitgestellte und auf Cuda aufbauende Bibliothek *Thrust*, welche, wie die hier vorgestellten Expression Templates, das Programmieren auf Grafikkarten vereinfachen soll.

4 Evaluierung

Dass die Implementierung von Expression Templates auf der Grafikkarte überhaupt möglich ist wurde gezeigt. Es wurde auch eine neue Technik vorgestellt, welche allem Anschein nach noch keine Verwendung findet. Die Verwendung eben dieser lohnt sich allerdings nur, wenn die Implementierungen auch mehr Leistung zeigen als klassische Expression Templates für Hauptprozessoren. Außerdem wäre es erfreulich, wenn die Bibliothek es schafft ähnliche Leistung wie von Hand geschriebene Kernel zu erreichen.

Um die Grafikkarten-Variante mit dem Hauptprozessor vergleichen zu können wurde eine klassische Implementierung von Expression Templates hinzugezogen, welche neben der Parallelisierung mit *OpenMP* keine weiteren Optimierungen beinhaltet.

Da zu dem als erstes präsentierten Ansatz, dem generieren des Kernel-Quelltextes zur Laufzeit, bereits wissenschaftliche Arbeiten und Auswertungen existieren, wird sich im Folgenden hauptsächlich auf die moderne Cuda- und SYCL-Variante konzentriert. Das einzige was sich vom Laufzeitverhalten her ändert ist, dass die Expression Templates schon zur Übersetzungszeit des Hauptprogrammes kompiliert werden. Da der verwendete SYCL-Übersetzer und dessen Bibliotheken, wie auch die meisten anderen, noch nicht ausgereift sind, wird die Cuda-Variante am stärksten behandelt.

4.1 Vergleich der Benutzerfreundlichkeit

Es wurde erreicht, dass der Benutzer der Expression Templates sich nicht mit Datentransfer oder Parallelisierung auf der Grafikkarte beschäftigen muss. Die Bibliothek kann fast genauso verwendet werden wie eine Implementierung für den Hauptprozessor. Zugriffe auf den verwalteten Speicher sind nur über Methoden möglich, welche sich um den gegebenenfalls notwendigen Speichertransfer kümmern. Dabei sollte der Nutzer allerdings beachten, dass er Speichertransfers nach Möglichkeit vermeiden sollte. In einer häufig ausgeführten Schleife sollte das Mischen von Zugriffen vom Haupt- und dem Koprozessor aus vermieden werden. Dies ist bei Jacobi- oder Gauß-Seidel-Lösern nicht nötig. Bei dem Verfahren der konjugierten Gradienten wird sich zeigen, dass der Transfer verkraftbar ist.

Einzig für das Definieren eigener Funktionen in der Cuda-Variante müssen Annotationen vom Benutzer der Bibliothek angebracht werden. Ansonsten muss er nichts über Grafikkartenprogrammierung wissen und kann ignorieren, dass die Ausdrücke anders als üblich ausgewertet werden.

4.2 Performancevergleich mit handgeschriebenen Kerneln

Die Abbildung 1 zeigt, wie viele Giga-Gleitkommazahloperationen pro Sekunde für verschiedene Implementierungen eines Jacobi-Verfahrens erreicht wurden. Das verwendete Gitter hatte immer 5000×5000 Punkte. Das Gitter speichert alle Einträge in doppelter Genauigkeit. Für die Aktualisierung eines Gitterpunktes wurden nur die vier unmittelbaren Nachbarpunkte verrechnet. Bei den Hauptprozessoren handelt es sich hier um zwei *Intel Xeon E5620* Prozessoren in einem Rechner. Diese

