Effiziente parallele Implementierung eines expliziten Euler-Verfahrens für Grafikprozessoren durch Diamant-Tiling

Julien Kulbe

Bayreuth Reports on Parallel and Distributed Systems

No. 3, March 2012

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Effiziente parallele Implementierung eines expliziten Euler-Verfahrens für Grafikprozessoren durch Diamant-Tiling

Masterarbeit
Universität Bayreuth
Lehrstuhl für Angwandte Informatik 2

Betreuer: Prof. Dr. Th. Rauber,
Dr. M. Korch und Dr. C. Scholtes

Julien Kulbe

20. März 2012

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Verwendung anderer als der angegebenen Hilfsmittel verfasst habe,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäß gängigen wissenschaftlichen Zitierregeln korrekt zitiert habe.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Viele naturwissenschaftliche Problemstellungen lassen sich auf mathematische Beschreibungen durch Differentialgleichungssysteme zurückführen. Das Lösen dieser Aufgaben ist einer der Schwerpunkte der numerischen Mathematik. Da diese Systeme sehr groß sein können und das Lösen sehr lange Zeit in Anspruch nimmt, ist das Parallelisieren und Optimieren von Algorithmen zum Lösen von Differentialgleichungssystemen ein interessantes Thema. Die heutigen Grafikprozessoren bieten durch eine Vielzahl an Multiprozessoren einen wesentlich höheren Grad an Parallelität als CPUs. Aus diesem Grund stellt gerade das Parallelisieren auf modernen Grafikkartenarchitekturen eine Herausforderung dar. Grafikprozessoren bieten unterschiedliche Granularitätsstufen zum Parallelisieren. Sie bestehen meist aus mehreren Multiprozessoren und diese wiederum aus mehreren Kernen. Jede Granularitätsstufe besitzt dabei Eigenschaften und Besonderheiten, die beachtet und analysiert werden müssen, um ihr Potential vollständig auszuschöpfen.

1.2 Ziele

Das Ziel der Arbeit ist es, einen effizienten Algorithmus zum Lösen von Differentialgleichungen für Grafikkarten zu entwickeln. Dazu wird das Euler-Verfahren verwendet, da es von der Struktur sehr einfach ist und viele Möglichkeiten zum Optimieren bietet. Aber auch andere Verfahren lassen sich für die GPU portieren. Dabei muss immer abgewägt werden, welche Optimierungen für das Verfahren angewendet werden können.

Zuerst soll eine Implementierung entwickelt werden, die sich am Algorithmus des Euler-Verfahrens für CPU orientiert. Da sich in dieser Version der Algorithmus am bestehenden Code orientiert, werden hier nur einige wichtige Optimierungen für Grafikprozessoren angewendet. Aufwendige Optimierungen, wie zum Beispiel unterschiedliche Speicherhierarchien werden in dieser Version nicht betrachtet, da dafür große Änderungen am bestehenden Code notwendig sind. Diese erste Version, *lineare OpenCL-Version* genannt, dient als Grundlage für alle weiteren Versionen. Das Ziel dieser Implementierung ist es, zu zeigen, dass bestehender CPU Code mit wenigen Änderungen effizient auf Grafikkarten portiert werden kann. Dabei wird die Laufzeit der GPU-Version auf verschiedenen Hardware Architekturen ermittelt und mit der Laufzeit der bestehenden CPU-Version verglichen.

Das nächste Ziel ist es, einen Algorithmus zu entwickeln, der möglichst viele Ei-

genschaften und Besonderheiten der Grafikkarten berücksichtigt. Diese Version basiert auf dem *Diamant-Tiling* und ist wesentlich komplexer als die lineare OpenCL-Version, bietet aber größere Möglichkeiten zur Optimierung. Das Diamant-Tiling ist ein von Orozco und Gao [12] entwickeltes Verfahren zur Verbesserung der Laufzeit von Algorithmen zur Berechnung von Finiten-Differenzen-Methoden [14] mittels Datenwiederverwendung.

Das Ziel dieser Diamant-Tiling Version ist es, durch viele Optimierungen eine möglichst minimale Laufzeit auf Grafikkarten zu erzielen. Es wird einerseits untersucht, wann und ob der Einsatz von GPUs zum Lösen von Differentialgleichungen sinnvoll ist und welche Geschwindigkeitsvorteile durch die Grafikkarte erreicht werden können. Außerdem wird untersucht, welche Besonderheiten der Grafikkarten beachtet werden sollten. Dazu werden verschiedene Varianten des Diamant-Tilings entwickelt, um durch Messungen eine Aussage treffen zu können, welche Vorteile aber auch Kompromisse bei der Verwendung bestimmter Optimierungen eingegangen werden müssen.

1.3 Aufbau der Arbeit

Nach der Einführung im ersten Kapitel wird im zweiten Kapitel auf OpenCL eingegangen, um die grundlegenden Programmierkonzepte der Grafikkartenprogrammierung zu zeigen und wichtige Begriffe zu erklären. Im dritten Kapitel werden die Differentialgleichungssysteme näher erklärt und ein Lösungsverfahren, das explizite Euler-Verfahren, vorgestellt. Danach folgt ein Kapitel über die Implementierung des Euler-Verfahrens. Anschließend wird der Algorithmus von der CPU auf die GPU portiert und schrittweise analysiert und optimiert, wobei im fünften Kapitel das lineare OpenCL-Verfahren näher erklärt und diskutiert wird und im sechsten Kapitel das Diamant-Tiling vorgestellt wird. Implementierungen des Diamant-Tilings werden dann im siebten bis neunten Kapitel vorgestellt. In den abschließenden Kapiteln 10 und 11 werden die Erkenntnisse evaluiert.

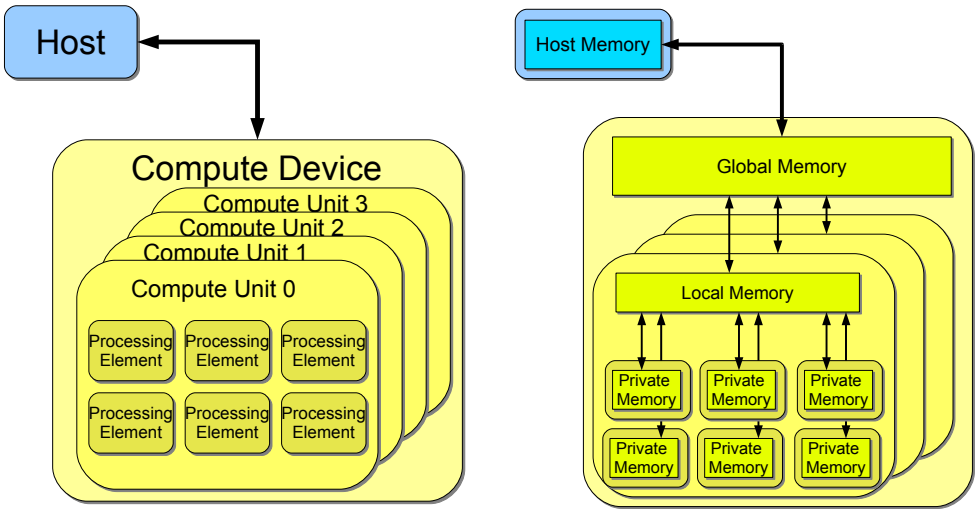
2 OpenCL

OpenCL (**Open C**ompute **L**anguage) [6] ist eine Programmierschnittstelle zur parallelen Programmierung von CPUs und GPUs. Sie wird von der Khronos Group entwickelt, der AMD, Apple, Electronic Arts, IBM, Intel, NVIDIA und andere Firmen angehören. Die ursprüngliche Version 1.0 von OpenCL wurde am 8. Dezember 2008 veröffentlicht. Als Grundlage für die Implementierungen dieser Masterarbeit diente die Version 1.1 von OpenCL, die am 15. Juni 2010 veröffentlicht wurde. OpenCL ist ein Standard zur daten- und taskparallelen Programmierung. OpenCL-Programme bestehen aus sogenannten Kernels, die während der Laufzeit auf verschiedene Geräte verteilt werden können. Die Kernels werden in der Programmiersprache OpenCL-C geschrieben. OpenCL-C ist eine an C (ISO C99) angelehnte Programmiersprache, die um weitere Datentypen und Schlüsselwörter erweitert wurde. Neue Datentypen sind die Vektordatentypen, zum Beispiel repräsentiert float4 einen Vektor aus 4 Gleitkommawerten. OpenCL-C besitzt auch Einschränkungen gegenüber C99. Beispielsweise ist die Pointerarithmetik sehr eingeschränkt und es dürfen keine rekursiven Methoden definiert werden.

Ein Kernel ist eine Funktion eines Programms und deren aufgerufene Unterfunktionen, die auf einem OpenCL-Gerät ausgeführt werden. Ein OpenCL-System besteht aus einem Host und mehreren OpenCL-Geräten (Devices). Der Host besitzt die Kontrolle über das Programm und kann zur Laufzeit bestimmen, welcher Kernel von welchem Device ausgeführt werden soll. Dazu ist es notwendig, dass erst während der Laufzeit der OpenCL-Compiler des jeweiligen Devices den Kernel übersetzt. Dies hat den Vorteil, dass OpenCL-Kernel flexibel und portabel sind, da sie unabhängig von der später verwendeten Hardware programmiert werden können. Im Folgenden ist der Host immer die CPU und das Device die GPU. Jedes Device besteht aus einer oder mehreren Recheneinheiten (*Compute Units*). Bei CPUs sind dies die einzelnen Kerne, bei GPUs die Multicoreprozessoren. Jede Recheneinheit der GPU besteht dann wieder aus mehreren Kernen (*Processing Elements*), die die Berechnungen durchführen. Die einzelnen Threads der GPU werden jeweils als *Workitem* bezeichnet. Jedes Workitem wird dabei einem Kern zugeordnet und bekommt einen Index, anhand dessen es eindeutig identifiziert werden kann. Die Gesamtzahl der Workitems sind die *globalen Workitems*. Alle Workitems, die einer Recheneinheit angehören, werden als *Workgroup* bezeichnet und sind die *lokalen Workitems*. (Anm.: Im späteren Verlauf der Arbeit wird die Anzahl der Recheneinheiten oft der Anzahl der Workgroups entsprechen, das heißt, jede Workgroup ist genau einer Recheneinheit zugeordnet. Daher werden die Begriffe der Recheneinheit und der Workgroup oft synonym verwendet)

Die Unterteilung in globale Workitems und lokale Workitems ist insbesondere

deshalb wichtig, da nur Workitems einer Workgroup mittels Barriers untereinander synchronisiert werden können. Alle Workitems eines Devices teilen sich den globalen Speicher (*global memory*). Der globale Speicher ist der größte Speicherbereich der Grafikkarte (bei heutigen Grafikkarten oft über 512 MB). Auf diesen Speicher kann der Host zugreifen, um Daten zu schreiben und/oder zu lesen. Der globale Speicher ist aber auch der langsamste Speicher. Daher sollte versucht werden in den Kernels die Anzahl der Lese- und Schreibzugriffe darauf zu minimieren. Der nächst schnellere Speicher ist der lokale Speicher (*local memory*) der zwischen 16 kB und 64 kB groß ist. Der lokale Speicher existiert für jede Recheneinheit und ist nur von den Workitems der Workgroup nutzbar, die auf der Recheneinheit gerade ausgeführt wird. Jedes Workitem besitzt noch einen privaten Speicher (*private memory*), der nicht für andere Workitems nutzbar ist, sondern zum Speichern für private Hilfsvariablen dient. Der OpenCL-Standard definiert dabei nicht, wie groß und wo der private Speicher ist. Abbildung 2.1 zeigt die Architektur und die Speicherhierarchie eines OpenCL-Systems mit einem Device.



(a) OpenCL Architektur

(b) OpenCL Speicherhierarchie

Abbildung 2.1: Architektur und Speichermodell eines OpenCL-Systems

2.1 Aufbau eines OpenCL Programms

Im Folgenden wird der prinzipielle Aufbau eines jeden OpenCL Hostprogramms erläutert, um die zentralen Begriffe von OpenCL einzuführen.

Im Host Code eines OpenCL-Programms muss zuerst ein OpenCL-System/eine OpenCL-Plattform (=*platform*) ausgewählt werden. Auf diesem OpenCL-System können dann ein oder mehrere Devices registriert werden. Anhand der Plattform

kann auf die einzelnen Devices zugegriffen werden. Den Devices wird dann jeweils ein Kontext (*Context*) zugeordnet. Ein Kontext dient zur Kommunikation zwischen verschiedenen Devices und zwischen Devices und dem Host. Danach wird eine Befehlswarteschlange (*Command Queue*) erstellt, anhand derer die verschiedenen Kernels gestartet werden können.

```
cl_uint numPlatforms;
cl_platform_id *platforms;
cl_platform_id platform;
cl_device_id device;
cl_context context;
cl_command_queue queue;

// Erhalte alle verfügbaren Plattformen
clGetPlatformIDs(0, NULL, &numPlatforms);
platforms = malloc(numPlatforms * sizeof(cl_platform_id);
clGetPlatformIDs(numPlatforms, platforms, NULL);
platform = platforms[i];

//Erstelle Device, Kontext und Befehlswarteschlange
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device,
 NULL);
clCreateContext(0, 1, &device, NULL, NULL, &err);
clCreateCommandQueue(context, device, 0, &err);
```

Nachdem das OpenCL-System initialisiert wurde, kann aus einer vorhandenen OpenCL-C Datei ein Kernel kompiliert werden. Der komplette Kernel muss zuvor in eine String-Variable eingelesen werden. Aus diesem String kann dann das Programmobjekt erzeugt werden, welches zunächst den Kernelquelltext enthält. Im Anschluss kann das Programmobjekt übersetzt werden. Aus diesem übersetzten Programmobjekt kann ein Kernel erzeugt werden. Jeder Kernel braucht dabei einen Funktionsnamen, der die Funktion angibt, welche dem Kernel als Einstiegspunkt dienen soll.

```
cl_program program;
cl_kernel kernel;

// Lade den Kernelquelltext aus source in ein
 Programmobjekt
program = clCreateProgramWithSource(context, 1, &source,
 NULL, &err);
// Kompiliere Programmobjekt
clBuildProgram(program, 0, NULL, (const char *)options,
 NULL, NULL);
// Erzeuge Kernel mit der Kernelmethode solve
kernel = clCreateKernel(program, "solve", &err);
```

Anhand des Kontextes können Daten vom Host-Speicher (im nachfolgenden Beispiel der hostpointer) zum globalen Speicher des Devices übertragen werden (im

Beispiel arg). Diese *Memory Buffer* können dem Kernel als Argumente zur Berechnung mitgegeben werden. Nachdem alle Argumente der Kernelmethode gesetzt wurden, kann der Kernel mit einer bestimmten Anzahl an globalen und lokalen Workitems gestartet werden. Nach der Beendigung der Kernels können die Daten wieder vom Host aus dem globalen Speicher ausgelesen werden.

```
// Erzeuge einen Memory Buffer
// und kopiere die Daten aufs Device
cl_mem arg = clCreateBuffer(context, CL_MEM_READ_WRITE |
 CL_MEM_USE_HOST_PTR, sizeof(float) * 1000, hostpointer, &
  err);
// Setzen der Kernelargumente
clSetKernelArg(kernel, 0, sizeof(arg), &arg);

// Starte OpenCL Kernel mit der Anzahl
// von globalen und lokalen Workitems
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global,
 local, 0, NULL, NULL);
// Warte auf die Beendigung des Kernels
clFinish(queue);

// Auslesen des Memory Buffer
clEnqueueReadBuffer(queue, arg, CL_TRUE, 0, sizeof(float)
 * 1000, hostpointer, 0, 0, 0);
```

2.2 Optimierung eines OpenCL Programms

Um das Verhalten von OpenCL Kernels für die GPU zu verbessern, müssen die Besonderheiten der Grafikkarte berücksichtigt werden. Als Grundlage der Optimierungen dienen die Erkenntnisse aus Game Programming Gems 8 [8, S. 575ff.]. Es wurden dabei verschiedene Optimierungen in Bezug auf OpenCL auf GPUs untersucht. Diese Optimierungen dienen als Ausgangspunkt für die Optimierung des Algorithmus und werden im Verlauf dieser Arbeit evaluiert.

Gemäß [8] gibt es mehrere Möglichkeiten folgende Schleife zu optimieren. Der Kernel ist eine Filterfunktion und die Schleife akkumuliert dabei für eine gegebene Filterweite die Daten aus dem globalen Speicher auf.

```
float sum = 0;
for (int c = 0; c < filterWidth; c++)
  sum += filter[idxFtmp + c] * input[idxIntmp + c];
}
```

2.2.1 Loop Unrolling

Damit die Anzahl der Schleifendurchläufe reduziert werden kann, ist es von Vorteil, Schleifen mit Loop Unrolling aufzuspalten. Dabei werden mehrere Schleifendurch-

läufe zu einer Iteration zusammengefasst. Haben die Daten keine Abhängigkeiten zueinander, lassen sich die Befehle parallel abarbeiten. Loop Unrolling bietet die Grundlage für die spätere Vektorisierung der Schleifen.

```
float sum = 0;
int c = 0;
while (c <= filterWidth - 4) {
  sum += filter[idxFtmp + c] * input[idxIntmp + c];
  sum += filter[idxFtmp + c + 1] * input[idxIntmp + c +
   1];
  sum += filter[idxFtmp + c + 2] * input[idxIntmp + c +
   2];
  sum += filter[idxFtmp + c + 3] * input[idxIntmp + c +
   3];
  c += 4;
}
for (; c < filterWidth; c++) {
  sum += filter[idxFtmp + c] * input[idxIntmp + c];
}
```

2.2.2 Vektorisierung

OpenCL C verfügt über Vektordatentypen. Meist werden vierwertige Vektoren benutzt, da die Grafikkarten auf vier float-Werte optimiert sind (Anm: Grafikkarten benutzen Vektoren mit vier float-Werten für Farben im RGBA-Raum und Koordinatenbrechungen mit Quaternionen und sind daher speziell für diese Vierervektoren optimiert). In der gegebenen Schleife kann also die Summe auch als float4 deklariert werden, um danach Vektoroperationen für die Berechnung zu benutzen. Der Befehl vload4 lädt dabei vier float-Werte aus einem Array in ein float4-Feld.

```
float4 sum4 = 0;
int c = 0; int c4 = 0;
while (c <= filterWidth - 4) {
  float4 filter4 = vload4(c4, filter+idxFtmp);
  float4 in4   = vload4(c4, input + idxIntmp);
  sum4 += in4 * filter4;
  c += 4; c4++;
}
for (;c < filterWidth; c++) {
  sum4.x =filter[idxFtmp + c] * input[idxIntmp + c];
}
```

2.2.3 Anzahl der Workitems und Workgroups

Die Anzahl der Workitems und Workgroups mit denen ein Kernel gestartet wird, ist ein weiterer wichtiger Punkt zur Optimierung. Um alle Recheneinheiten der Grafikkarte zu beschäftigen, sollte die Anzahl der Workgroups mindestens so groß

sein, wie die Anzahl der Recheneinheiten. Ist die Anzahl der Workgroups jedoch größer als die der Recheneinheiten, muss auf eine gleichmäßige Verteilung der Arbeit auf die unterschiedlichen Recheneinheiten geachtet werden. Es sollte also die Anzahl der Recheneinheiten der Grafikkarte abgefragt werden, um dann die Anzahl der Workgroups auf die Anzahl der Recheneinheiten zu setzen, beziehungsweise auf ein ganzzahliges Vielfaches davon, soweit dies möglich ist.

Die Anzahl der Workitems einer Workgroup sollte mindestens so groß sein, wie von einer Recheneinheit parallel bearbeiten werden kann. Auf AMD GPUs wird diese Größe als *wavefront* und bei NVIDIA als *warpsize* bezeichnet. Auf heutigen Grafikkarten beträgt sie 32 oder 64 Workitems. Jede Recheneinheit besitzt eine begrenzte Anzahl an Registern, die auf die einzelnen Workitems verteilt wird. Je größer die Anzahl der Register pro Workitem ist, desto geringer ist die maximal mögliche Anzahl an Workitems pro Workgroup. Die maximale Anzahl der Woritems pro Workgroup kann durch clGetKernelWorkGroupInfo abgefragt werden.

2.2.4 Lese- und Schreibzugriffe auf den globalen Speicher

Zum Austausch von Daten zwischen Host und Device kann der Host auf den globalen Speicher des Devices zugreifen. Der globale Speicher ist allerdings der langsamste Speicher des Devices und aus diesem Grund sollte versucht werden die Lese- und Schreibzugriffe auf den globalen Speicher zu minimieren.

Beim Zugriff auf den globalen Speicher sollte darauf geachtet werden, welches Zugriffsmuster am besten vom Device unterstützt wird. Die Grafikkarte ist optimiert für 128-Bit Werte (= vier float-Werte). Es ist also ratsam, jeweils 128 Bit am Stück zu lesen oder zu schreiben. Des Weiteren sollte angestrebt werden, dass alle Workitems gleichzeitig benachbarte Blöcke der Größe 128-Bit laden. Das ideale Zugriffsmuster (nach [8, S. 583]) liegt vor, wenn jedes Workitem bei seinem Workitemindex beginnt float4 (oder int4) Werte zu lesen. Dieser Anfangswert wird in den folgenden Iterationen um die Gesamtzahl der Workitems erhöht bis alle Daten aus dem Globalen Speicher gelesen wurden. Dies führt dazu, dass alle Workitems gleichzeitig einen großen Block von jeweils 128-bit Werten lesen (siehe Abbildung 2.2).

```
uint id = globale Workitem Id
uint ids = Anzahl globaler Workitems
for (idx = id; idx < filterWidth; idx += ids) {
...
}
```

2.2.5 Lokaler Speicher

Um die Anzahl der Lese- und Schreibzugriffe des globalen Speichers zu optimieren, ist es sinnvoll, häufig benutzte Daten zuerst in den lokalen Speicher zu laden und erst am Ende der Berechnung wieder in den globalen Speicher zu speichern. Jede Recheneinheit verfügt über einen lokalen Speicher. Dieser ist typischerweise

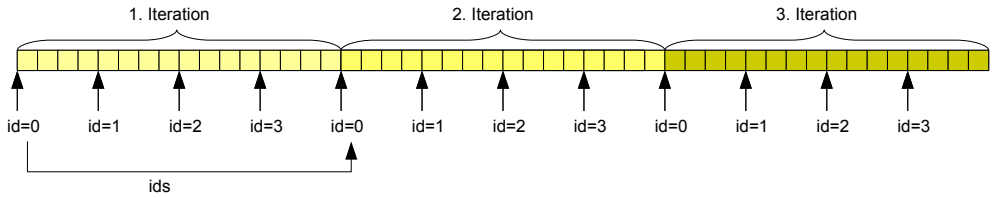


Abbildung 2.2: Zugriffsmuster des globalen Speichers

wesentlich schneller als der globale Speicher. Es können jedoch nur die Workitems einer Workgroup auf den lokalen Speicher der Recheneinheit zugreifen, auf der sie abgearbeitet werden. Das stellt für Daten, die von mehreren Workgroups gebraucht werden, ein Problem dar. Um Daten zwischen verschiedenen lokalen Speichern auszutauschen, müssen sie über den globalen Speicher ausgetauscht werden. Dazu ist aber eine Synchronisierung der Workitems notwendig.

2.2.6 Barriers

Barriers dienen zur Synchronisierung von Workitems. OpenCL bietet Barriers aber nur pro Workgroup an. Eine globale Barrier für alle Workitems ist durch die Verteilung der Workgroups auf verschiedene Recheneinheiten nicht praktikabel. Eine globale Barrier lässt sich realisieren, indem ein Kernel beendet wird und sich somit auch alle Workitems beenden. Der Host kann nun einen neuen Kernel starten und synchronisiert somit alle Workitems. Da dies relativ lange dauert, sollte versucht werden, die Anzahl der globalen Barriers zu minimieren und Sie durch lokale Barriers zu ersetzen.

