Fakultät für Informatik und Mathematik
Universität Passau

Effiziente Berechnung der FWT auf Grafikkarten

Bachelorarbeit

Richard Maltan
Matrikel-Nummer 58941

Betreuer Prof. Dr. Tomas Sauer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	application programming interface
CPU	Central processing unit
CU	Computing Units
CUDA	compute unified device architecture
FFT	Fast-Fourier-Transform
FWT	Fast Wavelet Transform
GPU	Graphics processing unit
HLSL	High Level Shading Language
PE	Processing Elements
PTX	Parallel Thread Execution
SIMT	Single Instruction, Multiple Thread
SM	Streaming Multiprocessors
SP	Streaming Processor

1 Einleitung

Eine Möglichkeit, Aufgaben in der Informatik zu beschleunigen, ist es, diese in kleinere Arbeitspakete aufzuteilen und parallel abzuarbeiten. Dies ist zwar nicht für alle Arten von Berechnungen sinnvoll, kann aber, wenn es möglich ist, einen großen Teil der Rechenzeit sparen. Diesen Ansatz macht man sich beim GPU-Computing zunutze.
GPUs besitzen aufgrund ihrer Architektur gegenüber klassischen CPUs Vorteile beim Ausführen einer großen Anzahl Threads. Waren diese Anfangs noch hauptsächlich für die Berechnung der grafischen Ausgabe zuständig, hat sich spätestens mit der Zusammenlegung von Pixel- und Vertex-Shadern zu voll programmierbaren Unfied Shadern ihr Anwendungsgebiet auf klassische Berechnungen ausgeweitet.
An den Anfängen des GPU-Computings waren solche APIs (application programming interface) noch mehr auf die Verwendung der GPU als grafische Beschleuniger zugeschnitten, sodass man nicht-grafische Berechnungen mit grafischen Termen ausdrücken musste, als Beispiel sei hier OpenGL genannt. Aber in den letzten Jahren entstanden auch andere APIs, die die Nutzung GPU für nicht-grafische Berechnungen in den Vordergrund stellten. Beispiele für solche APIs sind OpenCL, welche nicht nur für GPUs verwendet werden kann, und CUDA, ein Programmiermodell für NVIDIA GPUs.
Das Ziel dieser Arbeit ist es, die Programmierung mit CUDA näher zubringen. Dazu werden wir unter anderem anhand des Beispiels der Schnellen Wavelettransformation (Fast Wavelet Transform, FWT) die Grundlagen der CUDA C Runtime API betrachten. Darunter fallen Modifier für Funktionen, zur Kennzeichnung der Aufgabe und Ausführungsplattform, die Erstellung und der Aufruf einer parallelisierten Funktion, sowie die Verwendung der verschiedenen Speichertypen, die das CUDA Modell bereitstellt. Des weiteren werden wir sehen, wie man Informationen über die ausführende GPU zur Laufzeit auslesen und diese im Programm nutzen kann, um diese optimal auszulasten.
Abschließend vergleichen wir CUDA C mit OpenCL, einer weiteren API zur Implementierung von auf der GPU lauffähigen Programmen.

2 CUDA Einführung

Die *compute unified device architecture* (CUDA) ist ein von NVIDIA entwickeltes Programmiermodell, das es erlaubt, die Architektur von CUDA-fähigen GPUs für die parallele Bearbeitung von Daten auszunutzen. Erstmals eingesetzt wurde CUDA 2006 in der auf der Tesla-Architektur basierenden Geforce 8800 GTX[15, 23]. Die Tesla-Architektur führte sogenannte *Unfied Shader* ein, die aus der Zusammenlegung und Erweiterung der bisher verwendeten Vertex- und Pixel-Shader zu vollwertig programmierbaren Prozessoreinheiten entstanden [13]. Dies und die Tatsache, dass eine auf C basierende Sprache zur Programmierung verwendet werden konnte, die, anders als grafikorientierte Sprachen wie OpenGL oder HLSL (High Level Shading Language), kein Umdenken in grafische Terme erforderte, führten dazu, dass sich die Verwendung von GPUs für parallele Berechnungen stark vereinfachte.

Eine CUDA-fähige GPU ist im allgemeinen grob in folgende Komponenten aufgeteilt:

- Ein Array aus *Streaming Multiprocessors* (SMs), von denen jeweils mehrere zu einem Block zusammengefasst werden. Jeder SM besteht aus mehreren *Streaming Processors* (SPs).
- Einen Onboard-Speicher, der als Global Memory bezeichnet wird.