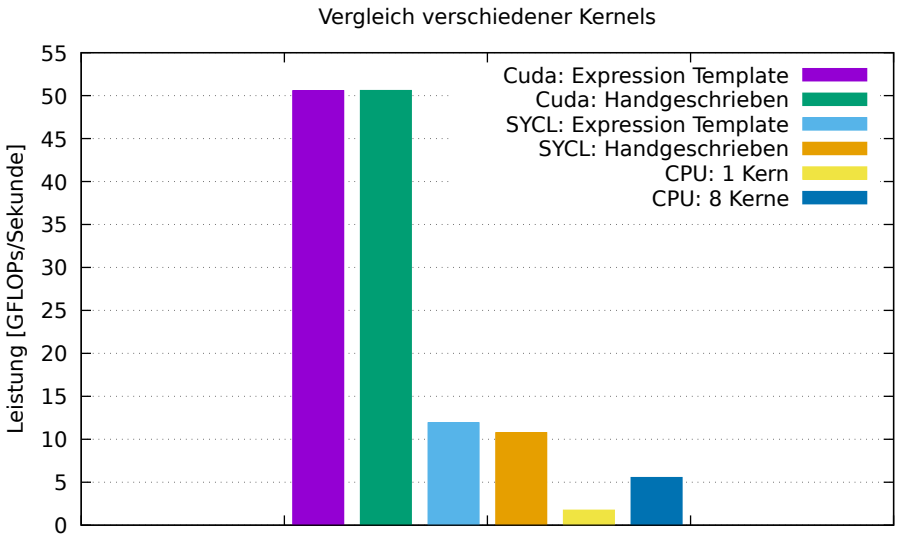


Abbildung 1: Vergleich von durch Expression Templates generierten mit normal geschriebenen Kerneln (und mit dem Hauptprozessor) für das gezeigte Jacobi-Verfahren

haben je vier physikalische Kerne, weshalb maximal für acht CPU-Kerne parallelisiert wurde. Zum Übersetzen der Vergleichsimplementierung kam der GNU C Kompiler in der Version 8.2.0 zum Einsatz. Die Grafikkarte auf der die Kernel liefen ist eine *GeForce GTX 1070*. Die Cuda-Version 9.1.85 und SYCL-Implementierung *Codeplay ComputeCpp - CE 1.1.2* wurden verwendet. Es wurde immer das höchste vom Übersetzer angebotene Optimierungslevel (Flag `-O3`) angewendet.

Bei den Messungen wurde die Zeit für den Datentransfer aus dem Hauptspeicher in den Grafikkartenspeicher nicht berücksichtigt. Da bei diesen Algorithmen allerdings nur zu Beginn und am Ende der Iterationen eine Datenübertragung vonnöten ist, kann man diese Zeit bei großen Gittern über die oft iteriert werden muss vernachlässigen. Hier wurden nur je 1000 Iterationen durchgeführt, was für eine echte Simulation natürlich viel zu wenig wäre. Die Leistung in Gleitkommazahloperationen pro Sekunde hängt allerdings nicht von der Anzahl der Iterationen ab.

Die von Hand geschriebenen Kernel für einen Expression-Template-Ausdruck der Form `B = (N(A) + E(A) + S(A) + W(A)) * 0.25` in SYCL und Cuda wurden beide einfach gehalten und entsprechen dem, was der Übersetzer nach *inlining* auch erzeugt haben sollte. `m` und `n` sind die Gitterbreite und Höhe. Das Bestimmen von i und j unterscheidet sich bei Cuda und SYCL, weshalb es hier fehlt.

```
int i = /* ... */;
int j = /* ... */;
```

```
if (i == 0 || i >= n - 1 || j == 0 || j >= m - 1)
  return;

double a1 = A[(i + 1) * m + (j     )];
double a2 = A[(i - 1) * m + (j     )];
double a3 = A[(i     ) * m + (j + 1)];
double a4 = A[(i     ) * m + (j - 1)];
B[i * m + j] = (a1 + a2 + a3 + a4) * 0.25;
```

Wie man auf Abbildung 1 sieht, haben der von Hand geschriebene Kernel und die Expression-Template-Variante unter Cuda identische Performance. Dies zeigt, dass der Kompiler das Expression Template auf fast den selben Code reduzieren konnte. Wie befürchtet erreichen beide SYCL-Implementierung nur eine deutlich schlechtere Leistung. Die Entwickler warnen allerdings auch davor, dass sich die Ausführung auf Nvidia-Grafikkarten noch in der Testphase befindet. Überraschenderweise erreicht hier die Expression-Template-Variante eine leicht bessere Performance. Alle durch die Grafikkarte beschleunigten Programme erzielen eine deutlich bessere Leistung als nur auf dem Hauptprozessor laufende Programme. Die Leistung im Vergleich zum Hauptprozessor konnte durch die Ausführung auf der Grafikkarte mehr als verneunfacht werden. Dank der Expression Templates bleibt der Code allerdings fast identisch und für den Programmierer ändert sich fast nichts.