3 ODE-Systeme und Lösungsverfahren

3.1 Definition von ODE-Systemen

Viele naturwissenschaftliche Probleme lassen sich auf gewöhnliche Differentialgleichungen (*ODE*, ordinary differential equation) mit gegebenem Anfangswert (IV, initial value) zurückführen. Solche Probleme werden als IVPs (initial value problems) bezeichnet. Ein IVP besteht zum einen aus Gleichungen, die das zeitliche Verhalten des Systems beschreiben (Gleichung 3.1) und zum anderen aus dem Anfangswert, der das System zum Zeitpunkt t_0 beschreibt (Gleichung 3.2). Die Lösung des Problems besteht in der Bestimmung des Zustands des Systems zu einem bestimmten Zeitpunkt $t_e > t_0$ (ausführlichere Informationen finden sich in [3] und [13]).

Abbildung 3.1 zeigt ein exemplarisches ODE-Problem. Die durchgezogene Kurve zeigt den physikalisch exakten Verlauf, der jedoch bis auf den Anfangswert bei y_0 unbekannt ist. Die gestrichelte Kurve zeigt den berechneten Verlauf eines Lösungsverfahrens für das gegebene ODE-Problem mit den berechneten Punkten y_i zu den Zeitpunkten t_i.

$$y^{'}(t) = f(t, y(t)) \tag{3.1}$$

$$y(t_0) = y_0 \tag{3.2}$$

Der Ausgangswert wird als n-stelliger Vektor modelliert. Aus diesem Ausgangswert werden iterativ mit einer gegebenen Schrittweite h neue Vektoren mit Werten berechnet, die den Zustand des System zum Zeitpunkt t_i repräsentieren. Das *explizite Euler-Verfahren* ist eine Methode zur Lösung dieser Systeme. Dieses Verfahren ist eine einfaches Verfahren, um IVPs zu lösen. Es wird dafür eine fest vorgegebene Schrittweite $h > 0$ gewählt. Mit dieser Schrittweite werden dann die Zustände des Systems zu den Zeitpunkten

$$t_i = t_0 + i \cdot h, \qquad i \in \mathbb{N} \tag{3.3}$$

berechnet.

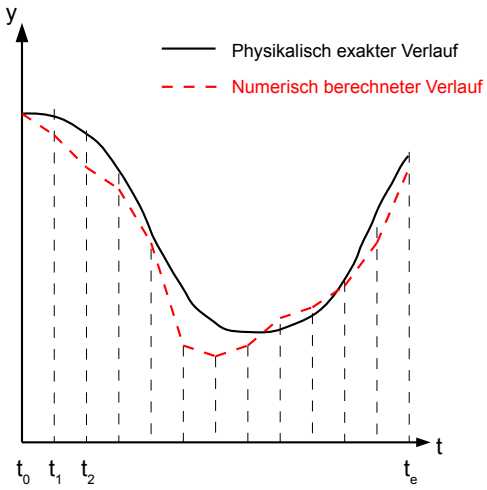


Abbildung 3.1: Berechnetes und exaktes Verhalten eines IVPs

3.2 Explizites Euler-Verfahren

Listing 3.2 zeigt den Pseudocode des expliziten Euler-Verfahrens. H beschreibt dabei die Zeitdifferenz zwischen t_0 und t_e. In jedem Zeitschritt werden aus dem n-stelligen Vektor y_cur für t die neuen Werte y_new für den Zeitpunkt $t+h$ berechnet. Die nächste Iteration verwendet dann y_new als Ausgangswerte für den Zeitschritt $t+h$. In der Implementierung wird dies dadurch gelöst, dass y_cur und y_new vertauscht werden. Die Funktion ode_eval_comp beschreibt das spezifische Problem, welches gelöst werden soll. Sie berechnet für eine gegebene Komponente j des Vektors aus den aktuellen Werten die Steigung der Werte zum Zeitpunkt t. Diese Änderung, wird mit der Schrittweite h gewichtet, auf den aktuellen Wert addiert:

$$y_{j,t+h} = y_{j,t} + h \cdot ode\_eval\_comp(j, t, y_t) \tag{3.4}$$

Daraus ergibt sich der neue Wert der Komponente j.

Die Methode berechnet genau den Eintrag mit Index j der rechten Seite des ODE-Systems aus Gleichung 3.1. Diese neu berechneten Werte besitzen keine gegenseitigen Abhängigkeiten und können daher auch parallel zueinander berechnet werden. Die Berechnung der Lösung für ein spezifisches Problem kann einen hohen Bedarf an Rechenaufwand darstellen, da zum einen die Anzahl der Komponenten sehr groß sein kann und zum anderen die Auswertung der Methode ode_eval_comp sehr komplex sein kann. Aufgrund des hohen Rechenaufwands und des hohen Grades an Parallelität bietet es sich an, ODEs auf GPUs zu berechnen.

Abbildung 3.3 zeigt das prinzipielle Vorgehen dafür, wie aus den n Ausgangswerten iterativ die nächsten n Werte bis zum Endzeitpunkt t_e berechnet werden. Die schattierten Werte stellen dabei Einträge des Vektors dar, welche schon be-

```
y_cur = Ausgangswerte_des_ODE_Systems;
for (i = 0; i < H / h; i++) {
  t = t_0 + i * h;
  for (j = 0; j < n; j++) {
    y_new[j] = y_cur[j] + h * ode_eval_comp(j, t, y_cur);
    swap(y_new, y_cur);
  }
}
print y_cur
```

Abbildung 3.2: Pseudocode explizites Eulerverfahren

rechnet wurden. Die weißen Einträge kennzeichnen dagegen noch nicht berechnete Werte. Zur Berechnung jedes Elements $y\_new[j]$, j je 1,...,n für den Zeitpunkt t_{i+1} werden bis zu $2 \cdot k + 1$ Ausgangswerte vom Zeitpunkt t_i benötigt. Im allgemeinen Fall könnten alle Werte vom Zeitschritt t_i gebraucht werden zur Berechnung von $y\_new[j]$. Bei vielen naturwissenschaftlichen Problemen lässt sich aber ein k bestimmen welches wesentlich kleiner als die Systemdimension n ist. Die Größe k, die das Speicherzugriffsmuster und die Lokalität der Funktion f charakterisiert, ist die Zugriffsdistanz. Die Zugriffsdistanz gibt an, wie groß der Abstand vom zu berechnenden Wert mit Index j zum Index $j + k$ des benötigten Wertes maximal ist.

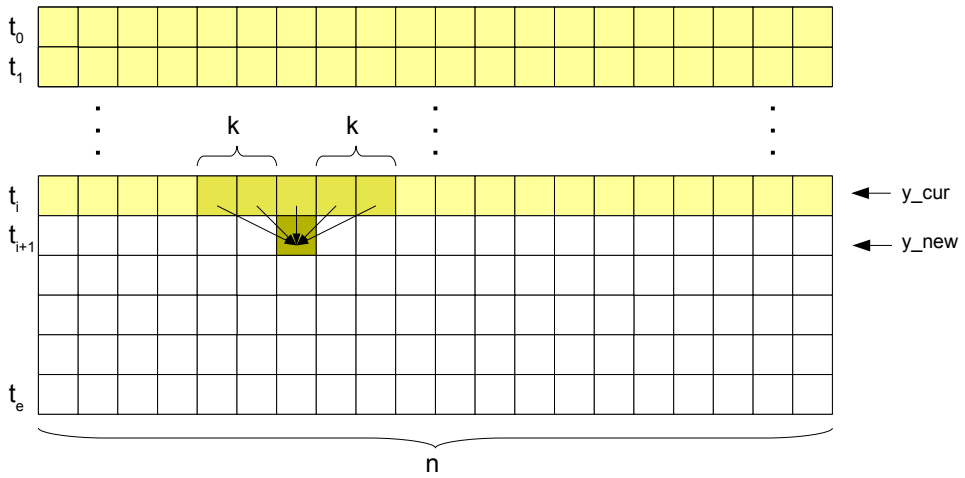


Abbildung 3.3: prinzipielles Vorgehen zur Berechnung

3.3 Ausgewählte Problemstellungen

Als Beispiele für simulierende Systeme wurden der Brüsselator (Bruss2d-Problem) und ein Modell für das Schwingen einer Saite (String-Problem) verwendet. Der Brüsselator ist ein Modell zur Beschreibung chemischer Oszillatoren.

In Abbildung 3.4 ist das Brüsselator-Problem dargestellt. Das System besteht aus zwei verschiedenen Stoffen mit den Konzentrationen A und B in einem zweidimensionalen Gitter. Das zweidimensionale Gitter besteht aus $m*m$ Gitterpunkten. Der n-stellige Vektor des resultierenden Systems repräsentiert die Konzentrationen der beiden Stoffe in den m^2 Gitterpunkten. Jedes Kästchen von Stoff A und B steht dabei für eine Konzentration, die in dem n-stelligen Vektor abgespeichert wird. Um die Werte des zweidimensionalen Gitters des Brüsselators im Speicher abzubilden, werden die Werte zeilenweise jeweils abwechselnd für die Konzentration für Stoff A und B im Speicher abgelegt. Zur Berechnung der neuen Konzentration von Stoff A mit Punkt P = 22, werden die Werte der Konzentration von A von allen angrenzenden Gitterpunkten von P und von P selbst, sowie die Konzentration von Stoff B im Punkt P benötigt. Beim Brüsselator mit einem $m * m$ Gitter dessen Werte zeilenweise abgelegt sind, entspricht die Zugriffsdistanz einer vollständigen Zeile des Gitters. Sie beträgt 2m, da jeweils die Werte von Stoff A und B pro Zeile abgelegt sind. Die Zugriffsdistanz wächst demnach mit der Systemgröße.

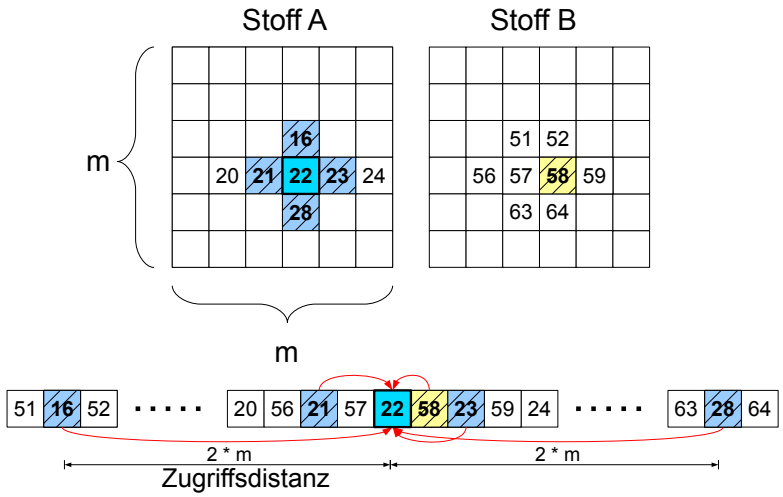


Abbildung 3.4: Abbildung des Brüsselators im Speicher

Das String-Problem ([3, S. 27]) beschreibt das System einer schwingenden Saite, die an beiden Enden fest eingespannt ist. Die Saite wird durch äquidistante Abstände in Massepunkte unterteilt. Der n-stellige Vektor repräsentiert die Auslenkungen sowie die Geschwindigkeiten in den verschiedenen Massepunkten. Um die neue

Auslenkung von Punkt P zu berechnen, wird die Geschwindigkeit von P selbst, der vorhergehenden Punkt P-1 und der nachfolgenden Punkt P+1 gebraucht. Die Zugriffsdistanz beträgt dabei 3. Die Zugriffsdistanz ist also hier unabhängig von der Größe des Systems (siehe Abbildung 5.7).

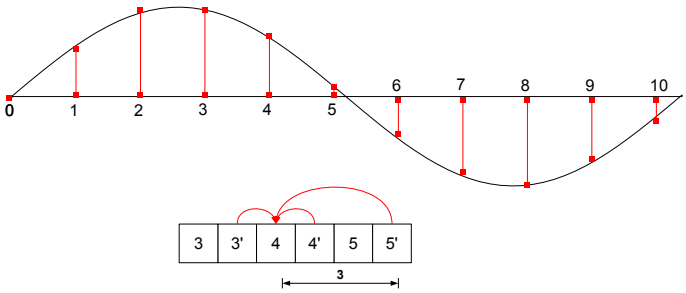


Abbildung 3.5: String Problem

4 CPU Implementierung des expliziten Euler-Verfahrens

4.1 Sequentielle Version

Als Ausgangspunkt der Optimierung steht eine Implementierung des expliziten Euler-Verfahrens von Dr. M. Korch [7] zur Verfügung. Listing 4.1 zeigt den zentralen Teil der Berechnung. Die Parameter der Funktion sind: die Startzeit zu der die Berechnung beginnen soll ($t0$), die Anfangswerte ($y0$) zum Startzeitpunkt, die Zeitspanne ($H = t_e - t_0$) über die berechnet werden soll, ein Zeiger auf das Feld, in dem der Lösungsvektor (y) eingetragen werden soll und die Schrittweite (h). Die Systemgröße n wird durch das zu lösende ODE-Problem vorgegeben (ode_size). Die Felder y0 und y besitzen jeweils n Einträge für die n Anfangswerte beziehungsweise Endwerte des Systems. Im Algorithmus wird zuerst ein Feld Y allokiert, in dem die Zwischenwerte der Berechnung gespeichert werden. Dieses Feld ist $2 * n$ groß, da es genügend Speicherplatz für die aktuellen Werte, sowie für die neu berechneten Werte des Systems bieten muss. Die Anzahl der Berechnungsschritte ($steps$) wird durch

$$\text{steps} = \frac{\text{Zeitspanne über die berechnet wird } (H)}{\text{Schrittweite } (h)} \tag{4.1}$$

errechnet. Danach werden die Ausgangswerte von y0 in das Feld Y kopiert, um die Berechnung zu initialisieren. In den einzelnen Berechnungsschritten wird jeweils das aktuelle (Y_cur) und das neue Feld (Y_new) alternierend bestimmt. Anschließend werden für jeden einzelnen Wert des ODE-Systems die neuen Werte aus den aktuellen Werten gemäß der Formel 3.4 berechnet.

4.2 Pthreads parallelisierte Version

Es werden in der Pthreads Version zwei Datenstrukturen verwendet, die über Argumente verfügbar sind. Einmal eine Datenstruktur die sich alle Threads teilen und eine für jeden Thread privat. In der gemeinsamen Datenstruktur stehen die Anzahl der Threads, die Startzeit, die Felder mit den Anfangs-, End- und Zwischenwerten und eine globale Pthreads-Barrier. Jeder Thread besitzt darüber hinaus noch eine private Datenstruktur für die Threadnummer und den Start- bzw. Endindex der Komponenten, die er in jedem Zeitschritt berechnen soll. In der Berechnungsfunktion werden zuerst die Daten für die gemeinsamen und privaten Datenstrukturen

```
1  void seq_expl_euler_solve(double t0, double *y0, double H,
     double *y, double h)
   {
     uint i, j, steps;
     double **Y

6    ALLOC2D(Y, 2, ode_size, double);

     steps = H / h;

     copy_vector(Y[0], y0, ode_size);
11
     for (i = 0; i < steps; i++)
     {
       double t = t0 + i * h;
       uint i_mod_2 = i & 1;
16     double *Y_cur = Y[i_mod_2];
       double *Y_new = Y[1 - i_mod_2];

       for (j = 0; j < ode_size; j++)
         Y_new[j] = Y_cur[j] + h*ode_eval_comp(j, t, Y_cur);
21   }

     copy_vector(y, Y[steps & 1], ode_size);

     FREE2D(Y)
26 }
```

Listing 4.1: sequentielle CPU-Variante

initialisiert und danach die einzelnen Threads gestartet. In der Threadfunktion erfolgt fast die gleiche Berechnung wie in der sequentiellen Version. Allerdings berechnet nicht jeder Thread den kompletten Vektor, sondern nur die Komponenten von seinem Start- bis zu seinem Endindex gemäß der Formel 3.4. Zwischen den einzelnen Schritten muss durch eine Barrier sichergestellt werden, dass sich die einzelnen Threads nicht überholen, damit keine falschen Werte zur Berechnung verwendet werden.

In der gesamten Arbeit wurden zur Bestimmung der Laufzeiten der verschiedenen Algorithmen für jede Laufzeit jeweils fünf Messungen vorgenommen. Es wurde aus den fünf verschiedene Messungen jeweils der geringste Werte genommen, um eventuelle Verfälschungen der Laufzeit durch andere Prozesse zu minimieren.

Die Messungen der Pthreads-Version wurde mit dem folgenden System durchgeführt:

	Pthreads
CPU:	
Prozessor	2x AMD Opteron 270
Anzahl Kerne	4
Taktung	2.0 GHz

In Abbildung 4.1 ist der Speedup der Pthreads Implementierung für das Brüsselator- und das String-Problem dargestellt. Der Speedup bezieht sich auf das komplette Programm inklusive Initialisierung. Es ist ein linearer Speedup für genügend große Testsysteme bis zu vier Threads (die Anzahl der Kerne des Prozessors) zu sehen. Beide Testprobleme skalieren also sehr gut mit dem Grad an Parallelität auf Multicoreprozessoren. Aufgrund dessen bietet sich das System gut zur Parallelisierung auf GPUs an, da auf GPUs viele Threads bzw. Workitems gleichzeitig rechnen können. Ist die Anzahl der Threads höher als die der Kerne, bricht der Speedup ein, weil die Arbeit nicht mehr gerecht auf die verschiedenen Kerne aufgeteilt wird.

Grafik 4.2 zeigt die Laufzeiten beider Testprobleme für verschiedene Systemgrößen mit vier Threads. Auf der x-Achse ist die Systemgröße des zu berechnenden Problems logarithmisch aufgetragen, um einen Vergleich zwischen kleinen Systemen (n = 20.000 - 80.000) und großen Systemen (n = 500.000 - 5.000.000) zu erhalten. Auf der y-Achse ist logarithmisch die Laufzeit mit Initialisierung in Sekunden aufgetragen. Es ist zu sehen, dass das Brüsselator-Problem aufwendiger ist und bei gleicher Systemgröße eine höhere Laufzeit besitzt als das String-Problem.

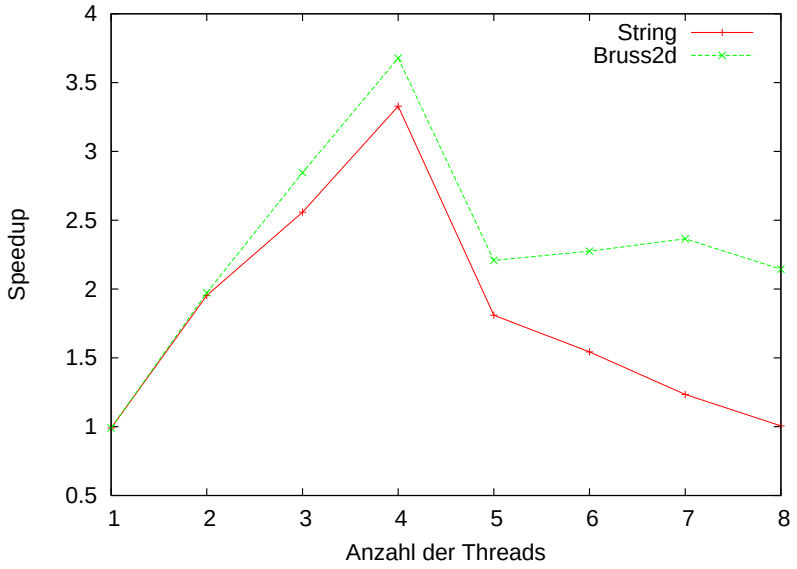


Abbildung 4.1: Speedup der Pthreads Version

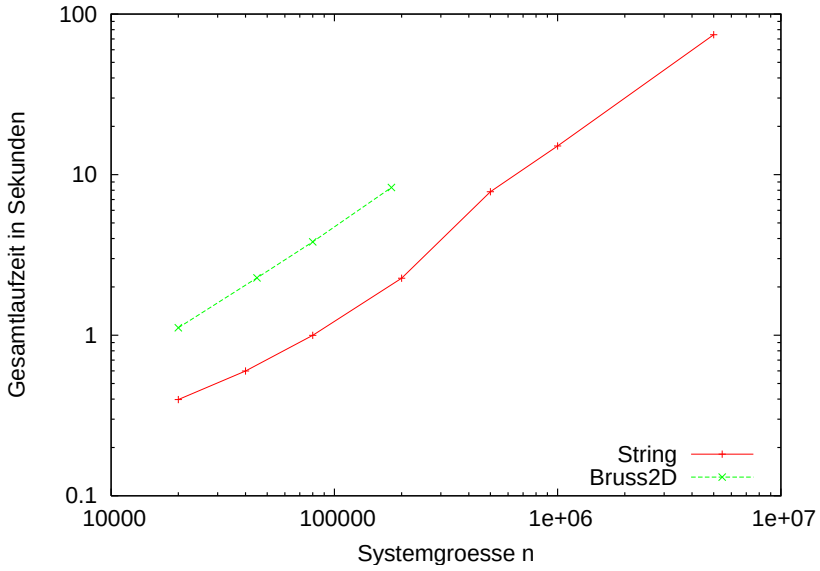


Abbildung 4.2: Laufzeit der Pthreads Version bei einer Systemgröße von 80.000

5 Lineare OpenCL Versionen

Wie in Abbildung 4.1 zu sehen ist, lässt sich das Euler-Verfahren sehr gut parallelisieren und zeigt auf Multicoreprozessoren einen linearen Speedup. Da heutige GPUs aus einer Vielzahl von Recheneinheiten und diese wiederum aus mehreren Kernen bestehen, bietet es sich an, das Euler-Verfahren für Grafikkarten zu implementieren und auf seine Skalierbarkeit zu untersuchen. Für diese Arbeit wurde als Programmierschnittstelle OpenCL benutzt. Da die betreffenden Optimierungen aber zum Großteil unabhängig von OpenCL sind, können die hier entwickelten Algorithmen auch für andere GPU-Programmierschnittstellen, wie zum Beispiel NVIDIA CUDA [10] oder DirectCompute [9], genutzt werden.

Es werden im Folgenden verschiedene OpenCL–Implementierungen des Euler-Verfahrens vorgestellt und miteinander verglichen.

5.1 Unoptimierte lineare OpenCL-Version

Die erste parallele lineare OpenCL-Version des expliziten Euler-Verfahrens ist zum Großteil eine Portierung der Pthreads-Version nach OpenCL. Sie beinhaltet die OpenCL-Initialisierung durch den Host sowie den Kernel für das OpenCL-Device. Eine schematische Version des Hostcodes ist in Listing 5.1 dargestellt. Neben den von der Pthreads-Version bekannten Parametern, gibt es hier weitere: die Gesamtzahl der Workitems (*global*), die Anzahl der Workitems pro Workgroup (*lokal*) und die Datei für den Kernel (*Kernelcode*). Zuerst wird das OpenCL-System initialisiert. Dies beinhaltet das Erstellen der Plattform, des Kontextes, des Devices und der Command Queue (siehe Kapitel 2.1). Danach werden die Dateien aus dem Kernel eingelesen und daraus das OpenCL-Programm (= Kernel) kompiliert. Dies beinhaltet zum einen die Datei für das Lösungsverfahren (*ocl_id*) sowie den Kernel für die ode_eval_comp Funktion des ODE-Problems (*ode_filename*). Da OpenCL nur float-Werte unterstützt, die vorliegenden Daten aber double Werte sind, müssen die Ausgangswerte des Systems erst umgewandelt werden, was zwar einen Overhead darstellt und die Rechengenauigkeit einschränkt, sich aber im Allgemeinen nicht vermeiden lässt (Anm: durch Erweiterung von OpenCL (Extension cl_khr_fp64) sind auf manchen GPUs auch double Werte verfügbar). Danach werden die Ausgangswerte des Systems als Kernelargument dem Device übergeben.