Die Anzahl der SMs ist vom Device abhängig, die Anzahl der SPs pro SM von der Unterstützen *Compute Capability*, also dem Feature-Set, das eine GPU unterstützt. Eine **Geforce GTX 460m** (Fermi-Architektur) mit compute capability 2.1 besitzt beispielsweise 4 SMs mit je 48 SPs.

Wie man die Compute Capability zur Laufzeit auslesen kann wird in Abschnitt 5.1 genauer erklärt.

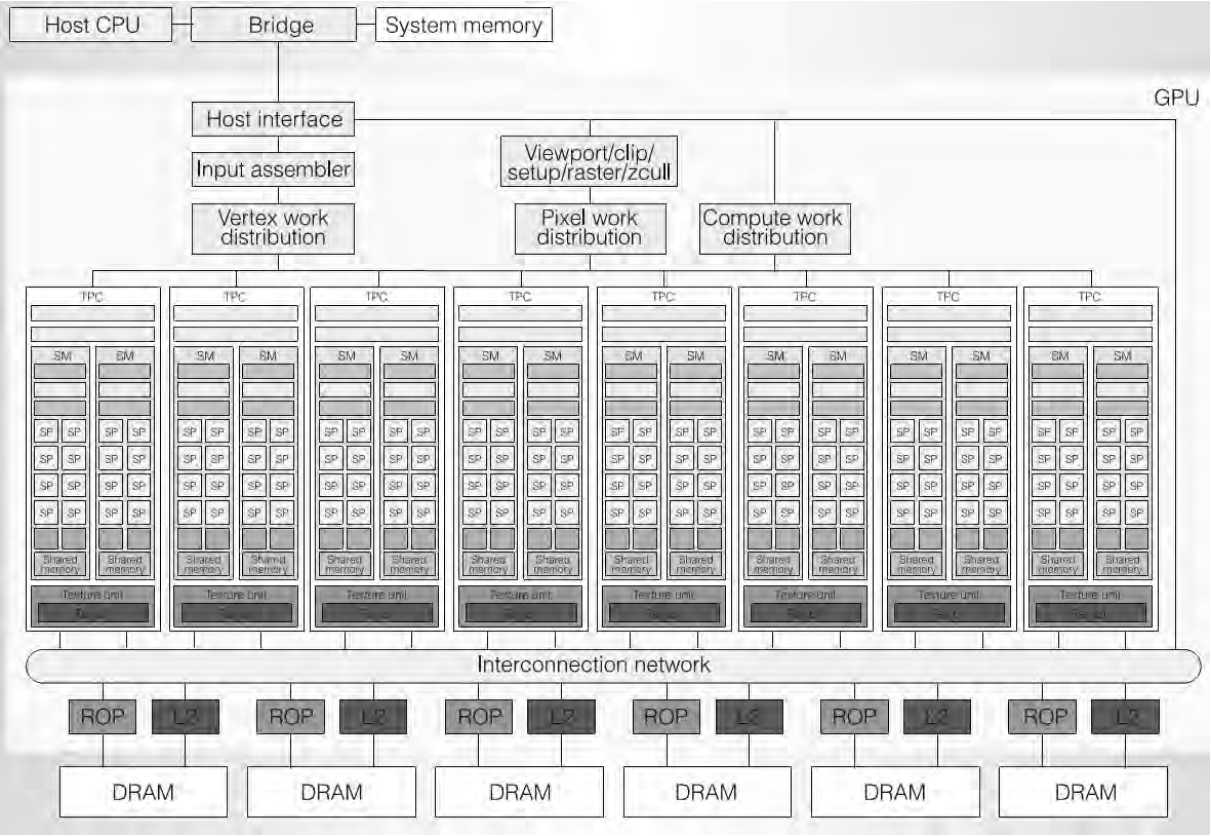

Abbildung 2.1: Aufbau einer NVIDIA GPU (Quelle: [15])

2.1 CUDA-Modell

Der Vorteil einer GPU gegenüber der CPU ist die Möglichkeit bei geeigneten Anwendungen die Berechnungen mithilfe von Threads zu parallelisieren. CUDA verwendet dafür ein *Single Instruction, Multiple Thread*-Modell (SIMT). SIMT besitzt Ähnlichkeiten zum *Single Instruction, Multiple Data*-Modell, mit dem Unterschied, dass eine einzelne Operation nicht nur auf mehrere Daten angewandt wird, sondern zusätzlich auf mehrere Threads, die ein SM bearbeitet.