4.3 Performancevergleich für Jacobi- und Gauß-Seidel-Verfahren mit verschiedenen Gittergrößen

Die in Abbildung 2 gezeigten Messungen wurden auf dem *emmy*-Cluster[8] des RRZE durchgeführt und die dort verfügbare Hardware verwendet. Als Übersetzer für den CPU-Code wurde diesmal der Intel C Compiler *icc* verwendet. Sämtliche Berechnungen wurden wieder mit doppelter Genauigkeit durchgeführt. In der Abbildung wird die Leistung auf beiden Prozessoren in Abhängigkeit zur Gittergröße aufgetragen. Es handelt sich um die selbe Art von Jacobi-Stern wie in Abbildung 1. Auf dem Hauptprozessor wurde auf bis zu 20 Threads parallelisiert.

Wie man sieht kann die Grafikkarte ihre volle Leistung erst bei größeren Gittern zeigen. Erst dann sind genug Recheneinheiten der Grafikkarte in Verwendung. Bei kleinen Gittern erzeugt die Parallelisierung durch OpenMP einen Mehraufwand der das Programm insgesamt verlangsamt. Gleichzeitig werden die Caches besser ausgenutzt, was bei größeren Gittern nicht mehr möglich ist, weshalb die Leistung für die CPU daraufhin wieder sinkt. Durch Cache-Blocking ließe sich dies allerdings zumindest teilweise verhindern. Grafikkarten besitzen meist keine sehr großen Caches und verstecken Speicherzugriffslatenz durch geschicktes Einlasten der Threads, weshalb der Effekt dort nicht zu sehen ist.

Der Gesamtspeicher von Grafikkarten ist allerdings auch bei modernen Karten längst nicht so hoch wie der Hauptspeicher von Hochleistungsrechnern. Für sehr große Gitter ist eine Lösung auf der Grafikkarte also im Moment auch keine Option.

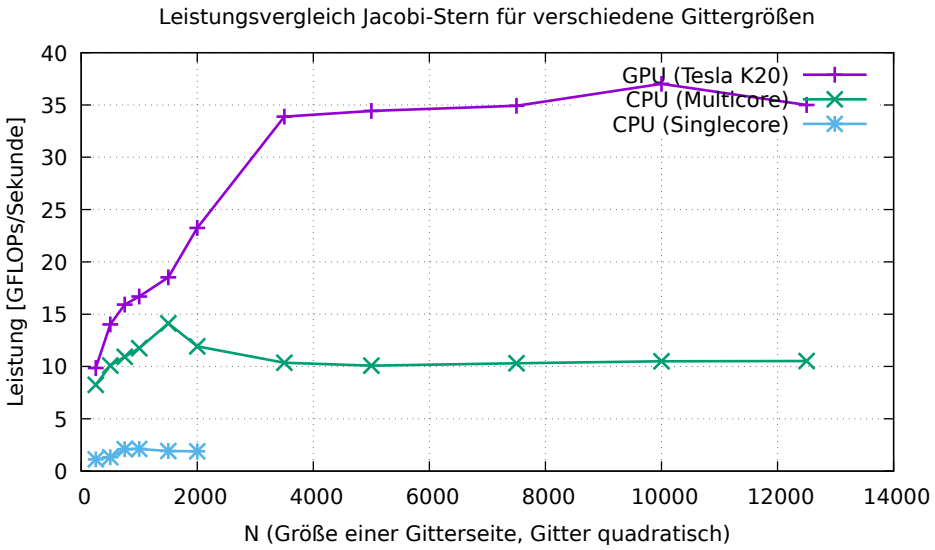


Abbildung 2: Hauptprozessoren: Zwei Intel Xeon 2660v2 mit 2,2GHz und je 10 Kernen, Grafikkarte: Nvidia Kepler K20

Möchte man keine rechteckige Fläche simulieren, sondern eine andere Form, muss diese auf ein rechteckiges Gitter gezerrt werden. Dafür werden spezielle Skalierungsfaktor für jeden Punkt in alle vier Richtungen berechnet, welche an die Gitterpunkte multipliziert werden. Sind die Skalierungswerte in den Gittervariablen $KoefN$, $KoefE$, $KoefS$ und $KoefW$ gespeichert könnte eine Aktualisierung wie folgt aussehen:

```
for (int i = 0; i < ITERS; ++i) {
  auto expr = N(A) * N(Koef1) +
              E(A) * E(Koef2) +
              S(A) * S(Koef3) +
              W(A) * W(Koef4);

  A.twoColorEvaluation(expr);
}
```