Das Starten der Workitems kann nicht wie in der Pthreads-Version implementiert werden, denn in der Pthreads-Implementierung werden die Threads einmalig gestartet und jeder Thread iteriert selbst über die Zeitschritte. Dies ist aber hier nicht möglich, da OpenCL nur über lokale Barriers verfügt. Bei einer Zugriffsdistanz größer Null gibt es immer Berechnungen, die auf die Werte von anderen

Workgroups zugreifen müssen. Daher wird nach jedem Zeitschritt eine globale Barrier benötigt. Erst wenn alle Werte des Zeitschrittes t_i berechnet wurden, darf die Berechnung für den Zeitschrit t_{i+1} beginnen. Deshalb erfolgt in der OpenCL Implementierung das Iterieren über die Zeitschritte im Host und in jeder Iteration wird ein neuer Kernel gestartet. Die Schleifenstruktur der Pthreads-Version wäre zwar effektiver, da sie nur mit einem Kernelaufruf auskommen würde, aber sie stellt nicht die richtige Abarbeitung der einzelnen Zeitschritte sicher und kann daher nicht in OpenCL verwendet werden.

Die Berechnungsdaten von y werden nur einmalig auf das Device kopiert und bleiben dort für alle Kernelaufrufe vorhanden, werden aber während der Berechnung überschrieben. Zwischen den einzelnen Kernelaufrufen werden jeweils nur der Zeitpunkt t und der Berechnungsschritt i vom Host für das Device aktualisiert. Die modifizierten Daten aus y werden erst am Ende der Berechnung zum Zeitschritt t_e wieder vom Host aus dem Device ausgelesen.

```
void expl_euler (int global, int lokal, String Kernelcode,
  float *y, float h, float H, float t_0) {
  Initialisiere OpenCL
  kompiliere Kernelcode auf Device

  setze Kernelargument (1, h)
  device_y = allokiere Speicher (2 * ode_size * float)
  kopiere y nach device_y
  setze Kernelargument (2, device_y)

  steps = H / h
  for (i = 0; i < steps; i++) {
    float t = t_0 + i * h
    setze Kernelargument (3, t)
    setze Kernelargument (4, i)
    starte Kernel (global, lokal)
    warte auf Beendigung des Kernels
  }

  kopiere device_y nach y
}
```

Listing 5.1: lineare OpenCL Version: Hostcode

Der Kernel für das Lösungsverfahren ist in Listing 5.2 zu finden. Die Kernelmethode benötigt folgende Parameter vom Host: den aktuellen Zeitschritt t, die Schrittweite h, die Iteration (step) und das Feld y, welches die aktuellen und die berechneten Werte enthält. Das Schlüsselwort __global steht dafür, dass die Werte im globalen Speicher stehen. Alle anderen Variablen ohne ein extra Schlüsselwort stehen im privaten Speicher der Workitems. Zuerst wird die globale Workitem-Id *id* und die Anzahl aller Workitems *ids* abgefragt. Der Vektor y ist $2n$-Elemente groß,

um Platz für die aktuellen und die berechneten Werte zu bieten. y_cur und y_new sind dabei Zeiger auf den Vektor y für die aktuellen und berechneten Werte. Nach jedem Zeitschritt werden y_cur und y_new vertauscht, damit die berechneten Werte y_new von Schritt t_i zu den aktuellen Werten y_cur von t_{i+1} werden. Anhand des Iterationszählers *step* kann dann in jedem Zeitschritt bestimmt werden, wo die aktuellen Werte in y stehen und wo die neu berechneten Werte gespeichert werden sollen. Anhand der Workitem-Id und der Anzahl der Workitems kann nun der Anfangs- und Endindex (*first* bzw. *last*) für jedes Workitem bestimmt werden. Danach berechnet jedes Workitem vom Anfangs- bis zum Endindex die neuen Werte für das jeweilige Problem.

```
__kernel void solve (float h, __global float *y, float t,
 int step) {
  int id = globale Workitem Id
  int ids = Anzahl an globalen Workitems

  __global float * y_cur = aktuelle Werte(y, step)
  __global float * y_new = neue Werte(y, step)

  int first = bestimme Anfangsindex (id, ids, ode_size)
  int last = bestimme Endindex (id, ids, ode_size)

  for (int i = first; i <= last; i++)
    y_new[i] = y_cur[i] + h * ode_eval_comp(i, t, y_cur)
}
```

Listing 5.2: unoptimierte lineare OpenCL Version: Kernelcode

Testsystem	Pthreads/GTX280
CPU:	
Prozessor	AMD Opteron 270
Anzahl Kerne	4
Taktung	2.0 GHz
GPU:	
Grafikkarte	NVIDIA GTX 280
Recheneinheiten	30
Anzahl Kerne	240
Kerne pro Recheneinheit	8
globaler Speicher	1023 MB
lokaler Speicher	16 kB
L2 Cache	/

In Abbildung 5.1 ist die Laufzeit der OpenCL-Version im Vergleich zur Pthreads-Version aufgetragen. Es wurde für beide die minimale Laufzeit genommen, um

einen vergleichbaren Wert zu erhalten (Pthreads-Version mit 4 Threads, OpenCL-Version mit 512 lokalen Workitems und 30 Workgroups). Auf der x-Achse ist die Systemgröße aufgetragen und auf der y-Achse die entsprechende Laufzeit in Sekunden. Es wurden verschiedene Laufzeiten gemessen. Die Gesamtlaufzeit ist die komplette Laufzeit des Lösungsverfahrens mit Berechnung und Initialisierung. Die Kernellaufzeit ist dagegen die Laufzeit, die für die Berechnung gebraucht wurde. Es ist zu sehen, dass bei der Pthreads-Version die Gesamt- und Kernellaufzeit nahezu identisch sind, da die Initialisierung bei Pthreads, die Erstellung der Threads, nicht lange dauert. Bei der OpenCL-Version ist dagegen die Initialisierungszeit wesentlich höher. Bei OpenCL muss zuerst die Grafikkarte gefunden und initialisiert werden, der Kernel muss geladen und kompiliert werden und die Daten müssen zwischen Host und Device transferiert werden. Dieser Overhead zur Initialisierung ist für verschiedene Systemgrößen nahezu konstant und deshalb nur bei sehr kleinen Systemen für die Gesamtlaufzeit signifikant.

Die Pthreads-Version ist für kleine Systeme schneller als die OpenCL-Version, da die Initialisierungszeit bei OpenCL hauptsächlich für die Laufzeit verantwortlich ist. Für große Systeme, bei der die Initialisierungszeit keinen wesentlichen Beitrag zur Laufzeit mehr beisteuert, ist die OpenCL-Version doppelt so schnell wie die Pthreads-Version. Es wird deutlich, dass schon dieser einfache unoptimierte Kernel auf einer GPU zu einer Verbesserung der Laufzeit gegenüber einer CPU-Version führt.

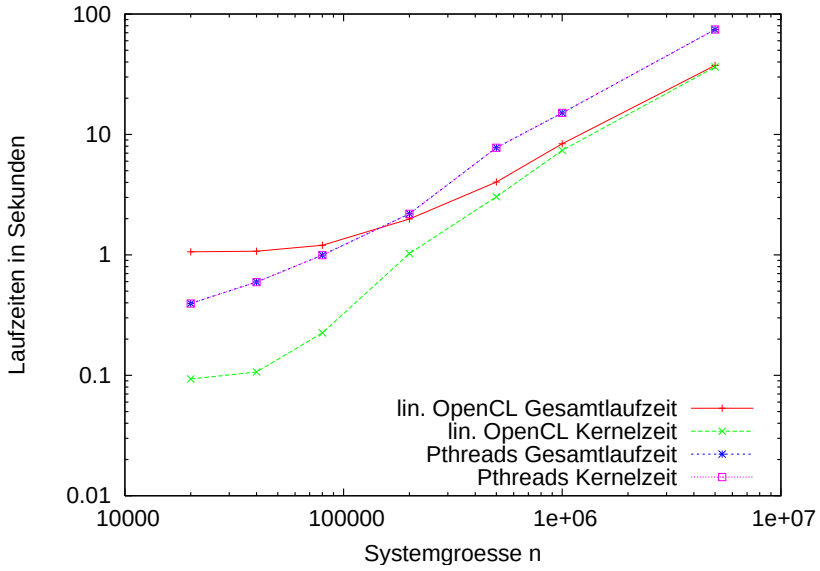


Abbildung 5.1: Laufzeit der linearen OpenCL-Version und der Pthreads-Version

5.2 Verbesserung des Speicherzugriffsmusters

Zur Verbesserung des OpenCL-Kernels wurde zuerst das Zugriffsmuster auf den globalen Speicher verändert (siehe dazu Punkt 2.2.4). Bisher wurde für jedes Workitem beziehungsweise für jeden Thread ein Anfangs- und Endindex berechnet. Dieses Verhalten ist jedoch für heutige CPU-Architekturen optimiert, die nicht einzelne Werte laden, sondern ganze Cachezeilen. Die meisten GPUs besitzen aber keine Caches und laden typischerweise 128-bit Blöcke. In Listing 5.3 ist der Kernel mit verbessertem Zugriffsmuster dargestellt. Jedes Workitem berechnet dabei immer vier float-Werte / 128-bit Blöcke. Der Anfangsindex jedes Workitems ist dabei seine globale Workitem Id * 4. Somit werden in der ersten Iteration der Schleife die ersten $4 * Workitemanzahl$ Werte von y berechnet. Für die nächste Iteration wird der Anfangsindex um genau diese Anzahl von $4 * Workitemanzahl$ weitergezählt. Dies führt dann zu dem in Kapitel 2.2.4 beschriebenen optimalen Speicherzugriffsmuster von GPUs.

In jeder Iteration werden dann vier aktuelle Werte gelesen und vier neue Werte geschrieben. Dies führt implizit zu einem Loop Unrolling (vgl. Kapitel 2.2.1).

```
for (int i = 4 * id; i <= ode size - 4; i+= 4 * ids) {
  y_new[i] = y_cur[i] + h * ode_eval_comp(i, t, y_cur)
  y_new[i+1] = y_cur[i+1] + h * ode_eval_comp(i+1, t,
   y_cur)
  y_new[i+2] = y_cur[i+2] + h * ode_eval_comp(i+2, t,
   y_cur)
  y_new[i+3] = y_cur[i+3] + h * ode_eval_comp(i+3, t,
   y_cur)
}
```

Listing 5.3: optimiertes Speicherzugriffsmuster des linearen OpenCL-Verfahrens

In Grafik 5.2 ist die Kernellaufzeit der unoptimierten und der optimierten linearen OpenCL-Version dargestellt. Es wird deutlich, dass bei kleinen Systemgrößen durch die Optimierung keine Verbesserung der Laufzeiten zu messen ist. Allerdings führt das Speicherzugriffsmuster schon bei mittleren Systemgrößen zu einer erheblichen Verbesserung der Laufzeit. Bei der größten gemessenen Systemgröße führt die Optimierung zu einem Speedup von 3,6 im Vergleich zur nicht optimierten Version.

In Grafik 5.3 sind für das String-Problem die Gesamtlaufzeiten der beiden linearen OpenCL-Versionen der Pthreads-Version gegenübergestellt. Es ist wieder zu sehen, dass die OpenCL-Versionen bei kleinen Systemgrößen durch die Initialisierung schlechtere Laufzeiten haben als die CPU-Version. Bei größeren Systemen sind jedoch die GPU-Versionen wesentlich schneller, die optimierte Version erreicht dabei eine Beschleunigung bis zum 7,4-fachen der CPU-Version.

Um die Laufzeiten miteinander vergleichen zu können, ist es notwendig, die optimale Anzahl an lokalen und globalen Workitems für jeden Kernel zu bestim-

men. Die lokalen Workitems, also die Workitems pro Workgroup, werden durch die NVIDIA GTX 280 auf maximal 512 begrenzt. In Grafik 5.4 ist der Speedup der Kernellaufzeiten gegenüber den lokalen Workitems aufgetragen. Es wurde dabei bei jeder Kurve eine feste Anzahl an Workgroups eingestellt und die lokalen Workitems von eins bis 512 variiert. Feste Anzahl pro Workgroup bedeutet, dass die Anzahl der globalen Workitems sich aus:

$$globale\_Workitems = lokale\_Workitems * Workgroups \tag{5.1}$$

berechnet.

Der Speedup jeder Kurve wurde dabei jeweils auf die Laufzeit mit einem globalen Workitem bezogen. Bei einer Workgroup ist zu Beginn ein linearer Anstieg des Speedups zu beobachten, der später dann immer weiter abflacht. Das Maximum von 250 wird bei 512 Workitems, also der maximalen Anzahl, erreicht. Die optimale Anzahl an Workitems pro Workgroup ist hier die maximale Anzahl an Workitems pro Workgroup für die entsprechende GPU. Es stellt sich also die Frage warum dies so ist, wenn nur 8 Kerne pro Recheneinheit existieren. Eine Besonderheit dieses Kernels sind die vielen Speicherzugriffe auf den globalen Speicher. Je mehr Workitems zur Verfügung stehen, desto besser kann die GPU die Latenzen der Speicherzugriffe verstecken. Anders sieht es bei 30 Workgroups aus, also einer vollen Auslastung der GPU. Hier gibt es nur bis ca. 64 Workitems pro Workgroup einen starken Anstieg des Speedups bis auf das 1100-fache (entspricht etwa einem Speedup vom 35-fachen pro Workgroup). Danach bleibt der Speedup konstant. Dies liegt daran, dass bei 30 Workgroups viel mehr globale Workitems aktiv sind und auf den Speicher zugreifen. Bei etwa 30 Workgroups mit jeweils 64 Workitems, also 1920 globalen Workitems, ist die Bandbreite des globalen Speichers ausgelastet und es kommt zu keinem weiteren Speedup mehr.

In Grafik 5.5 wurden die Workitems pro Workgroup konstant gehalten und die Anzahl der Workgroups variiert. Bei allen Kurven ist anfangs wieder ein linearer Anstieg des Speedups zu sehen, der dann abflacht und sich einer Obergrenze nähert. Wenn die Kurve abflacht, ist wieder die maximale Bandbreite erreicht. Je größer die Anzahl der lokalen Workitems ist, desto weniger Workgroups werden benötigt, um diesen maximalen Speedup zu erreichen.

Für diesen Kernel sollte die Anzahl an lokalen und globalen Workitems möglichst hoch gewählt werden um den maximalen Speedup zu erhalten (zum Beispiel lokale Workitems: 512, Workgroups: 30 auf der GTX 280).

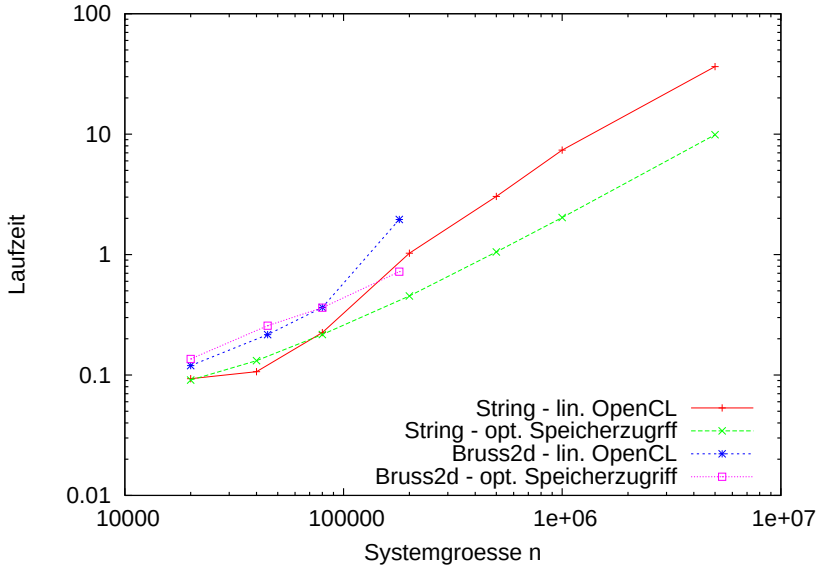


Abbildung 5.2: Laufzeiten der linearen OpenCL Implementierungen

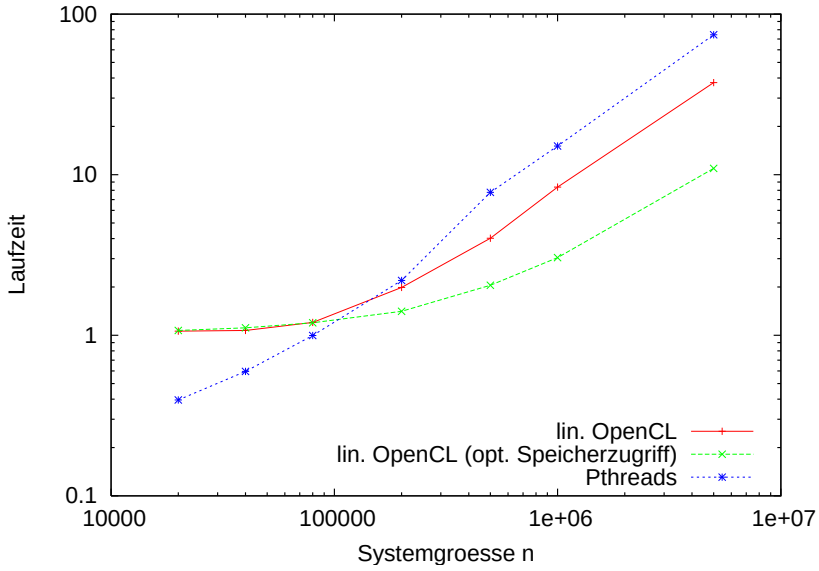


Abbildung 5.3: Laufzeiten der Implementierungen

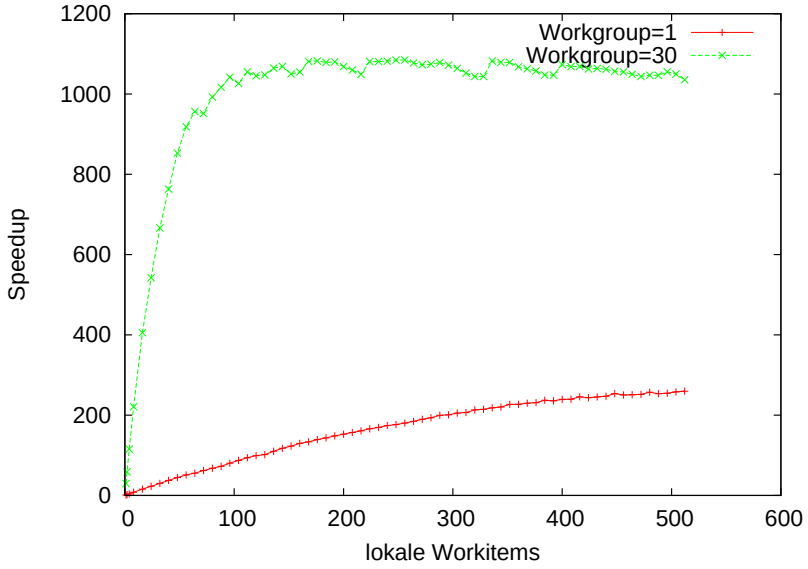


Abbildung 5.4: Speedup der Workgroups

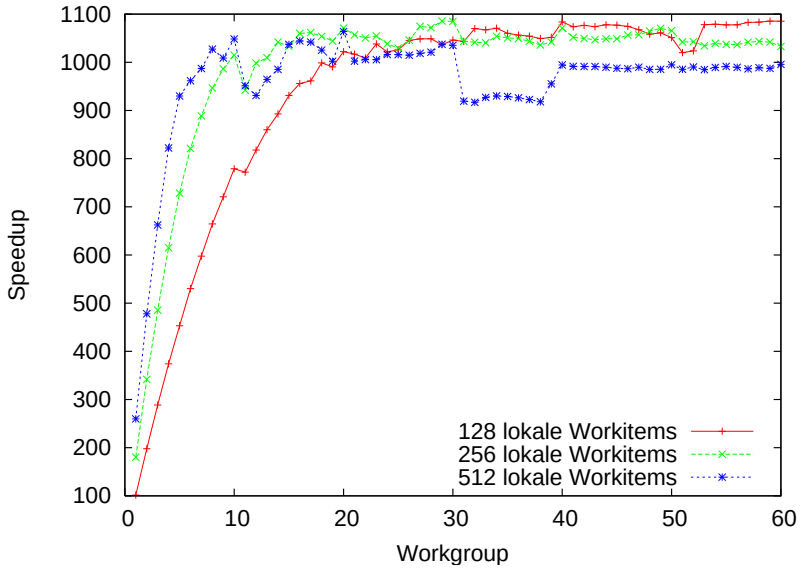


Abbildung 5.5: Speedup der lokalen Workitems

5.3 Vektorisierung der Problemfunktion

Da sich die vier Berechnungszeilen in Kernel 5.3 sehr ähnlich sehen, liegt es nahe, diese zu vektorisieren (vgl. Kapitel 2.2.2). Die Schleife sieht dann so aus:

```
for (int i = id; i < ODE_SIZE / 4; i += ids) {
  y_new[i] = y_cur[i] + h * ode_eval_comp4(i, t, y_cur);
}
```

Die normale Problemfunktion berechnet für einen gegebenen Index genau einen Rückgabewert. Wird die Schleife vektorisiert, muss auch die Problemfunktion angepasst werden, sodass für einen gegebenen Index vier Rückgabewerte berechnet werden. In Listing 5.6 und 5.7 sind die originale und die vektoriserte Problemfunktion für das String-Problem abgebildet. Daran ist ersichtlich, dass das Umschreiben nicht trivial ist, da alle Feldzugriffe auf die vektorisierten Werte angepasst werden müssen. Bei komplexeren Funktionen sollte daher abgewägt werden, ob sich der Aufwand für das Umschreiben der Funktion im Vergleich zum Geschwindigkeitsgewinn lohnt.

```
float ode_eval_comp(..)
{
  if(i % 2 == 0)
    return y[i+1];

  return STRING_mod_K * STRING_mod_K *
    (y[i-3] - 2 * y[i-1] + y[i+1]);
}
```

Abbildung 5.6: original String-Problem

```
float4 ode_eval_comp4(..)
{
  float4 res;
  res.x = y[i].y;
  res.z = y[i].w;

  res.y = STRING_mod_K * STRING_mod_K *
    (y[i-1].z - 2 * y[i].x + y[i].z);
  res.w = STRING_mod_K * STRING_mod_K *
    (y[i].x - 2 * y[i].z + y[i+1].x);

  return res;
}
```

Abbildung 5.7: vektorisiertes String-Problem

In Grafik 5.8 wird die Kernellaufzeit der vektorisierten Version des String-Problems mit der bisherigen Version verglichen. Es ist zu sehen, dass das Vektorisie-

ren der Problemfunktion einen Speedup um das 2,5-fache zur nicht vektorisierten Funktion gebracht hat. Bei der Berechnung größerer Systeme ist es sinnvoll, die Problemfunktion umzuschreiben.