2.1.1 Parallele Verarbeitung

Um im CUDA-Modell eine Funktion als parallelisiert zu kennzeichnen wird sie als Kernel deklariert (siehe hierzu Abschnitt 4.1). Wird diese Funktion innerhalb des Codes

aufgerufen, muss angegeben werden, in wie vielen Threads sie ausgeführt werden soll. Die Gesamtheit der so gestarteten Threads wird als `Grid` bezeichnet, welches entweder als eindimensionales Gebilde erstellt werden oder auch mehrdimensional sein kann [3]. Ein solches Grid besteht aus sogenannten Thread-Blöcken oder nur Blöcken, die ihrerseits wieder ein- oder mehrdimensional aus Threads bestehen.

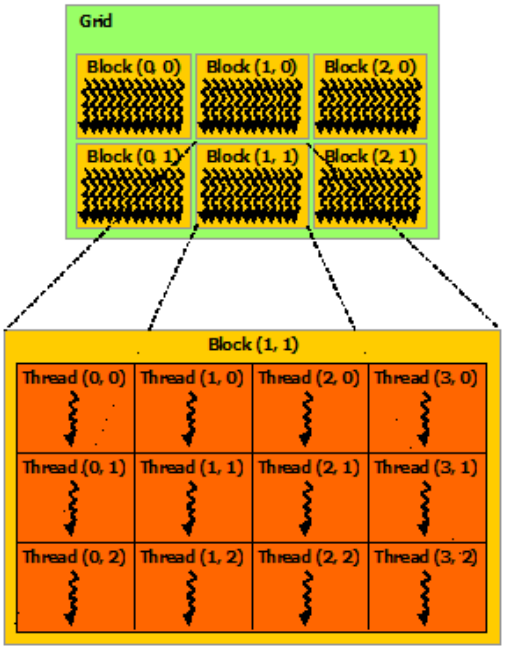

Abbildung 2.2: Grid- und Block-Aufbau (Quelle: [3])

Um diese Threads nun zu bearbeiten, werden die Blöcke den SMs einer GPU zugewiesen. Dabei unterteilt man diese in kleinere Gruppen zu je 32 Threads, und führt sie gemeinsam aus Instruktionen werden auf den gesamten Warp ausgeführt, was bedeutet, dass jeder Thread die gleiche Instruktion bearbeitet. Trotzdem können Threads unabhängig voneinander verzweigen, beispielsweise durch ein `if-then-else`-Konstrukt. Tritt eine solche Verzweigung auf, werden alle Wege nacheinander abgearbeitet und alle Threads, die den jeweiligen Weg nicht nehmen, zeitweilig deaktiviert. Ist eine Verzweigung abgeschlossen, werden alle Threads wieder aktiviert und die Ausführung fortgesetzt, was letztendlich dazu führt, dass alle Threads innerhalb eines Warps die gleiche Laufzeit haben.

Innerhalb eines Grids haben Threads und Blöcke für X-, Y- und Z-Richtung einen Index, der innerhalb einer auf der GPU ausgeführten Funktion abgerufen werden kann (siehe

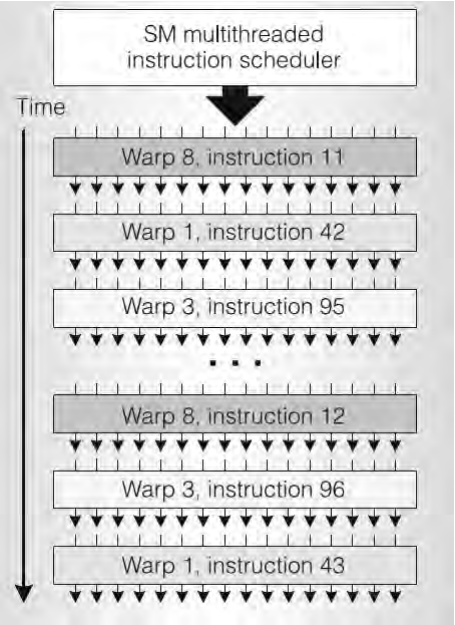

Abbildung 2.3: Ausführung von Thread-Warps (Quelle: [15])

Abschnitt 4.2.2). Diese können dazu verwendet werden für jeden Thread einen anderen Wert aus dem Speicher zu laden und zu bearbeiten.