Dabei handelt es sich um ein Gauß-Seidel-Verfahren mit zwei Farben. Abbildung 3 zeigt die Performance eines solchen Codes, gemessen unter den selben Bedingungen wie bei Abbildung 2. Die Leistung ist deutlich geringer als bei einem normalen Jacobi-Verfahren weil wesentlich mehr Speichertransfer stattfinden muss. Der Vorteil von großen Caches auf dem Hauptprozessor zeigt sich viel deutlicher. Andere (neuere) Grafikkarten wie die zweite in der Abbildung haben auch größere Caches, was bei kleinen Gittern einen Unterschied zeigt. Außerdem hat diese Karte auch sonst eine höhere Maximalleistung.

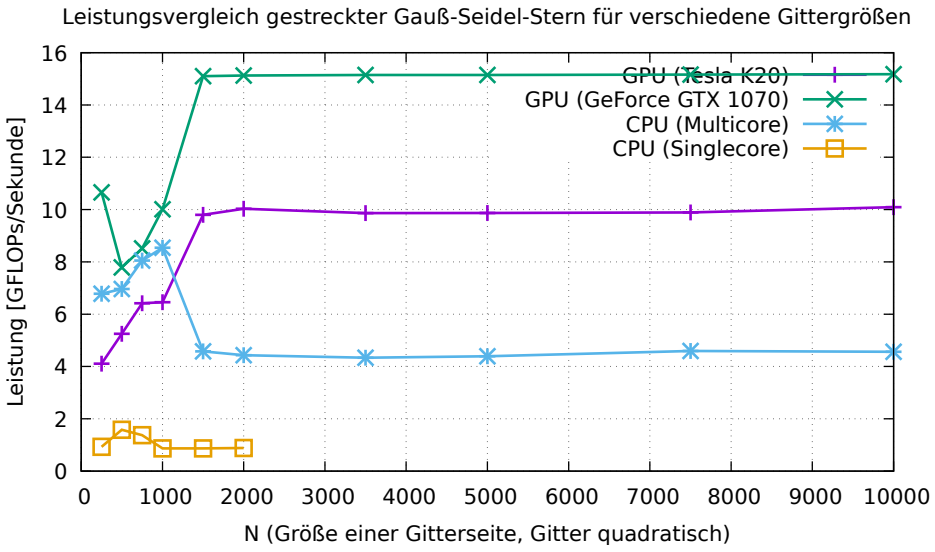


Abbildung 3: Hauptprozessoren: Zwei Intel Xeon 2660v2 mit 2,2GHz und je 10 Kernen, Grafikkarte: Nvidia Kepler K20 und Nvidia GeForce GTX 1070

4.4 Performancevergleich für das CG-Verfahren

Die getestete Implementierung für das CG-Verfahren kann in der Datei `src-cuda/cg.cu` betrachtet werden. Dieses Verfahren ist schwieriger zu parallelisieren, weil Normen gebildet werden und in einer Iteration auch auf Skalaren gerechnet wird. Des Weiteren werden pro Iteration durch den Algorithmus bedingt mehrere Kernel gestartet und Daten aus dem Grafikkartenspeicher in den Hauptspeicher kopiert. Weder CPU noch GPU erreichen eine so hohe Leistung wie bei dem Jacobi-Verfahren. Trotzdem zeigt Abbildung 4, dass die Beschleunigung durch die Grafikkarte eine höhere Leistung erreicht. Die Messung wurde wieder auf dem *emmy*-Cluster durchgeführt und auf dem Hauptprozessor mithilfe von OpenMP parallelisiert. Die selben Übersetzer mit den selben Flags wurden verwendet.

Insbesondere bei diesem Algorithmus ließe sich durch Optimierungen, die auch in die Expression Template Bibliothek eingebaut werden können, noch viel Leistung gewinnen. Im Ausblick werden diese noch kurz erläutert. Auch die CPU-Vergleichsimplementierung hätte man durch Cache-Blocking noch verbessern können.