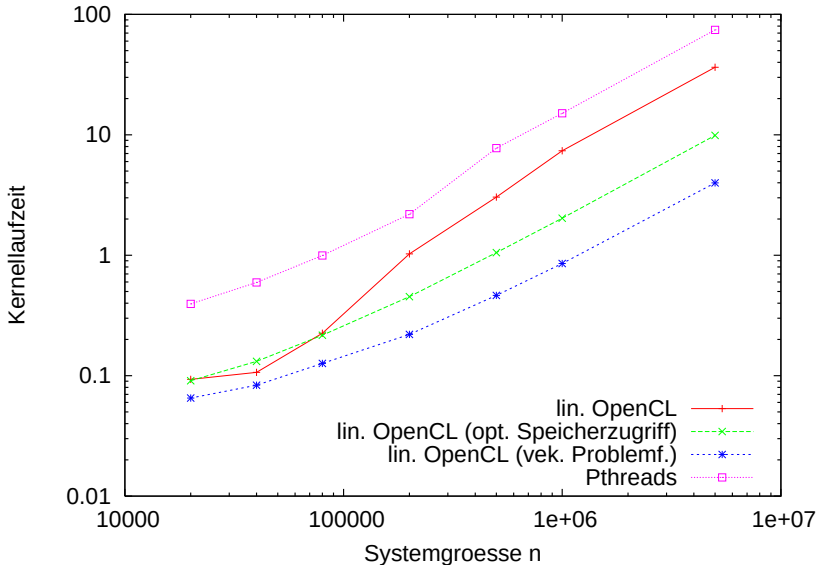


Abbildung 5.8: Laufzeit der vektorisierten Problemfunktion

5.4 Probleme und Erkenntnisse bezüglich des linearen OpenCL-Verfahrens

Durch das Loop-Unrolling, das Verändern der Zugriffsstruktur auf den globalen Speicher und das Anpassen der Problemfunktion konnten Laufzeitverbesserungen erzielt werden. Um aber weitere Optimierungsschritte anzugehen, bedarf es einer genaueren Analyse des Algorithmus, für weitere Laufzeitverbesserungen. Ein Problem ist, dass in jedem Kernelaufruf nur ein Zeitschritt berechnet wird. Dies führt dazu, dass das Synchronisieren des Hosts viel Zeit benötigt. Wie in Kapitel 2.2.6 beschrieben, sollte versucht werden ein Großteil dieser globalen Barriers durch lokale zu ersetzen.

Weiterhin laufen bisher alle Speicheroperationen über den globalen Speicher, welcher der langsamste Speicher ist. Es sollte also versucht werden, einen Großteil der Berechnungsdaten vom globalen Speicher in den lokalen Speicher zu verlegen, um teure I/O-Speicherzugriffe zu sparen.

6 Diamant-Tiling

6.1 Motivation

Viele Anfangswertprobleme der ODE-Systeme besitzen eine beschränkte Zugriffsdistanz. Das heißt, die Zugriffsdistanz ist wesentlich kleiner als die Problemgröße n. Für die folgenden Algorithmen wird davon ausgegangen, dass das Problem eine beschränkte Zugriffsdistanz aufweist, anderenfalls muss einer der vorherigen Algorithmen benutzt werden.

Das Prinzip des Diamant-Tilings wurde in [12] benutzt, um Finite-Differenzen-Methoden (FDTD, finite-difference time-domain) auf Mehrkernprozessoren zu berechnen. Durch den Einsatz von Diamant-Tiling wurde die Speicherwiederverwendung eröht und es konnten dadurch Laufzeitverbesserungen erzielt werden. Es wurden dabei auch unterschiedliche Tileformen untersucht. Dabei stellte sich heraus, dass die Diamant-Tiles die beste Performance für Datanabhängigkeiten bei FDTD bieten.

In der vorliegenden Arbeit wird darauf aufgebaut und das Diamant-Tiling in verschiedenen Implementierungen verglichen. In Kapitel 9 wird auch eine alternative Tileform untersucht.

6.2 Synchronisation

Basierend auf der Zugriffsdistanz *acc_dist* kann der n-stellige Vektor in *Blöcke* gleicher Größe mit jeweils *block_size* ($\geq$ Zugriffsdistanz *acc_dist*) Werten unterteilt werden. Dies hat den Vorteil, dass die Werte in Block j zum Zeitschritt t_i nur ihren eigenen Block j, den vorhergehenden Block $j-1$ und den nachfolgenden Block $j+1$ benötigen (siehe Abbildung 6.1). Damit kann das Problem behoben werden, dass pro Kernelaufruf nur ein Zeitschritt berechnet wird. Die Idee ist, jeder Recheneinheit eine bestimmte Anzahl (= *dia_blocks*) an Blöcken zuzuteilen. Aus diesen *dia_blocks* Blöcken zum Zeitpunkt t_i können *dia_blocks* -2 neue Blöcke für den Zeitschritt t_{i+1} berechnet werden. Es werden pro berechnetem Zeitschritt zwei Blöcke weniger, da die äußersten Blöcke nicht mit berechnet werden können, weil sie Abhängigkeiten zu Blöcken anderer Workgroups haben. Der Vorteil dabei ist, dass nach einem Zeitschritt die Workitems der Workgroup über eine lokale Barrier synchronisiert werden können. Erst wenn die Anzahl der Blöcke nur noch 2 beträgt und keine weiteren Blöcke mehr berechnet werden können, muss der Kernel beendet werden und es wird global synchronisiert. Aus diesem Muster ergibt sich die untere Hälfte eines *Diamanten* (vgl. Abbildung 6.1).

Die Diamanten werden auf die Workgroups / Recheneinheiten verteilt. Dabei wird ein ähnliches Muster benutzt wie in Kapitel 2.2.4. Jede Workgroup beginnt mit dem Diamanten ihrer Workgroup Id (Anm: eindeutiger Index jeder Workgroup), für jeden weiteren Diamanten wird der Iterationszähler um die Anzahl der Workgroups weitergezählt, bis alle Diamanten der Iteration berechnet wurden (in Abbildung 6.2 wurden die Diamanten auf zwei Workgroups verteilt). Die Berechnungen einer Workgroup sind unabhängig von denen anderer Workgroups und werden parallel zu diesen ausgeführt. Das Berechnen von mehreren Diamanten einer Workgroup erfolgt sequentiell. Erst wenn alle Workgroups ihre Diamanten berechnet haben, kann die nächste Iteration gestartet werden. Nachdem alle unteren Hälften berechnet wurden, wird deshalb eine globale Barrier benötigt, um die Konsistenz der berechneten Daten sicherzustellen.

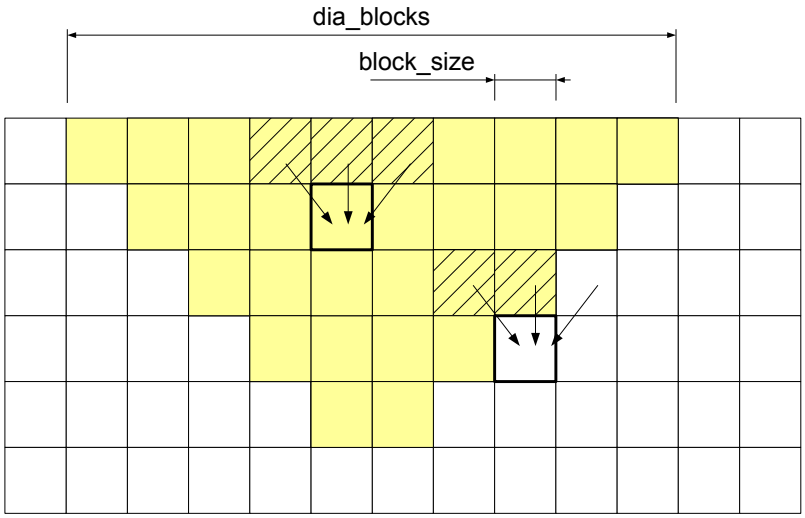


Abbildung 6.1: Aufbau eines Diamanten

6.3 Speicherverfahren

Da die Berechnung jedes Diamanten nur auf einer Recheneinheit erfolgt, können die Daten für den Diamant im lokalen Speicher der entsprechenden Workgroup abgelegt werden. Aus den Abbildungen 6.1 und 6.2 ist ersichtlich, dass die nächsten Diamanten die zwei äußersten Blöcke (schraffierte Blöcke) jedes Zeitschrittes der unteren Hälfte für ihre Berechnung der oberen Hälfte des Diamanten benötigen. In jedem Zeitschritt der unteren Hälfte müssen also die zwei äußersten Blöcke wieder in den globalen Speicher geschrieben werden.

In Abbildung 6.3 ist zu sehen, dass es durch das Vertauschen von y_new und y_cur in jedem Zeitschritt möglich ist, alle Blöcke des Diamanten zu speichern, ohne noch benötigte Werte zu überschreiben. In der Abbildung werden alle geraden

Zeitschritte (grob schraffiert) in dem vorderen Teil des $2n$ Werte großen Vektors y geschrieben und die ungeraden Zeitschritte (fein schraffiert) im hinteren Teil des Vektors.

Bei der Berechnung der oberen Hälfte des Diamanten müssen die zwei äußersten Blöcke, die von den vorhergehenden Diamanten berechnet wurden, geladen werden. Die Daten werden dabei vom globalen Speicher in den lokalen Speicher kopiert. Die Berechnungen der Zeitschritte innerhalb des Diamanten erfolgt nur noch auf Daten im lokalen Speicher. Die Berechnungszeit innerhalb des Diamanten verkürzt sich, da die Zugriffszeiten auf den lokalen Speicher geringer sind als auf den globalen Speicher und nur noch auf Daten im lokalen Speicher zugegriffen wird. Es entsteht aber auch ein Mehraufwand, da die Daten erst zwischen den verschiedenen Speichern kopiert werden müssen.

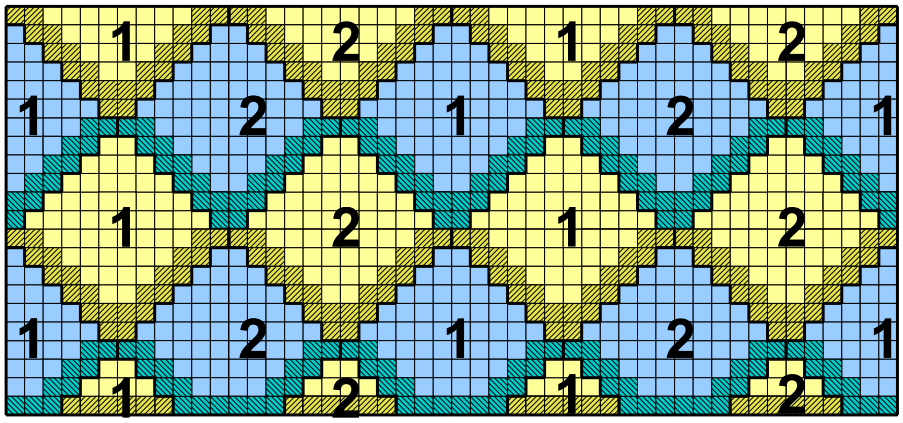


Abbildung 6.2: Vollständiges Diamant-Schema, Aufteilung der Diamanten auf (2 Workgroups)

6.4 Bestimmung der Diamantgröße

Eine ideale Aufteilung wäre es, die n Werte in so viele Diamanten wie es Recheneinheiten gibt zu unterteilen, damit jede Workgroup einen Diamant berechnet. Damit ergibt sich für die Anzahl der Blöcke pro Diamant:

$$dia\_blocks = \frac{n}{block\_dist * Recheneinheiten} \tag{6.1}$$

Um Fallunterscheidungen und degenerierte Diamanten zu verhindern, wird die Anzahl der dia_blocks Blöcke immer auf die nächstgrößere gerade Zahl erhöht, da für die Diamanten, wie sie in Abbildung 6.1 dargestellt sind, immer eine gerade Anzahl an Blöcken benötigt wird. Ein anderes wichtiges Entscheidungskriterium für die Blockanzahl pro Diamant ist die Größe des lokalen Speichers. Ist dieser

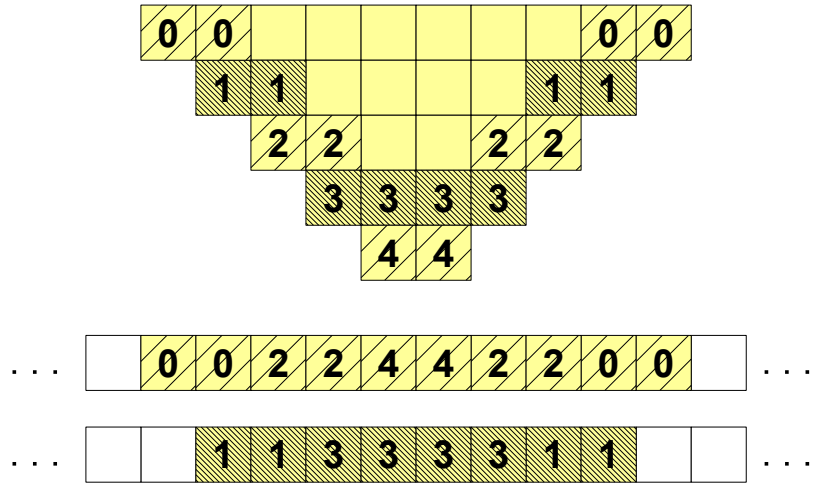


Abbildung 6.3: Speichern der Diamanten im globalen Speicher

zu klein, muss auch die Anzahl der Blöcke verringert werden. Außerdem ist eine gewisse Mindestgröße für *dia_blocks* erforderlich, denn je kleiner der Diamant wird, desto weniger Zugriffe werden auf den globalen Speicher werden eingespart. Diese Abwägungen werden im folgenden Beispielen veranschaulicht:

Auf einer Nvidia GTX280 und einer Systemdimension von n = 80.000
ergibt sich für die Probleme String und Brüsselator folgendes:

a) Das String-Problem hat eine Zugriffsdistanz von 4
Daraus ergibt sich für die Anzahl der Blöcke:
dia_blocks = 80.000 / (4 * 30) = 666,666 → 668
Jeder Diamant besteht also ideal aus 668 Blöcken

b) Das Brüsselator-Problem hat hier eine Zugriffsdistanz von 200
Daraus ergibt sich für die Anzahl der Blöcke:
dia_blocks = 80.000 / (200 * 30) = 13,333 → 14
Jeder Diamant besteht also ideal aus 14 Blöcken

Wie in der Beispielrechnung zu sehen ist, ist die Blockanzahl sehr von der Zugriffsdistanz abhängig. Je kleiner die Zugriffsdistanz ist, desto mehr Blöcke pro Diamant gibt es. Je mehr Blöcke es pro Diamant gibt, desto mehr Zeitschritte kann ein Diamant berechnen und desto weniger globale Barriers werden benötigt.

In den folgenden Kapiteln werden verschiedene Implementierungen des Diamant-Verfahrens vorgestellt, die verschiedene Speicherverfahren und Optimierungen benutzen.

7 Zeilenweises Diamant-Tiling

Die zuerst betrachtete Implementierung des Diamant-Verfahrens ist ein zeilenweises Verfahren. Der Algorithmus berechnet, wie die vorhergehenden Verfahren, die Werte des Diamanten pro Zeitschritt parallel. Alle Workitems einer Workgroup iterieren dabei parallel über einen Zeitschritt. Erst nach dem Zeitschritt erfolgt eine lokale Barrier, bevor der nächste Zeitschritt beginnt (siehe Bild 7.1).

In der oberen Diamanthälfte werden pro Zeile jeweils vier Blöcke aus dem globalen Speicher nachgeladen (schraffierte gelbe Blöcke). Die Berechnung der Blöcke erfolgt dann im lokalen Speicher (blaue Blöcke). Dazu wird, wie in den anderen Implementierungen, jeweils ein Feld für die neuen und die alten Werte benutzt und dieses wird alternierend pro Zeitschritt vertauscht. In der unteren Hälfte des Diamanten werden jeweils vier neu berechnete Blöcke pro Zeitschritt in den globalen Speicher geschrieben (blau schraffierte Blöcke).

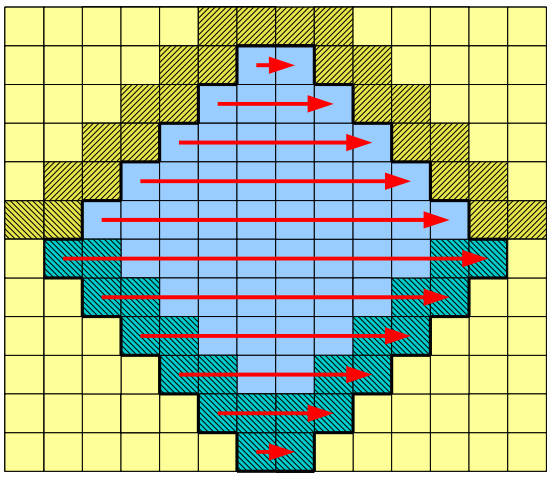


Abbildung 7.1: zeilenweises Diamant Tiling

7.1 Implementierung des Hostcodes

Nach dem Initialisieren erfolgt das Bestimmen der charakteristischen Größen des Diamanten (siehe Listing 7.2). Dazu wird zunächst die Größe des lokalen Speichers abgefragt. Die Größe eines Blockes (*block_size*) ergibt sich aus dem nächstgrößeren Vielfachen von vier der Zugriffsdistanz. Die Blöcke dürfen nicht kleiner gewählt werden als die Zugriffsdistanz, da sonst nicht mehr sichergestellt ist, dass ein Eintrag eines Blockes nur maximal die Nachbarblöcke für die Berechnung benötigt. Eine Vergrößerung der Blöcke ist nicht sinnvoll, da durch größere Blöcke insgesamt mehr Speicher- und Ladeoperationen auf den globalen Speicher anfallen. Außerdem führen größere Blöcke zu einer kleineren Blockanzahl pro Diamant, sodass sich die Anzahl der Zeitschritte pro Diamant verringert und sich damit die Anzahl der globalen Barriers erhöht. Eine minimale Vergrößerung auf das nächstgrößere Vielfache von vier ist notwendig, da der Kernel mit Vektoren der Größe vier (float4) vektorisiert ist.

In *dia_blocks_sum* wird die Anzahl aller zu berechnenden Blöcke, die auf die Diamanten verteilt werden müssen, gespeichert.

```
void expl_euler (...) {
Initialisere OpenCL
lokaler Speicher = Device Info (LOCAL_MEMORY)

int block_size = Zugriffsdistanz erhöht auf das nächste
 Vielfache von 4
int dia_blocks_sum = ode_size / block_size

Diamanten = Anzahl der Recheneinheiten
for (;;) {
  dia_blocks = dia_blocks_sum / Diamanten

  if (dia_blocks < Minimun Anzahl an Blöcken)
    exit

  if (Größe des Diamanten < lokaler Speicher)
    break

  Erhöhe Anzahl der Diamanten
}
...
```

Abbildung 7.2: Hostcode

Die *dia_blocks_sum* Blöcke einer Zeile müssen auf die verschiedenen Diamanten verteilt werden. In der for-Schleife in Listing 7.2 wird die Anzahl an Blöcken pro Diamant passend zur jeweiligen lokalen Speichergröße ermittelt. Zunächst wird die optimale Anzahl dia_blocks an Blöcken pro Diamant berechnet, um eine gleich-

mäßige Verteilung der Blöcke auf die Recheneinheiten zu gewährleisten. Für diese Anzahl wird der benötigte Speicherverbrauch im lokalen Speicher errechnet. Der lokale Speicherverbrauch ist zweimal die Anzahl der Blöcke + zwei Randblöcke multipliziert mit der Größe eines Blockes in Bytes (vgl. Abbildung 7.3 und Formel 7.1). Ist der lokale Speicher zu klein für diese Anzahl an Blöcken, muss die Anzahl reduziert werden. Das heißt aber auch, dass einige oder alle Workgroups mehr als einen Diamanten zu berechnen haben. Mit einer niedrigeren Anzahl an Blöcken wird der Speicherverbrauch pro Diamant geringer und auch die Anzahl der Zeitschritte pro Diamant wird geringer.

$$\text{benötigter lokaler Speicher} = 2 * (dia\_blocks + 2) * block\_size * 4 \text{ byte} \quad (7.1)$$

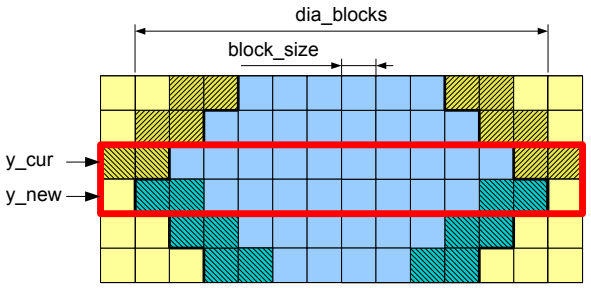


Abbildung 7.3: Lokaler Speicherverbrauch beim zeilenweisen Diamant Tiling

7.1.1 Vergrößerung der Anzahl der Diamanten

Sollte der lokale Speicher nicht für die errechnete Anzahl an Blöcken pro Diamant ausreichen, muss diese Anzahl verringert werden. Verringert sich die Anzahl an Blöcken pro Diamant, dann vergrößert sich die Anzahl der Diamanten, so dass es mehr Diamanten als Recheneinheiten gibt. Für die Erhöhung der Anzahl von Diamanten wurden drei unterschiedliche Strategien getestet.

multiplikative Strategie

Die multiplikative Strategie (multStrat) berechnet die Anzahl der Diamanten durch:

$$Diamanten = mult * Recheneinheiten \qquad mult \in \mathbb{N} \quad (7.2)$$

Das heißt, dass die Anzahl der Diamanten in jedem Schritt um die Anzahl der Recheneinheiten inkrementiert wird. Der Faktor *mult* gibt dabei an, wie viele Diamanten eine Recheneinheit zu berechnen hat. Ist der lokale Speicher zu klein für den maximalen Diamant wird *mult* erhöht und die Blockanzahl reduziert sich. Dies geschieht, bis eine Anzahl an Blöcken gefunden wurde, für die der lokale Speicher ausreichend groß ist. Wird die Anzahl der Blöcke reduziert, heißt das auch, dass

jede Workgroup mehr als einen Diamanten zu berechnen hat. Dieses Verfahren zur Berechnung der Diamantengröße hat den Vorteil, dass die Anzahl der Diamanten immer ein Vielfaches der Recheneinheiten ist, sodass im Sinne der Lastbalancierung jede Workgroup die gleiche Anzahl an Diamanten berechnen muss.

Zur Veranschaulichung wird diese Problemstellung betrachtet:

Auf der GTX280 wird das String-Problem mit einer Systemgröße von 80.000 berechnet:

lokaler Speicher	= 16.384 kB
ode_size	= 80.000
block_size	= 4
Gesamtanzahl der Blöcke	= 20.000
Recheneinheiten	= 30

Die Schleife aus 7.1 benötigt hierfür zwei Iterationen:

mult	dia_blocks	benötigter lokaler Speicher	
1	668	21.440	$>$ 16.384
	→ Speicherverbrauch größer als lokaler Speicher → mult erhöht sich		
2	334	10.752	$<$ 16.384
	→ Speicherverbrauch kleiner als lokaler Speicher		

Jede Workgroup berechnet also zwei Diamanten in jeder Iteration mit jeweils 334 Blöcken.

additive Strategie

Die additive Strategie (addStrat) berechnet die Anzahl der Diamanten durch:

$$Diamanten = mult + Recheneinheiten \qquad mult \in \mathbb{N} \tag{7.3}$$

Hierbei wird die Gesamtanzahl der Diamanten immer nur um jeweils eins erhöht. Demzufolge berechnen nicht alle Workgroups einen Diamanten mehr, sondern nur eine Workgroup berechnet einen Diamanten mehr. Der Nachteil der ersten Strategie ist, dass durch die Halbierung der Blöcke im schlechtesten Fall nur der halbe lokale Speicher benutzt wird. In der zweiten Strategie wird versucht möglichst den kompletten lokalen Speicher auszunutzen.