2.1.2 Verwendbare Speichertypen

Um Daten auf der GPU zu verwenden, müssen diese zunächst in deren Speicher kopiert werden. Das CUDA-Modell unterscheidet dabei zwischen mehreren verschiedenen Typen, die jeweils verschiedene Anwendungsgebiete haben. In den meisten Fällen wird der *Global Memory* verwendet, der dem Arbeitsspeicher der CPU entspricht. Dieser stellt für gewöhnlich die größte Menge an verfügbaren Speicher für GPU-Funktionen und kann von allen Threads in einem Grid gelesen und beschrieben werden.
Für Daten, auf die oft zugegriffen und geschrieben werden muss, ist die Verwendung des *Shared Memory* sinnvoll. Dieser hat eine geringere Zugriffszeit als Global Memory, allerdings wird für jeden Block eine Kopie von Variablen und Vektoren angelegt, sodass Threads nur die Daten für den eigenen Block ändern können.
Für Daten, die jeder Thread einzeln verwaltet werden die On-Chip-Register der GPU und sogenannter `Local Memory`, ein für jeden Thread reservierter Speicher für Vektoren im Global Memory, verwendet.

Eine Besonderheit stellt der `Constant Memory` dar, da er für GPU-Funktionen nur-lesbar ist. Werte in diesem Speicher werden sehr aggressiv gecached [13], was dazu führt, dass die Zugriffszeit darauf sehr gering ist.

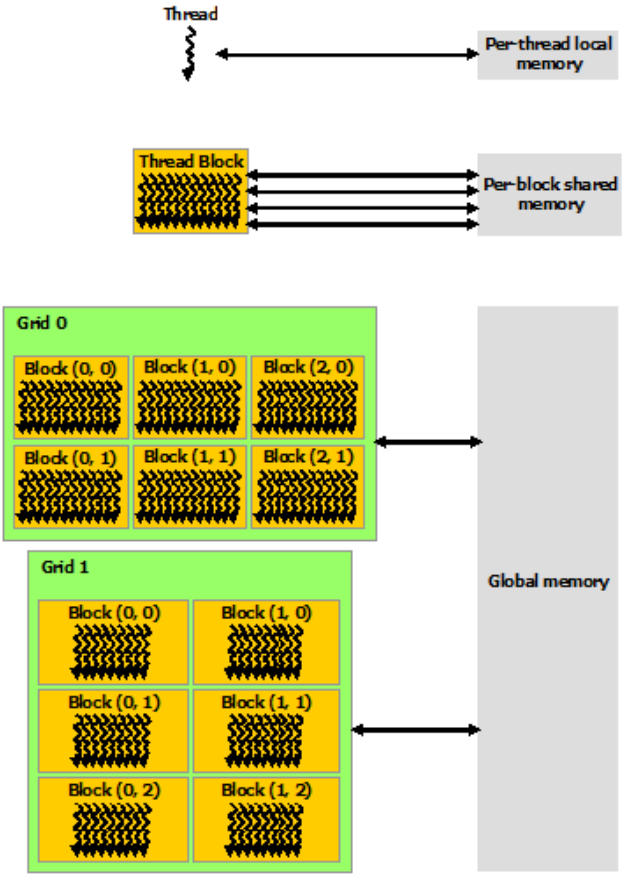

Abbildung 2.4: Speicherhierarchie in CUDA (Quelle: [3])

2.1.3 APIs

Zur Erstellung von Applikationen, die auf der GPU ausgeführt werden, gibt es verschiedene Möglichkeiten. Die einfachste dürfte CUDA C sein, ein C mit minimalen Erweiterungen. Will man mit CUDA ein Programm schreiben, stellt sich zunächst die Frage, welche API man verwenden soll. CUDA stellt dafür zwei APIs zur Verfügung, zum einen die *Runtime API*, die ein hohes Level an Abstraktion bietet und die *Driver API*, welche ein höheres Maß an Management erfordert.

Beide APIs bieten Vor- und Nachteile. Die Runtime API ist durch das hohe Maß an Abstraktion einfacher zu verwenden, da bestimmte Dinge wie beispielsweise Erstellung des Context[1] automatisiert erledigt werden. Die Driver API unterscheidet sich von der Runtime API unter anderem dadurch, dass die Erstellung des Contexts und der Module mit den Kernel-Funktionen manuell erfolgen muss und der Code insgesamt umfangreicher wird. Dafür bietet diese API eine höhere Kontrolle über das verwendete System.
Für gewöhnlich kann man die Runtime API der Driver API vorziehen, da diese einfacher zu verwenden ist, weniger Code erfordert und in den meisten Fällen keinerlei Nachteil gegenüber der Driver API hat. Im weiteren Verlauf wird nun die Runtime API besprochen. Einen Vergleich zwischen Runtime API und Driver API kann man in Kapitel 6 finden.