5 Schluss

5.1 Fazit

Das Ziel, eine Expression-Template-Bibliothek zu entwickeln, welche Ausdrücke auf der Grafikkarte auswertet, wurde erreicht. Die Implementierung mit SYCL oder modernen Cuda-Versionen bietet gegenüber bereits vorher bekannten Techniken

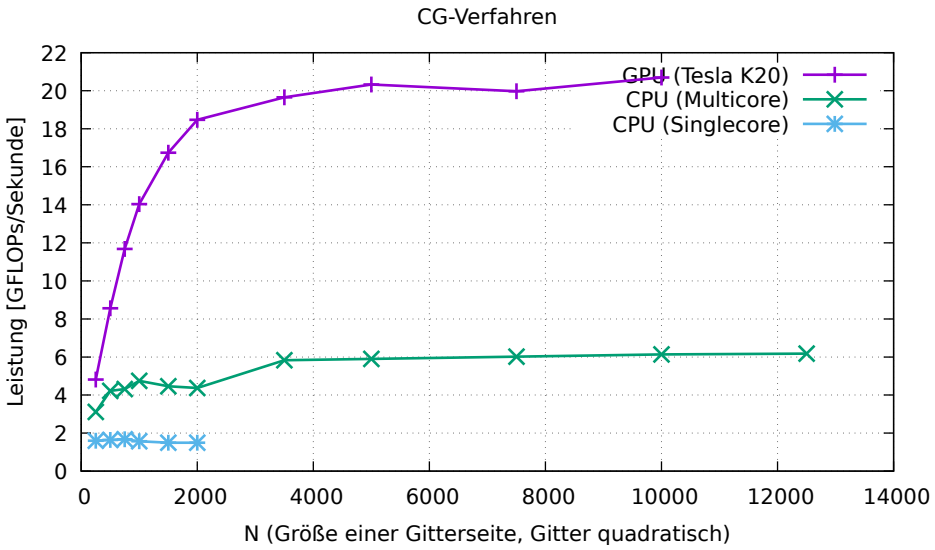


Abbildung 4: Vergleich der Performance des CG-Verfahrens auf der Grafikkarte und einem Hauptprozessor

den Vorteil, dass die Kernel nicht zur Laufzeit des Programms erst übersetzt werden müssen. Die eigentliche Auswertung von Expression Templates wird dadurch nicht schneller, allerdings kann das Programm früher mit dieser beginnen. Es entstehen des weiteren keine neuen Nachteile. Auch wenn die Leistung der Grafikkarten mit der aktuellen Implementierung noch nicht voll ausgeschöpft werden können, lässt sich die Implementierung an einigen Stellen noch so überarbeiten, dass dies möglich wird.

Mit der neuen Cuda-Implementierung konnten Kernel generiert werden, welche gegenüber von Hand geschriebenen Kerneln keine Performanceunterschiede zeigen. Im Vergleich zur Auswertung auf Hauptprozessoren konnte durch die Verwendung von Grafikkarten die Leistung deutlich gesteigert werden. Es ist gelungen die Bibliothek so zu implementieren, dass der Nutzer nicht wissen muss wie Grafikkarten programmiert werden. Für Fehlersuche, oder weil bestimmte, schlecht parallelisierbare Operationen dort besser laufen, kann optional auch auf dem Hauptprozessor ausgewertet werden.

SYCL scheint sehr vielversprechend, allerdings konnte aufgrund von Einschränkungen auf den Testrechnern und dem vergleichsweise jungen Alter dieses Standards nur ein SYCL-Übersetzer getestet werden. Die Leistung des durch diesen generierten Programms konnte nicht mit der Cuda-Version mithalten. Getestete Jacobi-Verfahren liefen zwar etwas schneller als auf dem Hauptprozessor, allerdings trotzdem enttäuschend langsam. Es werden aber viele SYCL-Übersetzer noch entwickelt und der Standard eignet sich gut um Expression Templates zu implementieren. Außerdem ist SYCL offen und baut auf OpenCL auf, mit welchem Grafikkarten

verschiedenster Hersteller und auch andere Beschleuniger programmiert werden können. Sobald SYCL an Verbreitung gewonnen hat und die Übersetzer ausgereift sind, wird es die wohl einfachste und beste Möglichkeit für Expression Templates auf Beschleunigern wie Grafikkarten sein.