Mit dieser Strategie ergibt sich für die Blockanzahl mit der oben beschriebenen Problemstellung:

mult	dia_blocks	benötigter lokaler Speicher
0	668	21.440
1	646	20.736
2	626	20.096
3	608	19.520
...		
10	500	16.064

nach zehn Iterationen wird eine Blockanzahl gefunden
deren Speicherbedarf kleiner als der lokale Speicher ist
die Ausnutzung des Speichers ist hierbei wesentlich besser als bei der multiplikativen Strategie

Als Resultat erhält man folgende Aufteilung:
20 Workgroups berechnen einen Diamanten mit 500 Blöcken
10 Workgroups berechnen zwei Diamanten mit 500 Blöcken

multiplikativ korrigierte Strategie

Die multiplikativ korrigierte Strategie (korrStrat) berechnet die Anzahl der Diamanten durch:

$$Diamanten = mult * Recheneinheiten - 1 \qquad mult \in \mathbb{N} \tag{7.4}$$

Die Anzahl der Diamanten ist immer um eins geringer als bei der ersten Strategie. Die Anzahl der Diamanten wird für die geraden Iterationen berechnet (siehe gelbe Blöcke in Abbildung 6.2). In ungeraden Iterationen entstehen am Anfang und Ende jeweils zwei halbe Diamanten und die Anzahl der Diamanten erhöht sich um eins. Die Workgroup, die den ersten halben Diamanten berechnet, berechnet auch den letzten halben Diamanten. Im Idealfall braucht die Berechnung für die zwei halben Diamanten genauso lange wie die Berechnung eines ganzen Diamanten, dann wäre die erste Strategie der Diamantverteilung optimal. Dies würde bedeuten, dass die Rechenzeit hauptsächlich zur Berechnung der Elemente verbraucht wird und die Rechenzeit für die Kontrollstrukturen und die Barriers vernachlässigt werden kann. Im schlechtesten Fall braucht ein halber Diamant genauso lange wie ein ganzer Diamant, wenn die Kontrollstrukturen und die Barriers signifikant für die Rechenzeit sind, da die Anzahl der Barriers für alle Diamanten identisch sind. Das heißt, dass mit der ersten Strategie eine Workgroup doppelt so viel wie die anderen Workgroups berechnet. In diesem Fall wäre die dritte Strategie, bei der beide halben Diamanten von unterschiedlichen Workgroups berechnet werden, optimal.

Mit dieser Strategie ergibt sich für die Blockanzahl mit der oben beschriebenen Problemstellung:

mult	dia_blocks	local_size
1	690	22.144
2	340	10.944

Als Resultat erhält man folgendes:
in geraden Iterationen berechnen 30 Workgroups zwei Diamanten mit jeweils 340 Blöcken
in ungeraden Iterationen berechnen 29 Workgroups zwei Diamanten, zwei Workgroups berechnen nur einen Diamanten

Laufzeiten der drei Strategien

In Abbildung 7.4 ist die Kernellaufzeit mit den unterschiedlichen Strategien dargestellt für verschiedene Systemgrößen n. Bei einer Systemgröße von über 60.000 reicht der vorhandene lokale Speicherplatz nicht mehr für alle Diamanten und einige Workgroups müssen mehrere Diamanten berechnen. Den größten Laufzeitsprung macht dabei die zweite Strategie, da hier nur eine Workgroup zwei Diamanten berechnen muss und alle anderen Workgroups warten müssen, bis der erste seine Arbeit beendet hat. Bei der ersten und dritten Strategie müssen alle Workgroups zwei Diamanten berechnen, die aber nur die halbe Größe haben. Bei einer weiteren Erhöhung der Systemgröße bis etwa 120.000 kommt es zu keiner weiteren Vergrößerung der Laufzeit der zweiten Strategie, da schon die erste Workgroup zwei Diamanten voller Größe berechnet und damit die höchste Laufzeit aufweist. Bei den anderen beiden Varianten kommt es zu einem gleichmäßigen Anstieg der Laufzeit. Es ist zu sehen, dass die Sprünge bei der Erhöhung der Diamantanzahl signifikant für die Laufzeit sind. Die Vergrößerung der Blockanzahl pro Diamant trägt jedoch weniger zur Laufzeit bei. Meistens wird durch die dritte Strategie die minimale Laufzeit für verschiedene Systemgrößen erzielt.

Mit den ermittelten Werten für die charakteristischen Größen des Kernels kann der Kernel initialisiert werden. Dem Kernel wird zusätzlich zu den vorherigen Argumente noch die Blockgröße $block\_size$, die Zugriffsdistanz und die Anzahl der Blöcke dia_blocks übergeben.
Danach folgt die Schleife über die Integrationsschritte, die die einzelnen Kernelaufrufe startet. Da jeder Diamant so viele Zeilen hoch hoch wie er Blöcke breit ist, berechnet jeder Diamant auch dia_blocks Zeitschritte. Die nächste Iteration startet dann genau $(dia\_blocks/2) * h$ später als der vorherige. (siehe Listing 7.5).

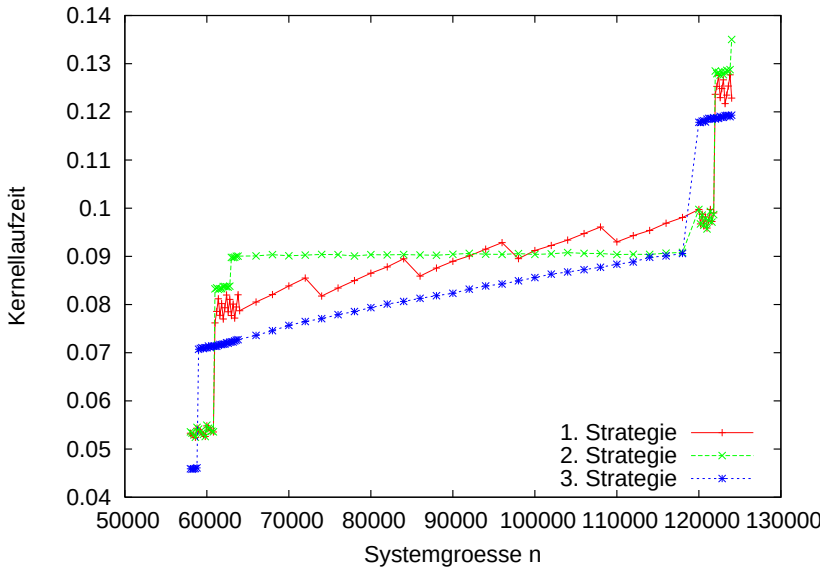


Abbildung 7.4: unterschiedliche Diamantverteilungsstrategien

```
...
int steps = H / h;
float t = t_0;
int max_iter = (2 * steps) / dia_blocks;

setze Kernelargument (block_size)
setze Kernelargument (dia_blocks)

for (iter = 0; iter <= max_iter; ++iter) {
  setze Kernelargument (iter)
  setze Kernelargument (t)

  starte Kernel (global, lokal)
  warte auf Beendigung des Kernels

  t = iter * dia_tiles / 2 * h_f;
}
```

Abbildung 7.5: Kernel starten im Hostcode

7.2 Implementierung des Kernels

Der Ablauf des Kernels ist in Listing 7.6 schematisch dargestellt. Der Vektor y enthält die Ausgangsdaten, die vom Host zum Device kopiert wurden. Der Vektor y_l ist ein Speicherbereich im lokalen Speicher der Recheneinheit in dem die Workgroups die Berechnungen durchführen. Jede Workgroup wird auf einer Recheneinheit ausgeführt und besitzt deshalb einen lokalen Speicherbereich, der unabhängig von den anderen Workgroups ist. Das lokale Feld jeder Workgroup ist dabei anfangs nicht vorinitialisiert.

```
def __kernel solve( ...,
  __global float4 *y,
  __local float4 *y_l)
{
  diamanten = Anzahl der Diamanten pro Zeile
  steps = dia_tiles / 2 - 1;
  for (group = 0; group < diamanten; group+=workgroups) {
    // obere Diamanthälfte
    for (step = steps; step >= 0; step--){
      lade Werte aus y nach y_l
      berechne neue Werte in y_l
    }
    // untere Diamanthälfte
    for (step = 1; step <= steps; step++) {
      berechne neue Werte in y_l
      speichere Werte von y_l nach y
    }
    barrier
  }
}
```

Abbildung 7.6: Pseudocode des zeilenweisen Diamant-Verfahrens

Die Anzahl der pro Zeitschritt zu berechnenden Diamanten wurde zuvor im Host bestimmt. Jede Workgroup besitzt eine Schleife über die Anzahl der Diamanten, die sie zu berechnen hat. Da für einen Diamanten der gesamte lokale Speicher benutzt wird, müssen die Diamanten auf einer Recheneinehit nacheinander berechnet werden. Die Diamanten auf verschiedenen Recheneinheiten werden aber parallel bearbeitet.

Die Variable *steps* gibt die Höhe bzw. die Anzahl der Zeitschritte pro Diamanthälfte an. $step = 0$ entspricht dem Zeitschritt in der Mitte des Diamanten, bei dem die maximale Anzahl an Blöcken zu berechnen ist. Jede andere Stufe *step* des Diamanten besitzt $2 * (steps - step)$ Blöcke.

Zunächst wird die obere Hälfte des Diamanten berechnet (siehe Abbildung 7.7 von step 5 bis 0). In dieser Phase werden in jedem Zeitschritt 4 Blöcke (schraffierte gelbe Blöcke) vom globalen Speicher in den lokalen Speicher kopiert. Die

Berechnung der neuen Blöcke (schraffierte blaue Blöcke) erfolgt danach nur noch im lokalen Speicher.

Der lokale Speicher ist wieder in zwei Teilbereiche für y_new und y_cur unterteilt, die nach jedem Zeitschritt vertauscht werden. In Abbildung 7.7 ist zu sehen, dass in $step = 5$ in das linke Feld ($= y\_cur$) die Werte vom globalen Speicher geladen werden und auf der rechten Seite des Feldes (y_new) werden die neu berechneten Werte abgelegt. Im nächsten Zeitschritt ($step = 4$) wird dann das rechte Feld zu y_cur und die nächsten Werte vom globalen Speicher werden neben die zuvor berechneten Werten geladen. Die neu berechneten Werte werden dann im linken Feld abgespeichert und überschreiben die zuvor geladenen Werte, da diese nicht mehr gebraucht werden.

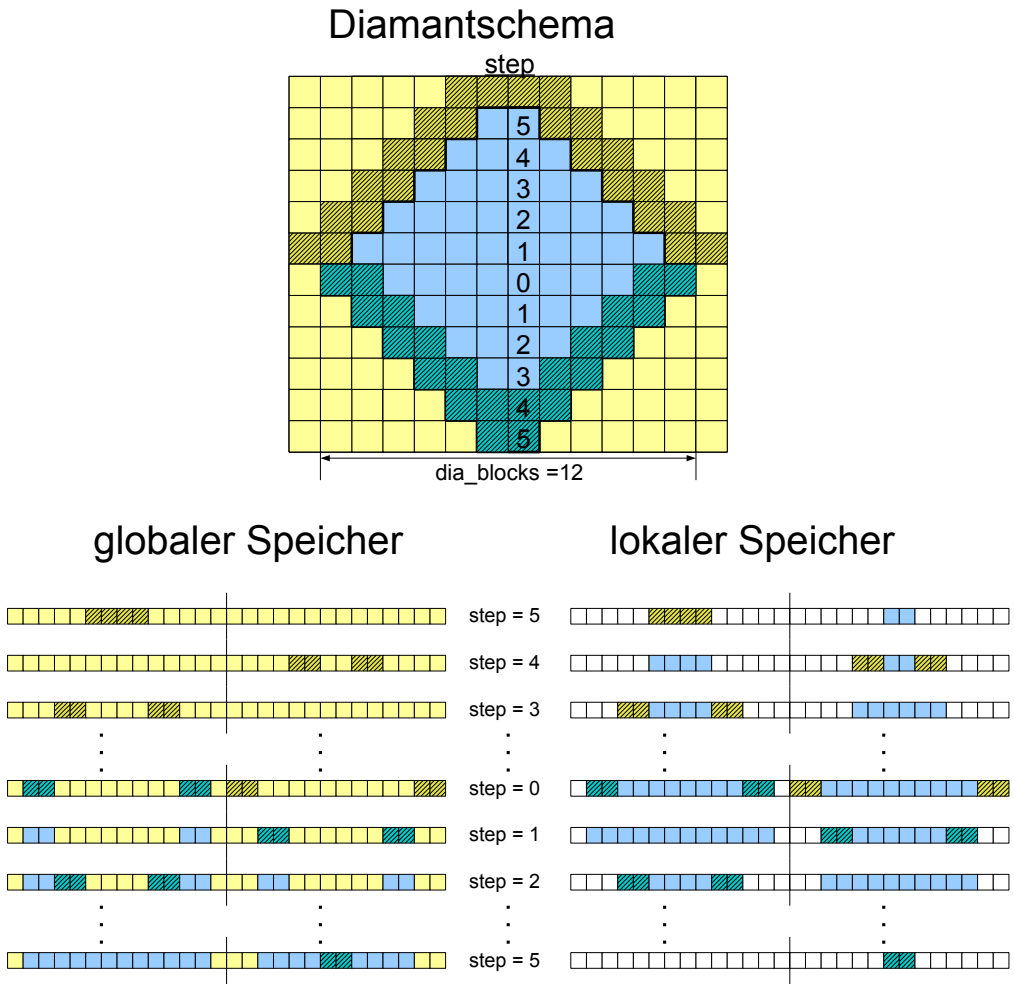


Abbildung 7.7: Speicherverfahren des zeilenweisen Diamant-Tilings

Das Laden, Speichern und Berechnen von Werten erfolgt wieder nach dem Schema aus Kapitel 2.2.4. Es wird vor jedem Laden, Speichern und Berechnen der Start- und Endindex der Werte berechnet und von allen Workitems parallel bearbeitet.

```
id = lokale Workitem Id
n = Anzahl der lokalen Workitems
offset = Startadresse des Diamanten im globalen Speicher
block_end = Endadresse des Diamanten im globalen Speicher
local_offset = Offset des Diamanten im lokalen Speicher
...

int begin = offset + step * block_size;
int end = block_end - step * block_size;

for (i = begin + id; i < end; i += n) {
  local_i = i - offset + local_offset;
  y_new[local_i].x = y_cur[local_i].x + h * ode_eval_comp
   (4 * i, t, &y_cur[local_offset - offset]);
  ...
}
```

Abbildung 7.8: Berechnungen beim Diamant-Verfahren im lokalen Speicher

Nachdem die obere Hälfte berechnet wurde, kann die untere Hälfte berechnet werden (in Abbildung 7.7 unten von step 1 bis 5). In dieser Phase der Berechnung eines Diamanten sind schon alle benötigten Daten im lokalen Speicher vorhanden und es können sofort die nächsten Zeitschritte berechnet werden ohne Daten neu aus dem globalen Speicher anzufordern. Nach dem Berechnen jeder Zeile müssen die äußersten vier Blöcke wieder in den globalen Speicher geschrieben werden, da sie zur Berechnung der nächsten Diamanten benötigt werden. Nach der Abarbeitung des Diamanten besteht der komplette globale Vektor aus neu berechneten Werten und diese können für den nächsten Diamanten wieder als Ausgangswerte benutzt werden.

(Anm: In Abbildung 7.7 besteht der globale Bereich nicht nur aus neu berechneten Werten nach $step = 5$. Dies liegt daran, dass es nur ein Ausschnitt des globalen Speichers ist und die noch gelben Blöcke von anderen Diamanten berechnet werden.)

7.3 Laufzeit und Speedup des zeilenweisen Diamant-Tilings

Testsystem	Pthreads/GTX280	GTX580
CPU:		
Prozessor	2x AMD Opteron 270	Intel E5530
Anzahl Kerne	2	4
Taktung	2.0 GHz	2.4 GHz
GPU:		
Grafikkarte	NVIDIA GTX 280	NVIDIA GTX 580
Recheneinheiten	30	16
Anzahl Kerne	240	512
Kerne pro Recheneinheit	8	32
globaler Speicher	1023 MB	1536 MB
lokaler Speicher	16 kB	48 kB
L2 Cache	/	768 kB
Betriebssystem	Linux Kernel 2.6.37	
Compiler	gcc 4.6.2	

Im Folgenden werden die gemessenen Speedups der OpenCL-Version mit Diamant-Tiling betrachtet. Es wurde für alle Messungen das String-Problem mit einer Systemgröße von 80.000 verwendet. In Abbildung 7.9 wird der Speedup in Abhängigkeit von der Anzahl der Workgroups mit 128 und 256 lokalen Workitems pro Workgroup dargestellt. Hierbei sollte beachtet werden, dass wegen dem wesentlich komplexeren Kernelcode nun mehr Register als in der linearen OpenCL-Version benötigt werden. Da jede Workgroup nur eine bestimmte Anzahl an Registern für alle Workitems besitzt, ist die maximale Anzahl lokaler Workitems nur noch 256. Für eine höhere Anzahl an Workitems kann ein identisches Verhalten beobachtet werden. Es gibt einen linearen Speedup bis zu 30 Workgroups, da bei 30 Workgroups alle 30 Recheneinheiten komplett ausgelastet sind. Bei über 30 Workgroups gibt es einen starken Abfall des Speedups, da es jetzt eine ungleichmäßige Verteilung der Diamanten/Workgroups auf die Recheneinheiten gibt. Bei 60 Workgroups gibt es wieder eine Lastbalancierung und es ergibt sich der gleiche Speedup wie bei 30 Workgroups. Die Anzahl der Workgroups sollte in der Diamant-Version also möglichst ein Vielfaches der Recheneinheiten sein.

In Grafik 7.10 ist der Speedup in Abhängigkeit von der Anzahl lokaler Workitems dargestellt. Es ist wieder anfangs ein linearer Anstieg des Speedups festzustellen, der sich ab 125 lokalen Workitems einer Konstanten nähert. Der maximale Speedup von 750 liegt bei etwa 160 lokalen Workitems, es werden also wieder wesentlich

mehr lokale Workitems als Kerne pro Recheneinheit benötigt, um den maximalen Speedup zu erreichen. Dies lässt darauf schließen, dass wieder Speicherbandbreiten den Speedup begrenzen. Die Anzahl der Workitems sollte also im Idealfall auf über 100 lokale Workitems gesetzt werden.

In Abbildung 7.11 sind die Laufzeiten der Diamant-Versionen im Vergleich zu den vorherigen Algorithmen aufgetragen. Es werden zusätzlich zu dem bereits vorgestellten Diamant-Verfahren noch zwei weitere betrachtet. Ein Verfahren benutzt lediglich globalen Speicher und keinen lokalen Speicher und bei einem anderen wurde zusätzlich die Problemfunktion vektorisiert. Vergleichbar sind daher das Diamant-Verfahren mit dem linearen OpenCL-Verfahren und die beiden Verfahren mit der vektorisierten Problemfunktion. Die Diamant-Version erreicht dabei einen Speedup von 1,75-fache, der linearen OpenCL Version. Das heißt, dass die Verwendung des lokalen Speichers und der Einsatz lokaler Barriers die Laufeit des Kernels fast halbiert. Bei den Versionen mit der vektorisierten Problemfunktion ist der Speedup durch das Diamant-Verfahren jedoch wesentlich geringer. Bei kleinen Systemgrößen wird noch ein Speedup von 1,5 erreicht, bei größeren Systemen nur von 1,05. Die Ursache dafür ist in Abbildung 7.12 zu erkennen.

In Abbildung 7.12 ist die normierte Kernellaufzeit gegenüber der Systemgröße aufgetragen. Die normierte Kernellaufzeit berechnet sich aus:

$$\text{normierte Kernellaufzeit} = \frac{\text{Kernellaufzeit}}{\text{Systemgröße n}} \tag{7.5}$$

Sie stellt die durchschnittliche Berechnungszeit für ein einzelnes Element dar. Bei beiden Diamant-Versionen ist zu sehen, dass mit wachsender Systemgröße die normierte Laufzeit geringer wird (bis zu einer Systemgröße von 60.000). Dies liegt daran, dass bei größeren Systemen auch die einzelnen Diamanten größer werden und damit der lokale Speicher besser ausgenutzt werden kann und es weniger globale Barriers gibt. Ab einer Systemgröße von 60.000 müssen aber mehrere Diamanten von einer Workgroup berechnet werden (siehe Abbildung 7.4). Deshalb steigt die Laufzeit sprunghaft wieder an. Bei 60.000, also einer vollen Ausnutzung des lokalen Speichers, wird ein Minimum der normierten Laufzeit erzielt, da bei einer vollen Ausnutzung des lokalen Speichers der höchste Speedup des Diamant-Verfahrens erreicht wird. Jede weitere Vergrößerung des Systems bringt keinen weiteren Speedup mehr für jedes einzelne Element. Dieses Speedupmaximum wird immer wieder bei Vielfachen von 60.000 erreicht. Der Speedup des Diamant-Verfahrens ist also primär abhängig von der Größe des lokalen Speichers.

Die lineare OpenCL-Version dagegen hat einen stetigen Speedup pro Einzelelement. Bei kleinen Systemen ist die normierte Laufzeit wesentlich schlechter als die der Diamant-Versionen, bei größeren Systemen wird aber annähernd die Laufzeit der Diamant-Version erreicht. Der Speedup der linearen OpenCL-Version ist stark abhängig von der Anzahl der globalen Barriers. Da bei allen Systemgrößen die Anzahl der globalen Barriers konstant ist, fällt die Ausführungszeit für die globalen Barriers für kleine Systeme wesentlich stärker ins Gewicht als bei größeren Systemen, bei denen die Berechnung der Elemente signifikant für die Laufzeit ist.