2.2 NVCC-Compiler

CUDA-Sourcecode setzt sich gewöhnlich aus auf der CPU ausgeführten C/C++ Code und GPU-Code in CUDA C zusammen. Um solchen Code zu kompilieren wird der *NVCC*-Compiler von NVIDIA verwendet, der mit dem CUDA-Toolkit[18] mitgeliefert wird. Zusätzlich wird ein C/C++-Compiler benötigt, der den CPU-Code übersetzt, beispielsweise die *GNU Compiler Collection*[11].

Das Kompilieren läuft in mehreren Schritten ab:

1. Das CUDA-Frontend teilt den Code in Device- und Host-Code auf.
2. Der Device-Code wird je nach Einstellung durch den CUDA-Compiler in ein Binärformat und/oder in PTX-Code übersetzt. Dieser Code wird in einen `device code descriptor` eingebettet, welcher in den Host-Code eingefügt wird.
3. Der Host-Code wird mit dem C/C++-Compiler des Systems übersetzt.

PTX (*Parallel Thread Execution*) ist ein Assembler-Format zur Erstellung von parallel ausführbaren Funktionen, beispielsweise einem CUDA-Kernel.

1 siehe auch Kapitel 6

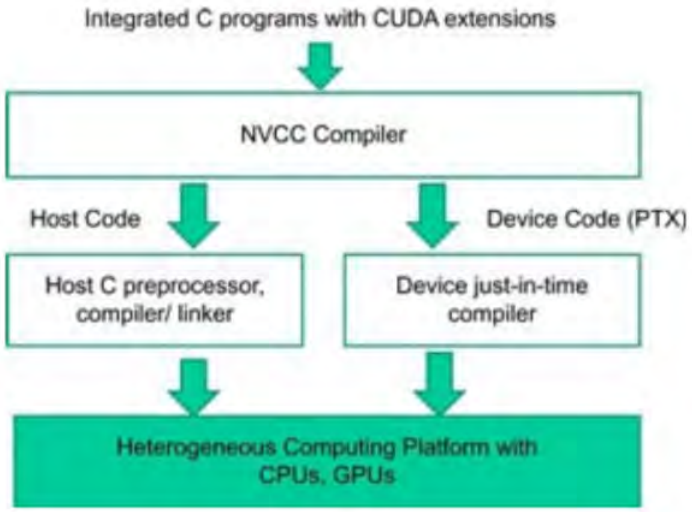

Abbildung 2.5: Kompilationsprozess mit NVCC (Quelle: [13])

3 Beispiel: Fast Wavelet Transform

Ein Beispiel für eine auf der GPU parallelisierbare Anwendung ist die schnelle Wavelettransformation (FWT). Diese wird in der Signalverarbeitung zur Zeit/Frequenz-Analyse verwendet. (Entnommen aus [24])

3.1 Theorie

Definiert ist die Wavelettransformation folgendermaßen:

$$W_\psi f(u,s) := \int_{\mathbb{R}} f(t)\, \frac{1}{\sqrt{|s|}} \overline{\psi\left(\frac{t-u}{s}\right)}\, \mathrm{d}t \tag{3.1}$$

wobei f eine Funktion, u den *localization parameter*, ψ ein sogenanntes Wavelet und s die *Scale* dieses Wavelets bezeichnet. Ein Wavelet ist eine mittelwertfreie Funktion, die in diesem Fall

$$\text{normalisiert} : \|\psi\|_2 = 1 \text{ und zulässig} : \int_{\mathbb{R}} \frac{|\widehat{\psi}(\xi)|^2}{|\xi|}\, \mathrm{d}\xi < \infty$$

ist.

Um die Wavelettransformation auf der GPU zu nutzen muss man diese etwas umformen. Dies ist etwas aufwändiger und wird deshalb hier nicht ausführlich besprochen. Diese lassen sich in [24] nachlesen.