5.2 Ausblick und Verbesserungsmöglichkeiten

Die Implementierung könnte an einigen Stellen noch verbessert werden: Zum einen wurden bisher nur die Operationen implementiert, welche für Jacobi-, Gauß-Seidel- und CG-Verfahren gebraucht werden. Zum anderen wurden einige noch mögliche Optimierungen nicht gemacht.

Cache-Blocking ist für Hauptprozessoren eine wichtige Maßnahme, um die Leistung von Programmen zu verbessern. Auf Grafikkarten kann statt eines Caches der von Threads in einem Block geteilte Speicher dafür verwendet werden. Bei von Hand geschriebenen Kerneln hat sich eine Verbesserung der Performance bereits gezeigt. Bei Ausdrücken, in denen auf die Nachbarn eines Gitterpunktes zugegriffen wird, lohnt sich diese Maßnahme. Die Werte aus dem Ausdruck werden aus dem Grafikkartenspeicher in den lokalen, geteilten Speicher kopiert und von dort aus ausgelesen. Dafür muss allerdings der Expression Template Ausdruck analysiert werden, um nicht nur einen Gauß-Seidel- und Jacobi-Typ zu unterscheiden, sondern auch um zu erkennen, welche Nachbarn in welchem Radius zum aktuellen Punkt ausgelesen werden. Auch bei Operationen wie der Matrix-Matrix-Multiplikation bietet sich Cache-Blocking an.

Der Start eines Kernels vom Hauptprozessor aus auf der Grafikkarte ist mit etwas zusätzlichem Verwaltungs- und Datentransferaufwand verbunden. Es kann deswegen sinnvoll sein, nicht jeden Zuweisungsoperator einen neuen Kernel starten zu lassen, sondern für Ausdrücken, die es erlauben, nur einen einzigen Kernel für getrennte Berechnungen zu starten. Dafür könnte ein Operator mit geringer Präzedenz verwendet werden, etwa der Komma- oder Logische Operatoren.

Skalare im globalen Speicher der Grafikkarte zu hinterlegen sollte, wo möglich, vermieden werden. Wenn ein Wert sowieso im Hauptspeicher aktuell ist, sollte er als Teil des Expression Templates als Argument des Kernels kopiert werden. Dadurch können im Kernel Zugriffe auf den Grafikkartenspeicher vermieden werden. Wie Reduktionen am effizientesten implementiert werden, kann architekturabhängig sein. Die passende Größe eines Threadblockes hängt wie bereits erwähnt von vielen Faktoren ab. Es existieren Funktionen wie `cudaOccupancyMaxPotentialBlockSize`, um diese abzuschätzen, sie perfekt zu wählen kann allerdings stark von den einzelnen Kernels und der Architektur abhängen.

Die Implementierung unterstützt derzeit nur eine einzige Grafikkarte. In modernen Hochleistungsrechnern ist es allerdings nicht unüblich mehrere Beschleuniger zu verbauen. Eine Erweiterung dahingehend sollte möglich sein.

Literatur

[1] Bawidamann, Uwe und Nehmeier, Marco: *Expression Templates and OpenCL*. In: *Parallel Processing and Applied Mathematics*, Seiten 71–80, 2011.

[2] Harris, Mark: *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology, `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`, Zugriff: 23.08.2019.

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

[4] Khronos OpenCL Working Group: *OpenCL Specification*. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`, Zugriff: 22.08.2019.

[5] Khronos OpenCL Working Group: *SYCL Specification*, April 2019. `https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf`, Zugriff: 22.08.2019.

[6] Nvidia: *CUDA C Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, Zugriff: 01.08.2019.

[7] Pflaum, Christoph: *Simulation und wissenschaftliches Rechnen*. Vorlesungsfolien. `https://www.cs10.tf.fau.de/files/2018/06/pflaum-slides-siwir1.pdf`, Zugriff: 10.07.2019.

[8] RRZE - Hochleistungsrechnen (HPC): *Emmy Compute-Cluster*. `https://www.anleitungen.rrze.fau.de/hpc/emmy-cluster/`, Zugriff: 07.08.2019.

[9] Schwarz, Hans Rudolf: *Methode der finiten Elemente*. Teubner Studienbücher Mathematik, 1980.

[10] Veldhuizen, Todd: *Expression Templates*. 1994 C++ World Conference in Austin, Texas. `https://pdfs.semanticscholar.org/8f41/dfd51e4da3543ae27263fc1f469a05096730.pdf`, Zugriff: 21.08.2019.

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