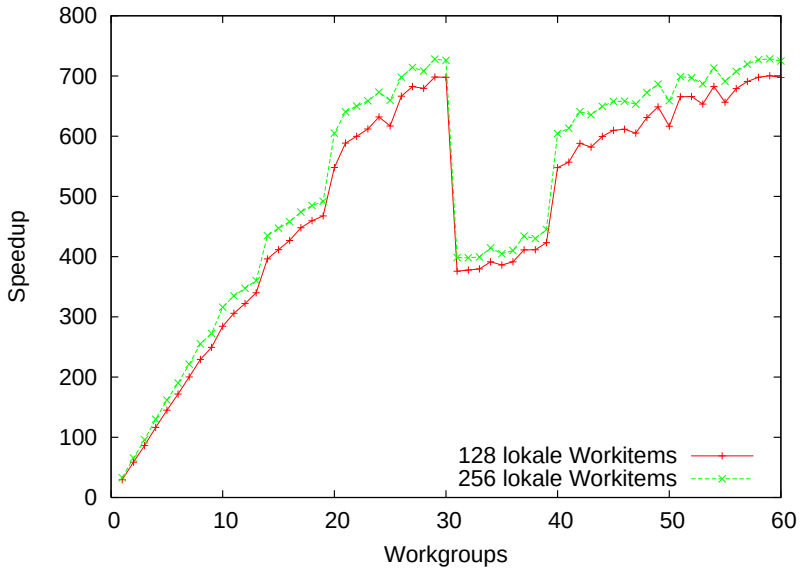


Abbildung 7.9: Speedup der lokalen Workitems

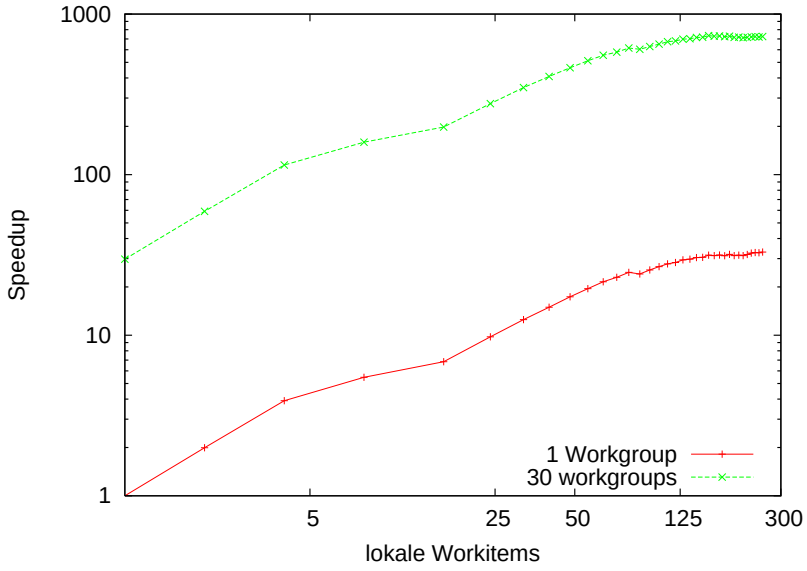


Abbildung 7.10: Speedup der Workgroups

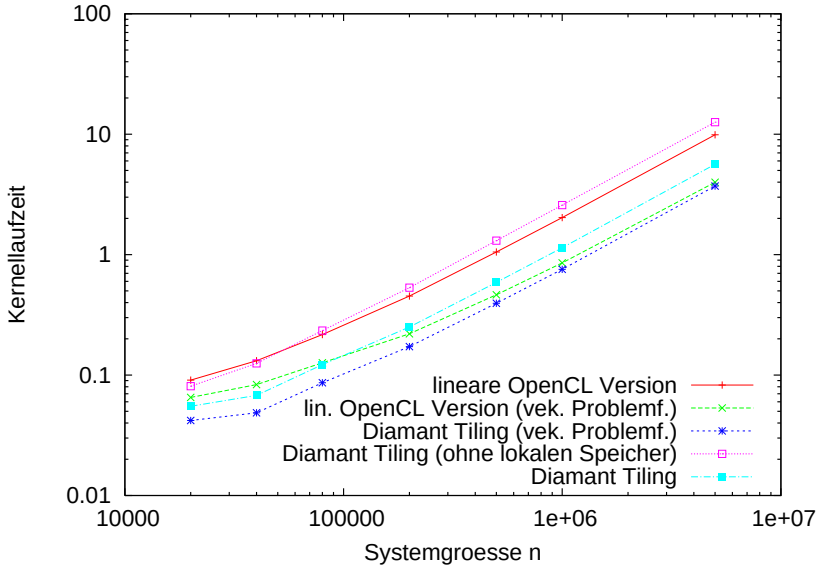


Abbildung 7.11: Laufzeit der Versionen

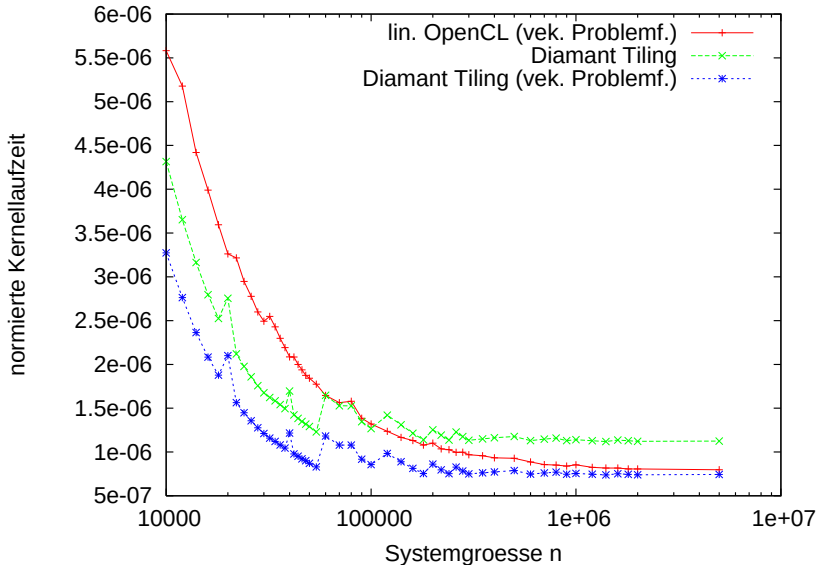


Abbildung 7.12: normierte Laufzeit der Versionen

Wenn der Speedup der Diamant-Version primär von der Größe des lokalen Speichers abhängt, dann müsste sich die Laufzeitdifferenz der Diamant-Version zur linearen OpenCL-Version bei GPUs mit einem größeren lokalen Speicher vergrößern. Um dies zu untersuchen, wurden die Laufzeitmessungen auch auf einer NVIDIA GTX 580 durchgeführt, die 48kB lokalen Speicher besitzt, im Gegensatz zur GTX 280 mit nur 16kB. Dabei muss jedoch beachtet werden, dass die GTX 580 einen L2 Cache besitzt. Jede Speicheranfrage auf den globalen Speicher lädt daher nicht nur einen 32bit Wert, sondern 128Byte, also eine komplette Cachezeile. In Abbildung 7.13 ist die normierte Laufzeit bei verschiedenen Problemgrößen zu sehen. Die Laufzeiten der Diamant-Versionen und der linearen Versionen sind nahezu identisch. Dies liegt daran, dass die lineare Version linear auf den globalen Speicher zugreift und daher stark vom Cache profitiert. Die Diamant-Version lädt immer nur einzelne Blöcke vom globalen Speicher und kann daher weit weniger vom Cache profitieren als die lineare Version. Da die Diamant-Version für GPUs ohne Cache optimiert wurde, wurde in Abbildung 7.14 der Cache durch deklarieren der Vektoren als *volatile* umgangen. *volatile* bewirkt, dass die GPU gezwungen wird, jeden Wert zu laden und damit nicht vom Cache profitieren kann. Die Laufzeiten der Diamant-Versionen bleiben dabei nahezu konstant, da sie nicht vom Cache profitieren. Die Laufzeiten der linearen Versionen steigen jedoch. Außerdem ist zu beobachten, dass die linearen Versionen vom Vektorisieren der Problemfunktion stark profitieren und beim Diamant-Verfahren das Vektorisieren zu fast keinem Speedup führt. Dies lässt darauf schließen, dass beim Diamant-Verfahren die Speicherbandbreite erschöpft ist und, dass daher die Vektorisierung der Problemfunktion nur noch zu einer geringen Beschleunigung führt.

In Abbildung 7.15 wurden die Laufzeiten für das Brüsselator-Problem auf der GTX 580 gemessen. Das Brüsselator-Problem hat eine wesentlich größere Zugriffsdistanz als das String-Problem, welche auch mit der Größe des Systems wächst. Dies hat zur Folge, dass die Blöcke der Diamanten wesentlich größer sind. Für das Diamant-Tiling wird eine gewisse Mindestanzahl an Blöcken vorausgesetzt, damit es effektiv arbeiten kann. Durch die geringe lokale Speichergröße wurde daher auf der GTX 280 auf Messungen mit dem Brüsselator-Problem verzichtet. Die Abbildung 7.15 zeigt die normierten Laufzeiten für verschiedene Systemgrößen auf der GTX 580. Ohne Cache ist die Diamant-Version wesentlich schneller als die lineare Version, mit Cache ist die lineare Version besser als die Diamant-Version, da sie sehr stark vom L2 Cache profitiert.

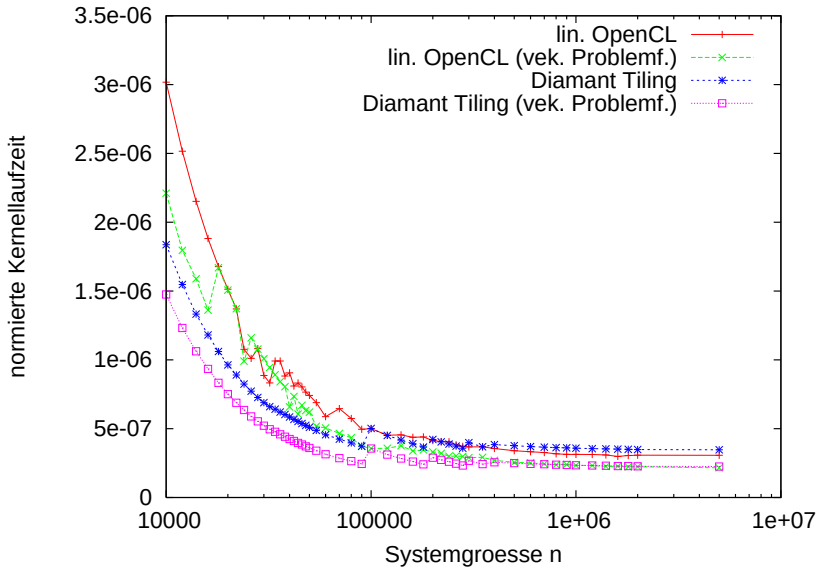


Abbildung 7.13: normierte Laufzeit auf der GTX 580 String-Problem

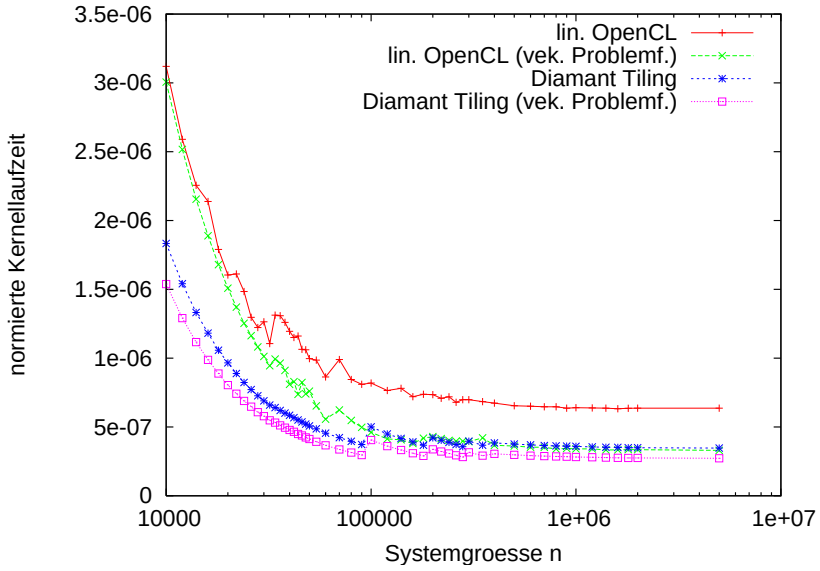


Abbildung 7.14: normierte Laufzeit auf der GTX 580 String-Problem (volatile)

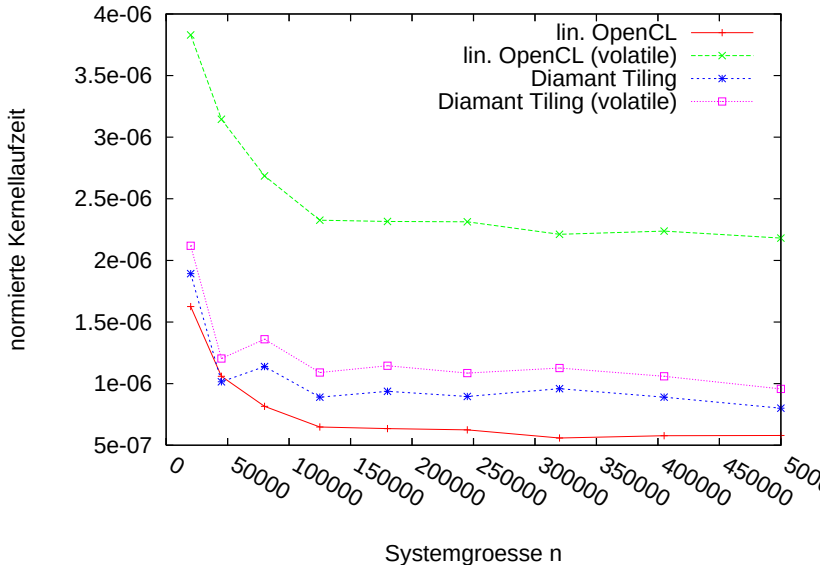


Abbildung 7.15: normierte Laufzeit auf der GTX 580 Bruss2d Problem

7.4 Probleme und Erkenntnisse des Diamant-Tilings

Das Diamant-Tiling ist eine Optimierung des Euler-Verfahrens für GPUs ohne L2 Cache. Gerade bei kleineren und mittleren Systemgrößen bietet es Laufzeitvorteile gegenüber dem linearen OpenCL-Verfahren. Das Diamant-Tiling profitiert weniger von der Vektorisierung der Problemfunktion, deshalb bietet es sich an, das Diamant-Tiling bei komplexen Problemfunktionen zu benutzen, ohne den Aufwand des Umschreibens der Problemfunktion zu haben. Der Speedup des Diamant-Tilings ist primär von der Größe des lokalen Speichers abhängig. Eine Möglichkeit zur weiteren Optimierung des Diamant-Verfahrens ist es, den lokalen Speicher effektiver auszunutzen. Dazu kann das Speicherverfahren so verändert werden, dass nicht in jedem Zeitschritt zwei vollständige Zeilen des Diamanten im lokalen Speicher benötigt werden.

8 Spaltenweises Diamant-Tiling

Die zweite Implementierung des Diamant-Tilings ist ein spaltenweises Verfahren. Das Ziel des spaltenweisen Diamant-Tilings ist es, durch eine bessere Ausnutzung des lokalen Speichers größere Diamanten pro Workgroup berechnen zu können. Der Speicherverbrauch pro Diamant beim zeilenweisen Verfahren ist:

$$\mathbf{2} * (dia\_blocks + 2) * block\_size * sizeof(float) \tag{8.1}$$

Es wird also etwa zweimal die Größe der Blöcke ($block\_size$) mal die Anzahl der Blöcke (dia_blocks) benötigt. Beim spaltenweisen Diamant-Verfahren wird durch ein besseres Speichermuster der Speicherbedarf pro Diamant halbiert, auf nur noch:

$$(dia\_blocks + 2) * block\_size * sizeof(float) \tag{8.2}$$

Um das zu gewährleisten, werden die Diamanten nicht mehr zeilenweise abgearbeitet, sondern in schrägen Spalten (siehe Abbildung 8.1). Dabei wird ein Nachteil dieses Verfahrens deutlich: Konnten beim zeilenweisen Verfahren noch ganze Zeilen parallel bearbeitet werden, muss nun jeder Block einzeln berechnet werden. Um Block 3 in Abbildung 8.4 zu berechnen, muss zuvor schon Block 1 berechnet worden sein. Daher ist das spaltenweise Verfahren gerade für Probleme mit großer Zugriffsdistanz und damit großen Blöcken geeignet. Bei Problemen mit kleinen Zugriffsdistanzen und damit kleinen Blöcken müssen die Blöcke gegebenenfalls vergrößert werden, um genügend Arbeit für alle Workitems einer Workgroup bereitzustellen.

8.1 Implementierung des Hostcodes

Der Hostcode des spaltenweisen Diamant-Verfahrens ist nahezu identisch mit dem des zeilenweisen Verfahrens. Der einzige Unterschied betrifft die Berechnung der Blockgröße. Beim zeilenweisen Verfahren war die Blockgröße immer die Größe der Zugriffsdistanz, denn je kleiner die Blöcke sind, desto mehr Zeitschritte können berechnet werden und umso kleiner wird die Anzahl der globalen Speicheroperationen. Beim spaltenweisen Verfahren wird aber immer nur ein Block parallel berechnet. Daher ist es notwendig die Blockgröße bei kleinen Zugriffsdistanzen zu vergrößern, damit jedes Workitem mindestens einen float4 Wert zu berechnen hat.

Zu untersuchen ist auch, ob es beim spaltenweisen Diamant-Tiling von Vorteil ist, die lokalen Workitems auf die maximale Anzahl zu setzen, da eine hohe Workitemanzahl zu einer großen Blockgröße führt.

Abbildung 8.2 zeigt die Laufzeit des Kernels für die Probleme String und Brüsselator mit einer Systemgröße von 80.000 und mit verschiedenen Anzahlen von

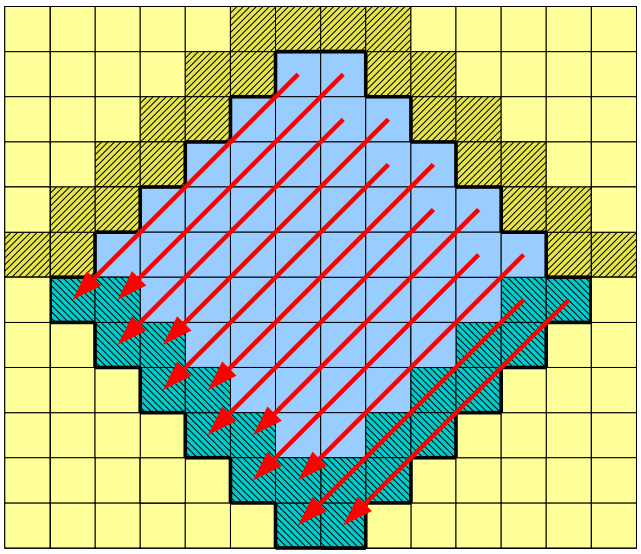


Abbildung 8.1: Spaltenweises Diamant-Tiling

lokalen Workitems. Die Messungen zeigen, dass die maximale Workitemanzahl die beste Laufzeit erbringt.

8.2 Implementierung des Kernels

In Abbildung 8.4 ist das Speicherverfahren des spaltenweisen Diamant-Tilings an einem Beispiel dargestellt. Das Gundverfahren basiert darauf, dass die Spalten von links nach rechts bearbeitet werden und nicht mehr benötigte Blöcke wieder überschrieben werden. Der berechnete Block k in Zeitschritt t_i überschreibt jeweils den Block $k-1$ vom Zeitschritt t_{i-1}, denn der Block $k-1$ von t_{i-1} kann höchstens noch zur Berechnung von Blöcken benötigt werden, die links von Block k von t_i sind und daher schon vorher berechnet wurden.

Ein Diamant mit dia_blocks Blöcken hat dia_blocks diagonale Spalten zu berechnen. Jede Spalte besteht dabei aus $dia\_blocks/2$ Blöcken. Der Speicherverbrauch im lokalen Speicher pro Diamant ist dabei die Größe zweier diagonaler Spalten und zweier Randblöcke (siehe Gleichung 8.2). Zur Berechnung eines Blockes der ersten Diagonale ($diag = 0$) werden jeweils zwei Blöcke aus dem globalen Speicher geladen und in den lokalen Speicher kopiert. Mit diesen Werten können die neuen Werte der ersten Diagonalen berechnet werden (Blöcke 1,3,7 in Abbildung 8.4). In jeder nachfolgenden Diagonale muss nur noch beim ersten Block ein neuer Block aus dem globalen Speicher geladen werden.

Das Speichern der lokalen Blöcke in den globalen Speicher erfolgt hauptsächlich

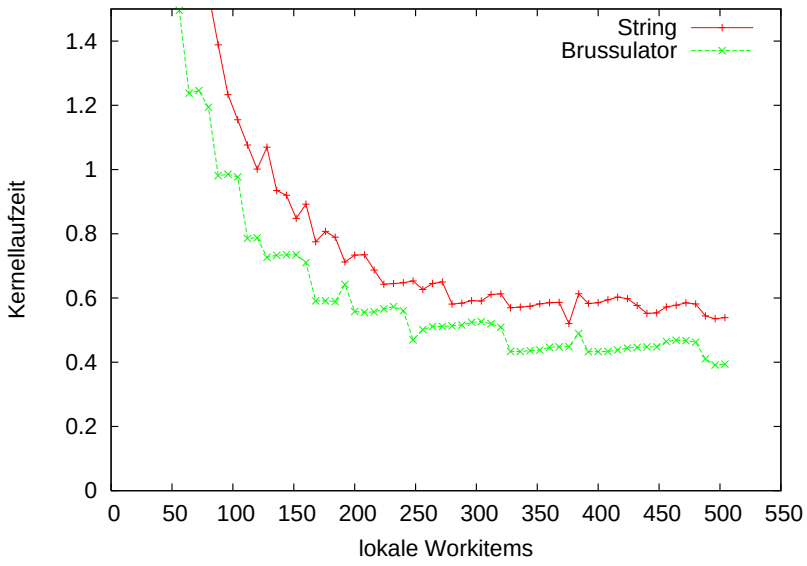


Abbildung 8.2: Anzahl lokaler Workitems

in den letzten beiden Diagonalen (rechte untere Diamanthälfte). Die Blöcke der linken unteren Diamanthälfte, die gespeichert werden müssen, sind die jeweils letzten Blöcke in jeder Diagonalen und die beiden letzten Diagonalspalten. Der Ablauf des Kernels ist in Listing 8.3 skizziert.

Optimierung des Bedarfs an zu kopierenden Werten

Eine Vergrößerung der Blöcke führt dazu, dass der Block k vom Zeitschritt t_i nicht mehr alle Werte von Block $k-1$ bzw. $k+1$ vom Zeitschritt t_{i-1} benötigt, sondern nur noch die Werte des Blockes rechts bzw. links bis zur Zugriffsdistanz. Um unnötige Lade- und Speicheroperationen zu sparen, die durch eine Vergrößerung der Blockgröße entstehen, werden die äußersten Blöcke nicht vollständig geladen/ gespeichert. In Abbildung 8.4 muss also der gesamte Block J geladen werden, da er komplett von Block 7 gebraucht wird, der Block I muss aber nur so weit geladen werden wie er gebraucht wird, also nur die Werte von rechts bis zur Zugriffsdistanz müssen geladen werden. Beim Speichern muss also der gesamte Block 7 kopiert werden, der Block 8 muss aber wiederum nur bis zur Zugriffsdistanz (von links) gespeichert werden.

Es folgt ein Vergleich der zu kopierenden Werte mit und ohne Optimierung des spaltenweisen Diamant-Tilings bei sehr kleinen Zugriffsdistanzen:

```
def __kernel solve( ... ,
  __global float4 *y,
  __local float4 *y_l)
{
  groups = Anzahl der Diamanten // = mult
  steps = dia_tiles;
  for (group = 0; group < groups; group+=workgroups) {
    for (diag = 0; diag < steps; diag++){
      for(height = 0; height < steps/2; height++) {
        if(diag == 0 || height == 0)
          lade Werte aus y in y_l
        berechne Werte in y_l
        if(diag >= steps - 2 || height == steps - 1)
          speichere Werte aus y_l in y
        barrier
      }
    }
  }
}
```

Abbildung 8.3: Pseudocode des spaltenweisen Diamant-Verfahrens

Betrachtet wird das String-Problem mit einer Zugriffsdistanz von 4
(Anm: Zugriffsdistanz von 3 wird auf das nächst größere Velfache von 4 aufgerundet, durch die Vektorisierung mit float4-Werten)
die Berechnung wird von jeweils 512 lokalen Workitems durchgeführt
daraus ergibt sich eine Blockgröße von:
block_size = 512 Workitems * 4 Werte = 2048 Werten

Die Anzahl der Werte die beim spaltenweisen Diamant-Tiling geladen/gespeichert werden müssen pro Zeile:

unoptimiert: alle vier Blöcke werden vollständig kopiert
→ 4 Blöcke * 2048 Werte = 8192 Werte

optimiert: zwei vollständige Blöcke und zwei Blöcke bis zur Zugriffsdistanz
→ 2 Blöcke * 2048 Werte + 2 Blöcke * 4 Werte = 4104 Werte

Im Vergleich dazu die Anzahl der Werte die beim zeilenweisen Diamant-Tiling geladen/gespeichert werden pro Zeile:
→ 4 Blöcke * 4 Werte = 16 Werte

→ Durch die Optimierung konnte der Bedarf an zu speichernden Werten fast halbiert werden.
Der Bedarf beim zeilenweisen Diamant-Tiling ist jedoch insbesondere für kleine Zugriffsdistanzen wesentlich geringer.

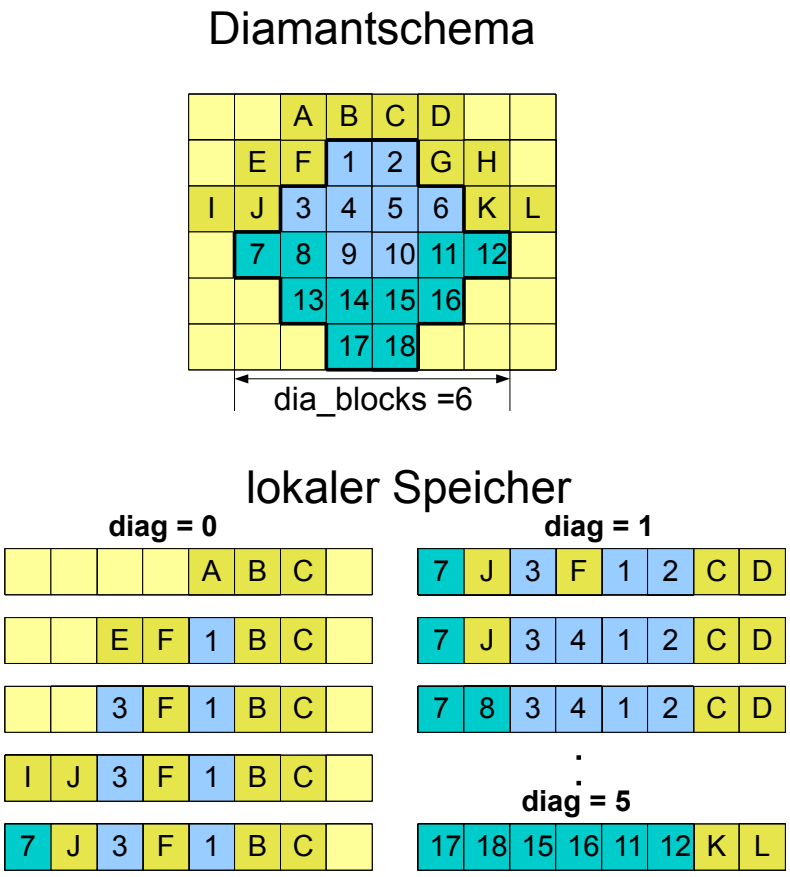


Abbildung 8.4: Spaltenweises Diamant-Tiling

8.3 Laufzeit des spaltenweisen Diamant-Tilings

Die Laufzeitmessungen für das spaltenweise Verfahren wurden alle auf der GTX 580 durchgeführt. Da das Verfahren für große Blöcke ausgelegt ist, ist zu erwarten, dass gerade das Brüsselator-Problem von der Optimierung profitiert. Für das Brüsselator-Problem können aber nur auf der GTX 580 größere Messungen durchgeführt werden, da bei der GTX 280 der kleinere lokale Speicher nur für geringe Systemgrößen Messungen zulässt.

In Abbildung 8.5 ist die normierte Laufzeit des String-Problems für die GTX 580 dargestellt. Es ist zu sehen, dass für kleine Systemgrößen keine Messungen für das spaltenbasierte Verfahren existieren, da durch die Vergrößerung der Blöcke nicht genügend Blöcke existieren und somit das Diamant-Verfahren nicht effektiv arbeiten kann. Ab einer Systemgröße von 100.000 wird der gesamte lokale Speicher benutzt und der maximale Speedup des Verfahrens kann erreicht werden. Die Laufzeit ist jedoch größer als beim zeilenweisen Verfahren. Dies kann mit der Vergrößerung der Blöcke begründet werden. Zum einen verringert dies die Zeitschritte pro Diamant und führt damit zu mehr globalen Barriers. Zum anderen erhöhen dies die Anzahl der Werte, die zwischen den Speichern kopiert werden müssen.

In Abbildung 8.6 ist die normierte Laufzeit für das Brüsselator-Problem für die GTX 580 dargestellt. Durch die größere Zugriffsdistanz des Brüsselator-Problems müssen die Blöcke ab einer bestimmten Problemgröße nicht weiter vergrößert werden (ab $n = 125.000$). Es ist zu sehen, dass das spaltenbasierte Verfahren auch hier kaum eine Laufzeitverbesserung erzielen kann. Das liegt vor allem daran, dass der Speedup durch die effektivere Speicherausnutzung und durch die höhere Anzahl von lokalen Barriers kompensiert wird.

8.4 Probleme und Erkenntnisse des spaltenbasierten Diamant-Tilings

Es hat sich herausgestellt, dass trotz der effektiveren Speicherausnutzung des spaltenbasierten Diamant-Tilings keine Laufzeitverbesserungen zum zeilenweisen Diamant-Tiling erzielt werden konnten. Dies liegt daran, dass zum einen die größeren Blöcke zu mehr Speicher- und Ladeoperationen führen und zum anderen mehr lokale Barriers benötigt werden und damit die Workitems häufiger unterbrochen werden. Das spaltenbasierte Verfahren bietet daher keine Laufzeitverbesserungen zum zeilenweisen Diamant-Verfahren. Das spaltenweise Verfahren bietet aber dahingehend einen Vorteil, dass sich dadurch Probleme mit großen Zugriffsdistanzen lösen lassen, für die ansonsten der Speicher beim zeilenweisen Verfahren nicht ausreichen würde.

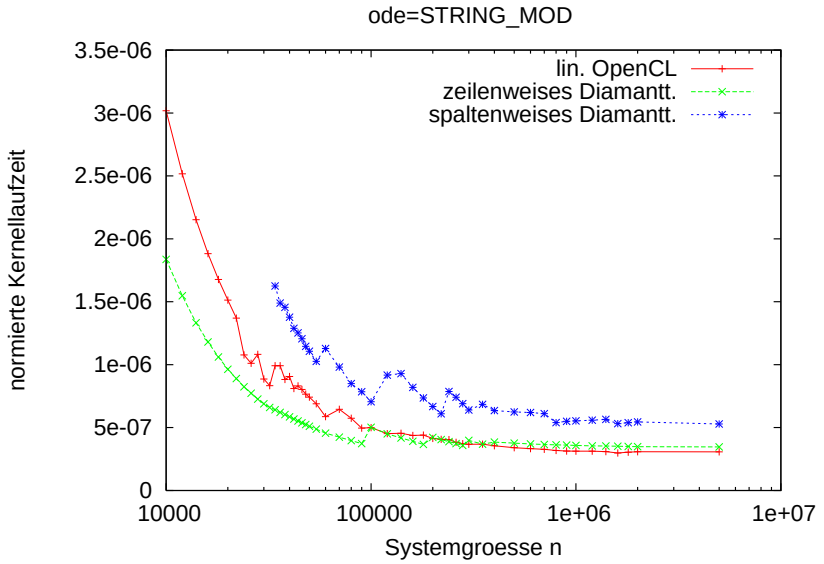


Abbildung 8.5: normierte Kernellaufzeit String

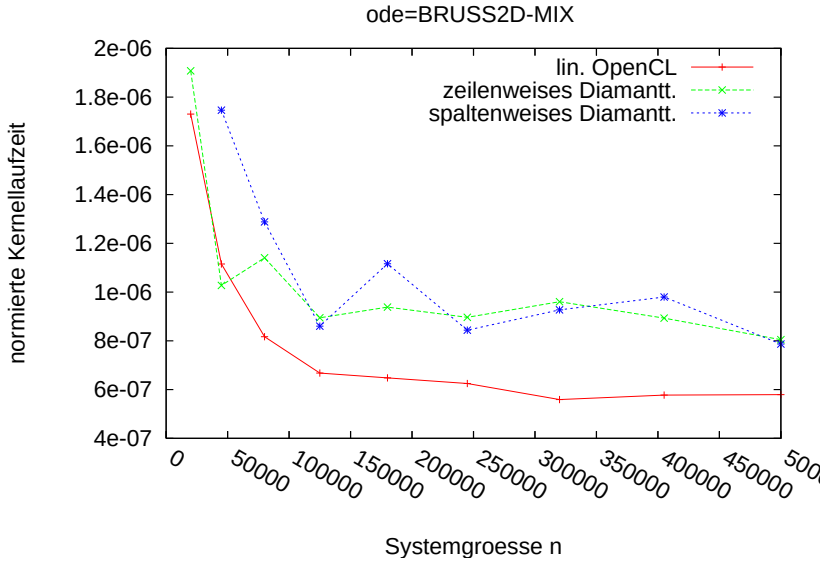


Abbildung 8.6: normierte Kernellaufzeit Bruss2d

9 Waben-Tiling

9.1 Motivation

Das Waben-Tiling ist eine Variation des zeilenweisen Diamant-Tilings. Durch die beiden bisher vorgestellten Versionen des Diamant-Tilings wurde deutlich, wie wichtig es ist, die Anzahl der Barriers zu minimieren und die Arbeit zwischen den Barriers zu maximieren. Beim spaltenweisen Verfahren gab es viele lokale Barriers und wenig Arbeit für jedes Workitem, was sich negativ auf die Laufzeit ausgewirkt hat. Beim zeilenweisen Verfahren war die Anzahl der Barriers gering, die Arbeit pro Zeile jedoch stark unterschiedlich. An der breitesten Stelle bei $step = 0$ (siehe Abbildung 7.7) ist die Arbeit maximal, an den oberen und unteren Rändern des Diamanten sind jedoch nur wenige Elemente pro Zeile zu berechnen. Gerade bei kleinen Zugriffsdistanzen, bei denen die Blöcke nur aus wenigen Elementen bestehen, werden an den Rändern nicht alle Workitems beschäftigt. Das Waben-Tiling versucht genau dies zu verbessern. Beim Diamant-Tiling berechnete jeder Diamant mit dia_blocks Blöcken genau $dia\_blocks - 1$ Zeitschritte. Beim Waben-Tiling wird eine feste Anzahl an Schritten pro Diamant vorgegeben (= *steps*) und damit werden die Enden jedes Diamanten abgeschnitten. Das Ergebnis ist, dass auch an den oberen und unteren Rändern noch genügend Arbeit für alle Workitems vorhanden ist.

Ein anderer Faktor der dazu beiträgt, dass die lineare OpenCL Version sehr gute Laufzeiten erzielt, ist, dass der lineare Zugriff auf große zusammenhängende Bereiche des globalen Speichers, wie er in Kapitel 2.2.4 beschrieben wurde, für die Grafikkarten optimal ist. Dieses Speicherzugriffsmuster kann aber nur in der linearen Version erreicht werden. In den Diamantversionen werden pro Zeile nur zwei eher kleine zusammenhängende Bereiche zwischen globalen und lokalen Speicher übertragen. Beim Waben-Tiling kann der Speicher an den Randzeilen wieder linear übertragen werden und das optimale Zugriffsmuster kann erreicht werden.

9.2 Aufbau der Waben

Abbildung 9.1 zeigt das prinzipielle Schema beim Waben-Tiling. Das Abarbeiten und Berechnen der Diamanten durch die Workgroups erfolgt wie bei der zeilenweisen Diamant-Version. Jede Workgroup berechnet ihren Diamanten zeilenweise.

Eine Besonderheit des Waben-Tilings ist, dass die Waben ineinander verschränkt sind. Da die verschiedenen Iterationen sich nun überlagern, kann Formel 6.1 nicht mehr zur Aufteilung der Waben auf die Workgroups benutzt werden. Die Größe

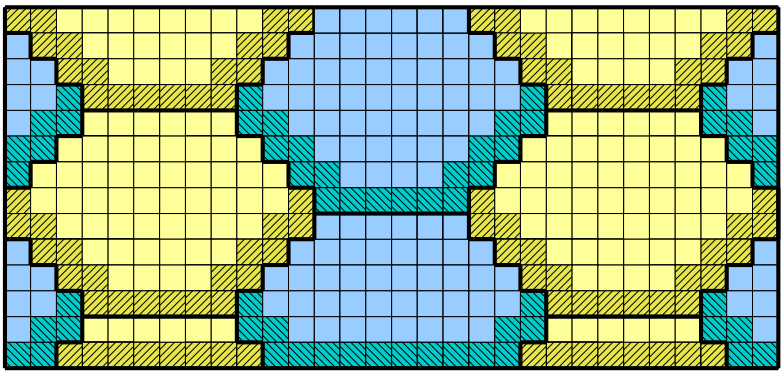


Abbildung 9.1: Waben-Tiling

eines Blockes wird wie beim zeilenweisen Diamant-Verfahren auf das nächste Vielfache von vier der Zugriffsdistanz gesetzt, damit die Blöcke so klein wie möglich werden, um unnötige Speicheroperationen zu vermeiden.

Die Gesamtzahl aller Blöcke (dia_blocks_sum) berechnet sich aus der Problemgröße geteilt durch die Größe eines Blockes:

$$dia\_blocks\_sum = n/block\_size \tag{9.1}$$

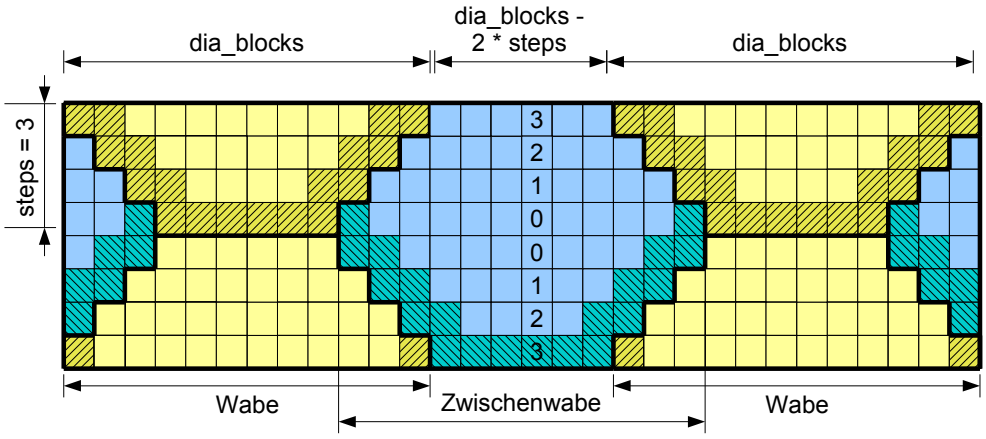


Abbildung 9.2: Wabe und Zwischenwaben mit steps = 3

Beim zeilenweisen Diamant-Tiling wurden alle Blöcke auf die Workgroups verteilt. Beim Waben Tiling müssen die Blöcke jedoch auf die Waben (gelbe Waben) und die *Zwischenwaben* (blaue Waben) aufgeteilt werden. Zur Bestimmung dieser Aufteilung betrachte man einen Zeitschritt, der $step = 0$ einer Wabe entspricht. In Abbildung refcomb2 entspricht dies $step = 3$ einer Zwischenwabe. Die Aufteilung

wird so vorgenommen, dass es in solch einem Zeitschritt bei *workgroups* Waben immer $workgroups - 1$ Zwischenwaben gibt. Die maximale Anzahl an Blöcken pro Zeitschritt der Waben wird wieder als *dia_blocks* bezeichnet. In einem Zeitschritt, in dem eine Wabe bei step 0 die maximale Anzahl an Blöcken pro Zeitschritt berechnet, haben die entsprechenden Zwischenwaben gerade die kleinste Anzahl an Blöcken (siehe Abbildung refcomb2). Die kleinste Anzahl der Blöcke ist abhängig von dem Zeitschritt nach dem eine Wabe abgeschnitten wird. In der Abbildung wird jede Wabe und Zwischenwabe nach dem Zeitschritt 3 (= steps) nach oben und unten abgeschnitten. Die Anzahl der Blöcke bei der maximalen Anzahl von Zeitschritten ist $dia\_blocks - 2 * steps$.

Die Gesamtzahl aller Blöcke muss auf die *workgroups* Waben und die *workgroups* −1 Zwischenwaben verteilt werden:

$$\begin{aligned} dia\_blocks\_sum &= \text{Blöcke der Waben} + \text{Blöcke der Zwischenwaben} \\ dia\_blocks\_sum &= workgroups * dia\_blocks \quad + \\ &\quad (workgroups - 1) * (dia\_blocks - 2 * steps) \end{aligned}$$

Durch Umformung lässt sich daraus die Anzahl der Blöcke pro Diamant für eine bestimmte Schrittweite (*steps*) berechnen:

$$\begin{aligned} dia\_blocks\_sum = \quad & 2 * workgroups * dia\_blocks + \\ & (workgroups - 1) * (-2 * steps) - dia\_blocks \end{aligned}$$

$$\begin{aligned} dia\_blocks\_sum + 2 * steps * (workgroups - 1) = \\ (2 * workgroups - 1) * dia\_blocks \end{aligned}$$

$$dia\_blocks = \frac{dia\_blocks\_sum + 2 * steps * (workgroups - 1)}{2 * workgroups - 1}$$

9.3 Ermitteln der optimalen Wabengröße

Wird die Größe der Waben minimiert ($steps = 0$), so wird ein ähnliches Verhalten wie beim linearen OpenCL-Verfahren erzielt. Es wird dabei das optimale Zugriffsmuster für alle Werte vom globalen Speicher erzielt, da alle Werte auf einmal geladen werden. Jedoch werden die Daten im lokalen Speicher nicht wiederverwendet. Werden die Waben vergrößert, erhält man einen positiven Effekt, indem die Werte im lokalen Speicher nun wiederverwendet werden. Allerdings gibt es auch einen negativen Effekt, da der Zugriff auf die globalen Daten nicht mehr immer dem optimalen Muster folgt. Werte am Rand müssen in jedem Schritt wieder nachgeladen werden, was zu einem nicht optimalen Zugriffsverhalten auf den globalen Speicher führt. Wird die Wabengröße maximiert, ergibt sich daraus das Verhalten des zeilenweisen Diamant-Tilings. Durch Variation der Größe steps ist es möglich

einen optimalen Wert zu erreichen, der für die jeweilige Hardware den Kompromiss zwischen einer Wiederverwendung des lokalen Speichers und den optimierten Speicherzugriffen auf den globalen Speicher findet.

Nachfolgend sind die Laufzeiten des String-Problems mit einer Systemgröße von 2.000.000 dargestellt. In Abbildung 9.3 ist die Laufzeit für die GTX 280 mit ca. 500 Blöcken pro Diamant in Abhängigkeit von der maximalen Schrittweite steps aufgetragen. Erwartungsgemäß wird bei kleinen Schrittweiten mit wachsender Schrittweite die Laufzeit geringer, da eine Wiederverwendung durch den lokalen Speicher erzielt wird. Ab einer Schrittweite von ca. 50 steigt die Laufzeit wieder an, da der unoptimierte Zugriff auf den globalen Speicher signifikant wird.

Bei der minimalen Laufzeit ist das Verhältnis der Wiederverwendung des lokalen Speichers zum optimierten Speicherzugriff auf den globalen Speicher optimal. Die Wiederverwendung wird durch die max. Schrittweite charakterisiert und die Menge der Daten, die zwischen den Speichern transferiert werden muss, wird durch die Anzahl der Blöcke charakterisiert. Beim Minimum der Laufzeit beträgt das Verhältnis der Blöcke pro Diamant zur Schrittweite: $\sim 500/40 =\sim 12.5$.

In Grafik 9.4 ist die Abhängigkeit von der maximalen Schrittweite auf der GTX 580 dargestellt. Durch ihren größeren lokalen Speicher besteht hier jeder Diamant aus ca. 1500 Blöcken. Es ist zu sehen, dass das Minimum bei etwa 100 Schritten erreicht ist. Dies ist höher als bei der GTX 280, jedoch im Verhältnis zur Blockgröße ($1500/100 =\sim 15$) stellt sich ein Wert ähnlicher Größenordnung ein. Das bei manchen Schrittweiten auftretende plötzliche Ansteigen der Laufzeiten ist dadurch zu erklären, dass durch Variation der maximalen Schrittweite auch die Anzahl an Blöcken und damit die Anzahl an Diamanten verändert wird und dies zu Laufzeitsprüngen führen kann.

Als Beispiel des betrachtet man den Laufzeitsprung in Abbildung 9.4:
bei einer maximalen Schrittweite von 100 auf 120 kommt es zu einem Laufzeitsprung:

Die GTX 580 besitzt 16 Recheneinheiten
In der folgenden Tabelle sind für die max. Schrittweiten die Anzahl der Blöcke pro Diamant angegeben, die Gesamtanzahl alle Diamanten und das Verhältnis wenn alle Diamanten auf 16 Recheneinheiten aufgeteilt werden:

Schrittweite	Blöcke	Diamanten	#Diamanten / #16 Rechene.
100	1526	176	11
110	1528	177	11.06
120	1530	178	11.1

Durch die Änderung der maximalen Schrittweite ändert sich auch die Anzahl der Blöcke pro Diamant und die Anzahl der Diamanten. Bei 100 ist die Anzahl gerade ein Vielfaches der Recheneinheiten, es werden also genau 11 Diamanten von jeder Workgroup berechnet. Ab Schrittweite 110 muss eine Recheneinheit 12 Diamanten berechnen, was einen Sprung in der Laufzeit verursacht.

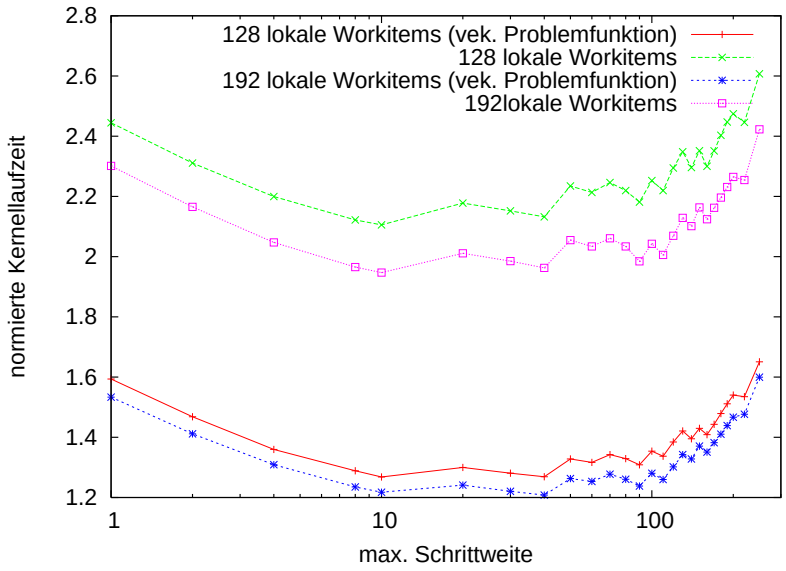


Abbildung 9.3: Laufzeiten für verschiedene Maximalschrittweiten GTX 280

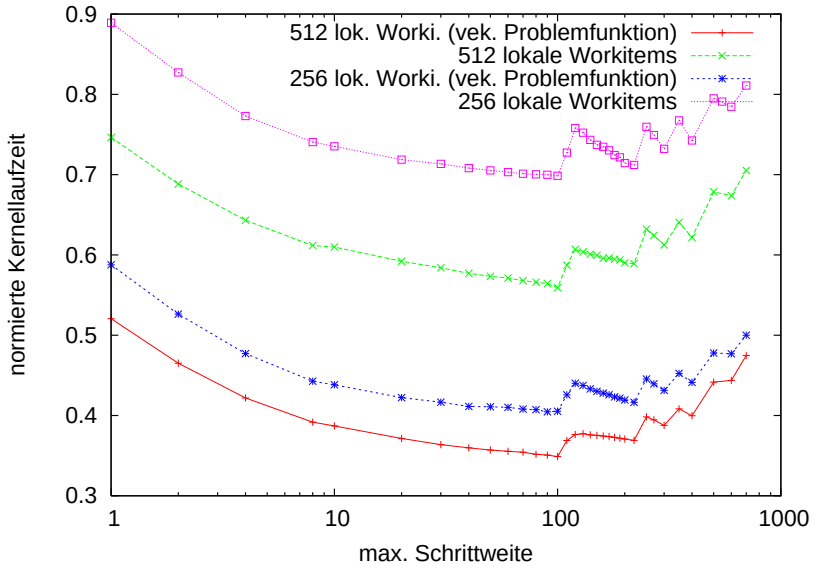


Abbildung 9.4: Laufzeiten für verschiedene Maximalschrittweiten GTX 580

9.4 Laufzeit und Speedup des Waben-Tilings

In den Abbildungen 9.5 und 9.6 sind die normierten Laufzeiten des Waben-Tilings im Vergleich zum zeilenweisen Diamant- und linearen OpenCL-Verfahren aufgetragen. In Abbildung 9.5 sind die Implementierungen ohne vektorisierte Problemfunktion, in Abbildung 9.6 die mit vektorisierter Problemfunktion dargestellt. Bei kleinen Systemgrößen verhält sich das Diamant-Tiling besser als das Waben-Tiling, da hier jede Workgroup kleine Diamanten berechnet und es nicht sinnvoll ist, die Diamanten abzuschneiden. Bei größeren Problemen ist das Waben-Tiling jedoch deutlich schneller als das lineare Verfahren und das Diamant-Tiling.

In Abbildung 9.7 und 9.8 sind die entsprechenden Messungen auf der GTX 580 dargestellt. Es ergibt sich ein ähnliches Verhalten wie bei der GTX 280. Das lineare OpenCL-Verfahren profitiert stark vom Cache, durch das es nur auf die Daten des globalen Speichers zugreift. Der limitierende Faktor ist dabei die Speicherbandbreite des globalen Speichers, dadurch kann auch durch die Vektorisierung kein weiterer Speedup erzielt werden. Das Diamant- und Waben-Tiling profitiert weniger vom Cache und kann daher ohne Vektorisierung keinen weiteren Laufzeitverbesserungen zum linearen OpenCL-Verfahren erlangen. Durch die Vektorisierung wird der Zugriff auf die Daten im lokalen Speicher verbessert und mit dem Diamant- und Waben-Tiling können bessere Laufzeiten erzielt werden als mit dem linearen Verfahren.

9.5 Probleme und Erkenntnisse des Waben-Tilings

Das Waben-Tiling ist eine Verbesserung des zeilenweisen Diamant-Verfahrens. Gerade auf GPUs ohne L2 Cache erzielt das Waben-Tiling wesentlich bessere Ergebnisse als das lineare Verfahren. Aber auch auf Grafikkarten mit L2 Cache erzielt das Waben-Tiling meist bessere Laufzeiten als die linearen Verfahren. Das Waben-Tiling ist daher eine gute Optimierung des zeilenweisen Diamant-Verfahrens, da es die Stärken des Diamant-Tiling mit denen des linearen OpenCL-Verfahrens kombiniert.

Bei großen Zugriffsdistanzen oder kleineren lokalen Speichern, wo das spaltenweise Diamant-Tiling seine Stärken besitzt, wäre es also auch ratsam eine Kombination des spaltenweisen Verfahrens mit dem Prinzip des Waben-Tilings zu verwenden.

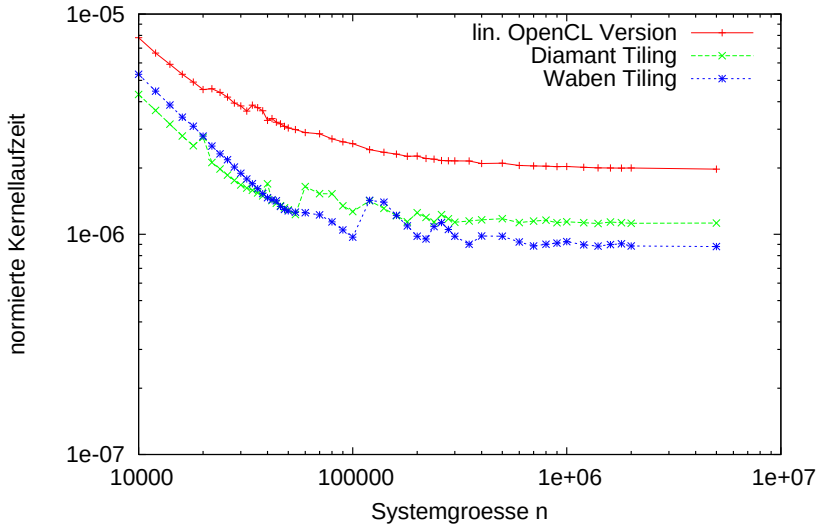


Abbildung 9.5: normierte Kernellaufzeit GTX 280 (ohne Vektorisierung)

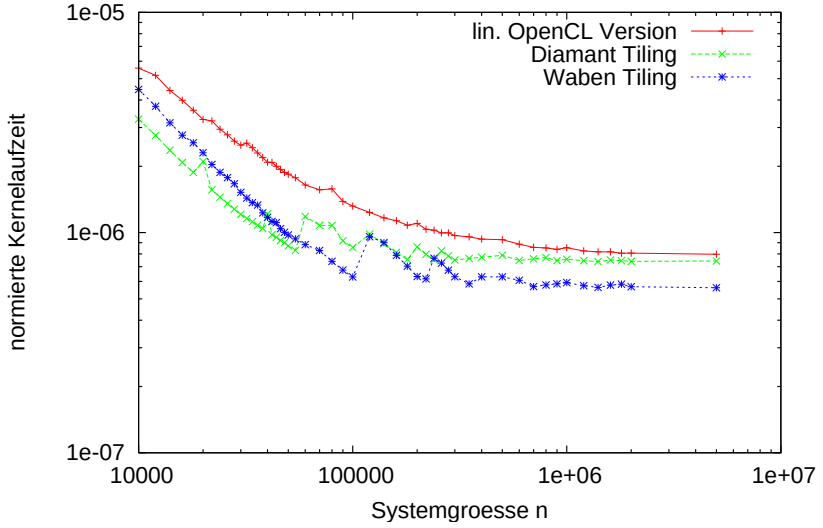


Abbildung 9.6: normierte Kernellaufzeit GTX 280 (vektorisierte Problemfunktion)

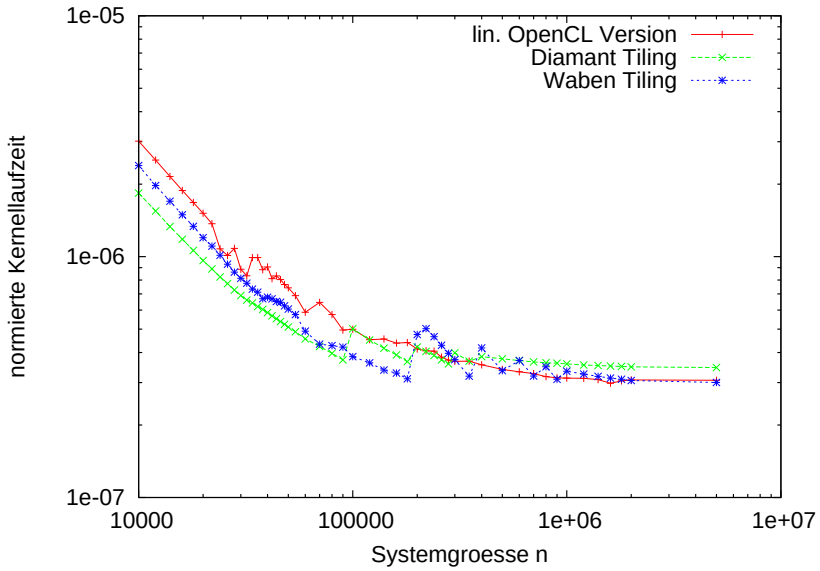


Abbildung 9.7: normierte Kernellaufzeit GTX 580 (ohne Vektorisierung)

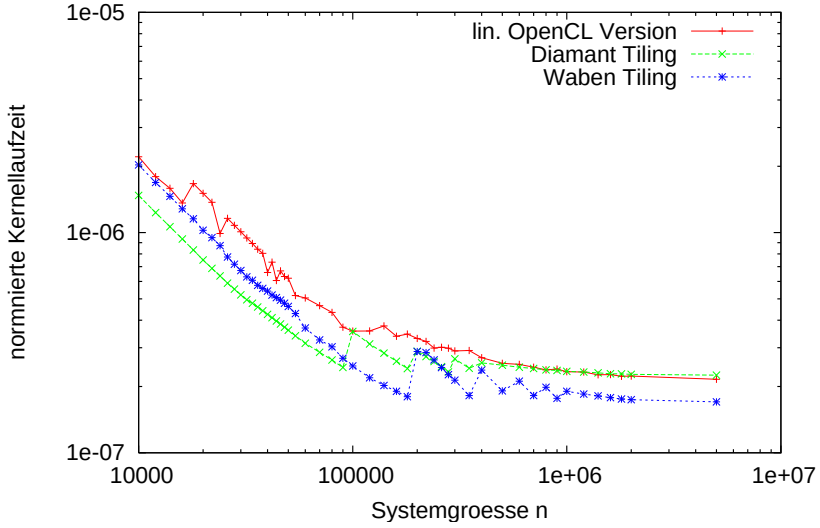


Abbildung 9.8: normierte Kernellaufzeit GTX 580 (vektorisierte Problemfunktion)

10 Evaluierung

10.1 Evaluierung zum Einsatz von Grafikprozessoren

Abschließend soll die Frage betrachtet werden, inwiefern sich der Einsatz von Grafikkarten zum Lösen von Differentialgleichungssystemen lohnt. Dabei wurden das lineare OpenCL-Verfahren und das Diamant/Waben-Tiling mit der Pthreads-Version auf einem Mac Pro verglichen.

	Pthreads/GTX280	GTX580	Mac Pro
CPU:			
Prozessor	2x AMD Opteron 270	Intel E5530	2x Intel E5520
Anzahl Kerne	4	4	8
Taktung	2.0 GHz	2.4 GHz	2.26 GHz
GPU:			
Grafikkarte	GTX 280	GTX 580	/
Recheneinheiten	30	16	
Anzahl Kerne	240	512	
globaler Speicher	1023 MB	1536 MB	
lokaler Speicher	16 kB	48 kB	
L2 Cache	/	768 kB	

In den Tabellen 10.1 und 10.2 ist zu sehen, dass die beiden GPU-Implementierungen wesentlich schneller sind als die Pthreads-Version. Bei Problemen mit kleinen Zugriffsdistanzen ist ein Speedup um das bis zu 129fache festzustellen Bei größeren Zugriffsdistanzen wird ein Speedup um das 73fache erreicht. Die GTX 580 erzielt dabei immer mindestens einen Speedup, der um das 12fache größer ist als der von den beiden Intel Prozessoren. Mit der Grafikkarte GTX 280 wurde ein um mindestens das 2,5fache größere Speedup erreicht. Werden nun die aktuellen Anschaffungskosten ([2] Intel E5520 400€, NVIDIA GTX 580 450€, NVIDIA GTX 280 200€) für die Hardware verglichen, wird deutlich, dass die GPUs weniger kosten dafür aber wesentlich leistungsfähiger sind als die beiden CPUs. Gerade bei Aufgaben, die ein hohes Parallelisierungspotential bieten, wie das Lösen von Differentialgleichungen, ist der Einsatz von Grafikkarten Kosten-effektiv.

Implementierung	Gesamtlaufzeit	Speedup
CPU		
PThreads (1 Thread)	2679sec	1
PThreads (16 Threads)	357sec	7,5
GTX 280		
lin. OpenCL Verfahren	84,9sec	31,5
Waben Tiling	61,3sec	43,7
GTX 580		
lin. OpenCL Verfahren	26,7sec	100
Waben Tiling	20,75sec	129

Abbildung 10.1: Problem Stringmit einer Systemgröße von n = 100.000.000

Implementierung	Gesamtlaufzeit	Kernellaufzeit	Speedup (Kernel)
CPU			
PThreads (1 Thread)	28,33sec	28,33sec	1
PThreads (16 Threads)	4,74sec	4,74sec	6
GTX 280			
lin. OpenCL Verfahren	2.64	1.83	15,5
GTX 580			
lin. OpenCL Verfahren	1,14sec	0,39sec	73
Diamant Tiling	1,36sec	0,47sec	60

Abbildung 10.2: Problem Brüsselator mit einer Systemgröße von n = 500.000

10.2 Evaluierung zum Optimieren von Algorithmen unter Grafikprozessoren

Im Folgenden sollen die Optimierungstechniken aus Kapitel 2.2 hinsichtlich ihres Aufwands gegenüber ihrem Performancegewinn evaluiert und bewertet werden.

Lese- und Schreibzugriffe auf den globalen Speicher

Gerade bei teuren Speicherzugriffen ist es wichtig, diese effizient durchzuführen. Das Speicherzugriffsmuster aus Punkt 2.2.4 hat sich als sehr effektiv erwiesen. Wie in Kapitel 5.2 ausgeführt wurde, ist das Verändern des Speicherzugriffsmusters für alle Schleifenstrukturen, die auf den globalen Speicher zugreifen, einfach durchzuführen und erzielt einen hohen Speedup.

Loop Unrolling und Vektorisierung

Da Grafikkarten üblicherweise für Vierervektoren von Floats optimiert sind, ist es von Vorteil diese nach Möglichkeit im Code zu benutzen, um das ideale Zugriffsmuster und damit die 128-bit Wortbreite zu erreichen. Jedoch kann es sehr

aufwendig werden, einen bestehenden Code zu vektorisieren, wie beim Parallelisieren der Problemfunktionen ersichtlich war.

Bestimmen der Anzahl der Workitems und Workgroups

Die optimale Anzahl der Workgroups ist meist durch die Anzahl der Recheneinheiten bestimmt und lässt sich zur Programmausführungszeit per API-Befehl abfragen. Dies ist notwendig um eine Lastbalancierung der Workgroups auf die Recheneinheiten zu gewährleisten.

Die optimale Anzahl an lokalen Workitems ist eine experimentelle Größe, bei der darauf geachtet werden sollte, dass die Anzahl immer wesentlich größer ist als die Anzahl der Kerne pro Multiprozessor. Ein guter Standardwert ist die maximale Anzahl an Workitems für den jeweiligen Kernel. Dies ermöglicht dem Scheduler auf der GPU möglichst viele Latenzen durch Speicherzugriffe zu verstecken.

Lokaler Speicher

Die Verwendung des lokalen Speichers ist für Daten, die eine hohe Wiederverwendung besitzen, sinnvoll. Dabei ist es aber erforderlich, die Daten und ihre Abarbeitung einzelnen Recheneinheiten zuzuweisen. Dies ist nicht immer machbar oder es ist sehr aufwendig, da der Algorithmus teilweise tiefgreifend verändert werden muss (siehe Diamant-/Waben-Tiling).

Barriers

Der Einsatz von lokalen Barriers ist eng verknüpft mit der Benutzung des lokalen Speichers, da auch hier die Aufteilung der Daten auf einzelne Recheneinheiten erforderlich ist. Dabei ist es vorteilhaft, so viele globale Barriers wie möglich durch lokale Barriers zu ersetzen. Der Einsatz von lokalen Barriers führt dazu, dass weniger Kernel gestartet werden müssen, welches eine Laufzeitersparnis darstellt. Dabei sollte jedoch darauf geachtet werden, die Anzahl der Barriers nicht zu erhöhen, denn dadurch wird die Arbeit zwischen den Barriers minimiert und die Laufzeit kann sich erhöhen (wie beim spaltenweisen Diamant-Tiling gesehen).

11 Zusammenfassung

Das Ziel der Arbeit war es, das Euler-Verfahren auf Grafikprozessoren zu portieren und dafür zu optimieren. Dies ist zum einem mit dem linearen OpenCL-Verfahren und zum anderen mit den Diamant-/Waben-Tilings gelungen. Das lineare OpenCL-Verfahren ist zwar ein sehr einfaches Verfahren, welches nicht alle Optimierungen der GPU ausnutzt, aber es werden gute Laufzeiten erzielt, die wesentlich schneller sind als vergleichbare CPU-Versionen. Algorithmen die schon für die CPU parallelisiert wurden, zum Beispiel mit Pthreads oder OpenMP, lassen sich leicht für die GPU portieren. Auf der GPU können dann schon einige wichtige aber einfache Optimierungen zu sehr guten Laufzeiten führen, die sich nur schwer mit der CPU erreichen lassen.

Das Diamant-/Waben- Tiling ist wesentlich komplexer, führt aber auch meist zu noch besseren Ergebnissen als das lineare OpenCL-Verfahren, da es mehr Vorteile der Grafikprozessoren ausnutzen kann. Bei sehr zeitintensiven Berechnungen, bei denen es wichtig ist, die Laufzeit zu minimieren, wäre es also von Vorteil den bestehenden Code derart anzupassen, dass möglichst alle Eigenschaften und Besonderheiten der GPUs berücksichtigt werden. Dabei muss beachtet werden, dass sich einige Optimierungen gegensätzlich verhalten können. Wie beim Waben-Tiling gesehen, musste ein Kompromiss zwischen einem optimalen Zugriffsmuster und der Wiederverwendung des lokalen Speichers gefunden werden.

Der Einsatz von Grafikprozessoren zum Lösen von Differentialgleichungen ist bei Problemen, die lange zum Berechnen benötigen, zu empfehlen. Das Initialisieren der GPU und das Übersetzen des Kernels benötigen allerdings Zeit, die sich über die Laufzeit der Berechnung amortisieren muss.

OpenCL hat sich im Laufe der Arbeit als eine geeingete Programmierschnittstelle für Grafikprozessoren herausgestellt, bei der jedoch anzumerken ist, dass sie speziell für Grafikprozessoren entwickelt wurde. OpenCL-Kernel lassen sich zwar auch auf CPUs ausführen, jedoch sind viele Besonderheiten auf Prozessoren nicht vorhanden und führen dort nicht immer zu einem Laufzeitgewinn, wie zum Beispiel der Einsatz lokaler Barriers und lokaler Speicher. Optimierungen wie zum Beispiel das Speicherzugriffsmuster, verhalten sich auf CPU und GPU teilweise nachteilig, sodass spezielle Kernel für GPU und CPU entwickelt werden müssen, um den optimalen Speedup auf allen Devices zu erhalten.

Warum findet die GPU Programmierung trotz enormer Geschwindigkeitssteigerungen gegenüber Prozessoren noch keine breite Verwendung in kommerziellen Programmen? Ein Grund kann darin gesehen werden, dass bisher keine standardisierte Schnittstelle für alle Grafikprozessoren zur Verfügung stand. Jeder Grafikkartenhersteller bevorzugte seine eigene API (NVIDIA CUDA, ATI Stream). Mit

OpenCL steht eine geeignete Schnittstelle für alle Grafikkarten zur Verfügung, die den Einsatz von GPUs für allgemeine Berechnungen weiter verbreiten kann.

Die jüngste Vergangenheit hat gezeigt, dass die Steigerung der Geschwindigkeit von Prozessoren zum Großteil durch die Erhöhung der Anzahl der Kerne und nicht mehr so sehr durch Erhöhung des Prozessortaktes erreicht wird. Deshalb ist zu erwarten, dass der Einsatz von Grafikkarten zur Berechnung von zeitintensiven Prozessen in Zukunft zunimmt. Einen wichtigen Faktor werden dabei PC Spiele darstellen, da sie den Systemen durch immer neue Innovationen wie verbesserte Physik Modellierung und KI immer mehr Leistung abverlangen. Beispiele, bei dem der Geschwindigkeitsgewinn von GPUs für nicht Grafik-bezogene Aufgaben benutzt wird, sind Spiele wie "Batman: Arkham City" und "Batman: Arkham Asylum" ([5],[4]), die die Physik des Spiels durch GPU-Physix ([11]) berechnen lassen. Aber auch Aufgaben des wissenschaftlichen Rechnens, nutzen immer mehr Grafikprozessoren zum Berechnen. Daher beziehen schon drei der fünf schnellsten Supercomputer ([1]) einen Großteil ihrer Rechenkapazitäten aus Grafikkarten.

Sollten sich in der Industrie einheitliche Standards bei der GPU-Programmierung durchsetzen, mit APIs, die auch über mehrere Jahre hinweg beständig sind, so ist zu erwarten, dass die GPU-Programmierung in Zukunft immer breitere Verwendung findet.

Literaturverzeichnis

[1] TOP500 Supercomputer Site. Website: http://www.top500.org, besucht am 13.03.2012.

[2] Amazon. Website: http://www.amazon.de, besucht am 2.12.2011.

[3] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving ordinary differential equations: Nonstiff problems.* Springer series in computational mathematics. Springer, 1993.

[4] PC Games Hardware. Website: http://www.pcgameshardware.de/aid,850363/Batman-Arkham-City-Trailer-zeigt-GPU-Physx-Update-Bildvergleich-mit-und-ohne-Effekte-in-hoher-Aufloesung/Action-Spiel/News/, besucht am 14.12.2011.

[5] Eidos Interacrive. Website: urlhttp://www.batmanarkhamcity.org/, besucht am 14.12.2011.

[6] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.1*, 2011.

[7] Matthias Korch. *Effiziente Implementierung eingebetteter Runge-Kutta-Verfahren durch Ausnutzung der Speicherzugriffslokalität.* Doctoral thesis, University of Bayreuth, December 2006.

[8] Adam Lake. *Game Programming Gems 8.* Course Technology, 2010.

[9] Microsoft. DirectCompute. Website: http://www.microsoftpdc.com/2009/P09-16, besucht am 14.12.2011.

[10] NVIDIA. Cuda. Website: http://www.nvidia.de/object/cuda_home_new_de.html, besucht am 14.12.2011.

[11] NVIDIA. GPU-Physix. Website: http://www.geforce.com/Hardware/Technologies/physx, besucht am 14.12.2011.

[12] Daniel A. Orozco and Guang R. Gao. Mapping the FDTD Application to Many-Core Chip Architectures. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 309–316, Washington, DC, USA, 2009. IEEE Computer Society.

[13] K. Strehmel and R. Weiner. *Numerik gewöhnlicher Differentialgleichungen.* Teubner Studienbücher. B.G. Teubner, 1995.

[14] Kane S. Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *IEEE Trans. Antennas and Propagation*, pages 302–307, 1966.

Zusammenfassung

Die hier vorliegende Arbeit beschäftigt sich damit, das explizite Euler-Verfahren auf Grafikprozessoren zu optimieren. Dabei werden die Speicherhierarchien, lokale Datenwiederverwendung, Ausnutzung der Speicherbandbreite der GPU und die Synchronisierung zwischen Host und Device genauer untersucht. Dabei werden zwei Implementierungen näher betrachtet, das Diamant-Tiling und das lineare Verfahren, da sie sich gut eignen um die Optimierungen genauer zu untersuchen. Es stellt sich dabei heraus, dass Optimierungen wie die lokale Datenwiederverwendung und der optimale Zugriff auf den Speicher sich gegensätzlich verhalten. Ein Mischverfahren (das Waben-Tiling), dass dabei die Vorteile des linearen Verfahrens und des Diamant-Tilings vereint, führt daher zu den besten Laufzeiten.

Abstract

The aim of the thesis is to investigate the Euler method for GPUs. The goal is to analyze the memory hierarchies, local data-reuse, memory bandwidth of the GPU and the synchronization of the host and the device. Two implemenations are considered closer the diamond tiling and the linear methode, since they are well suited to investigate further improvements. It turns out that the optimizations as local data-reuse and optimum access to the memory bandwidth behave contrary. In the end a combined system (the honeycomb tiling) that combines the advantages of the linear method and the diamond tiling leads to the best results.