Die Idee hinter der FWT sind die Annahmen, dass bei Software-Applikationen die Eingabewerte meistens nicht in Form einer Funktion, sondern als Vektor der Länge $N \in \mathbb{N}$ (f_N) vorliegen und man eine endliche Menge an *Scales* $S_M := \left[s_j \;:\; j \in \mathbb{Z}_M\right]$, $M \in \mathbb{N}$

verwendet, sowie die Feststellung, dass man die Definition der Wavelettransformation als Faltung

$$f * g := \int_{\mathbb{R}} f(t)\, \overline{g(\cdot - t)}\, \mathrm{d}t \tag{3.2}$$

mit

$$g = \psi_s := \frac{1}{\sqrt{|s|}} \psi\left(\frac{\cdot}{s}\right) \tag{3.3}$$

auffassen kann. Der Vorteil dabei ist, dass man eine Faltung mithilfe der *Fast-Fourier-Transform* (FFT) effizient berechnen kann:

$$W_\psi f(\cdot, s) := f * \psi_s = \left(\widehat{f}\, \widehat{\psi_s}\right)^\vee \tag{3.4}$$

Durch einige Umformungen ([24]) erhält man:

$$\left[W_\psi f(t_k, s) \quad : \quad k \in \mathbb{Z}_N\right] := \left(\widehat{f_N} \odot \widehat{\psi}_{N,s}\right)^\vee \tag{3.5}$$

wobei $\odot$ das *Hadamard*-Produkt und

$$\widehat{\psi}_{N,s} := \left[\sqrt{|s|}\, \widehat{\psi}\left(\frac{k}{N}\frac{2\pi s}{h}\right) \;:\; k \in \mathbb{Z}_N\right] \tag{3.6}$$

ist, mit h als *sampling distance*.

Da wir annehmen, dass die Anzahl an *Scales* endlich ist, lässt sich $\widehat{\psi}_{N,s}$ als Matrix auffassen:

$$\widehat{\Psi}_{M,N} := \left[\sqrt{|s|}\, \widehat{\psi}\left(\frac{k}{N}\frac{2\pi s_j}{h}\right) \quad : \quad \begin{matrix} k \in \mathbb{Z}_N \\ j \in \mathbb{Z}_M \end{matrix}\right] \in \mathbb{C}^{M \times N} \tag{3.7}$$

Jedes Element dieser Matrix steht für eine Skalierung $s_j \in S_M$ und eine Werteposition $k \in \mathbb{Z}_N$ im Wertevektor f_N.

Zusammen mit $\left(\widehat{f_N} \odot \widehat{\psi}_{N,s}\right)^\vee$ erhält man nun eine Matrix

$$\widehat{W} := \left[\left(\widehat{f_N}\right)_j \left(\widehat{\Psi}_{M,N}\right)_{j,k} \quad : \quad \begin{matrix} k \in \mathbb{Z}_N \\ j \in \mathbb{Z}_M \end{matrix} \right] \tag{3.8}$$

in der die Ergebnisse der Wavelettransformation für alle vorgegebenen *Scales* s und alle Werte aus f_N stehen. Nun muss nur noch die inverse FFT von $\widehat{W}$ berechnet werden.

3.2 Implementierung

In dieser Form könnte man eine einfache CPU-Implementierung schreiben, die beispielsweise mit zwei `for`-Schleifen die Elemente der Matrix berechnet (Listing 3.1)

Listing 3.1: Pseudocode FWT auf CPU

```
// h : sampling distance
double constant = (2.0 * PI) / ( ndata * h);

//nscales : number of scales
for (int j = 0; i < nscales; j++)
{
    //ndata : number of data
    for (int k = 0; k < ndata; k++)
    {
       //fft_data[] : data vector after applying FFT
       //scales[] : FWT Scales
       out[k][j] = fft_data[k]
        * Wavelet( constant * ( scales[j] * k) )
        * sqrt( scales[j] );
    }
}
```

In obigen Code haben wir zusätzlich folgende Umformung vorgenommen:

$$\frac{k}{N}\frac{2\pi s}{h} := ks\left(\frac{2\pi}{Nh}\right) \tag{3.9}$$

Da die Anzahl der Werte N und die *sampling distance* h über die gesamte Berechnung unverändert bleiben, kann man den Term in einen variablen Teil ks und einen konstanten Teil $\frac{2\pi}{Nh}$ aufteilen.

Für die Parallelisierung des Codes werden wir hier eine zweidimensionale Anordnung der Thread-Blöcke verwenden, wobei wir den Y-Index `pos.y` für die verwendete *Scale* und den X-Index `pos.x` jedes Threads in jeder Zeile für einen Wert verwenden, sodass wir auf folgendes Schema kommen:

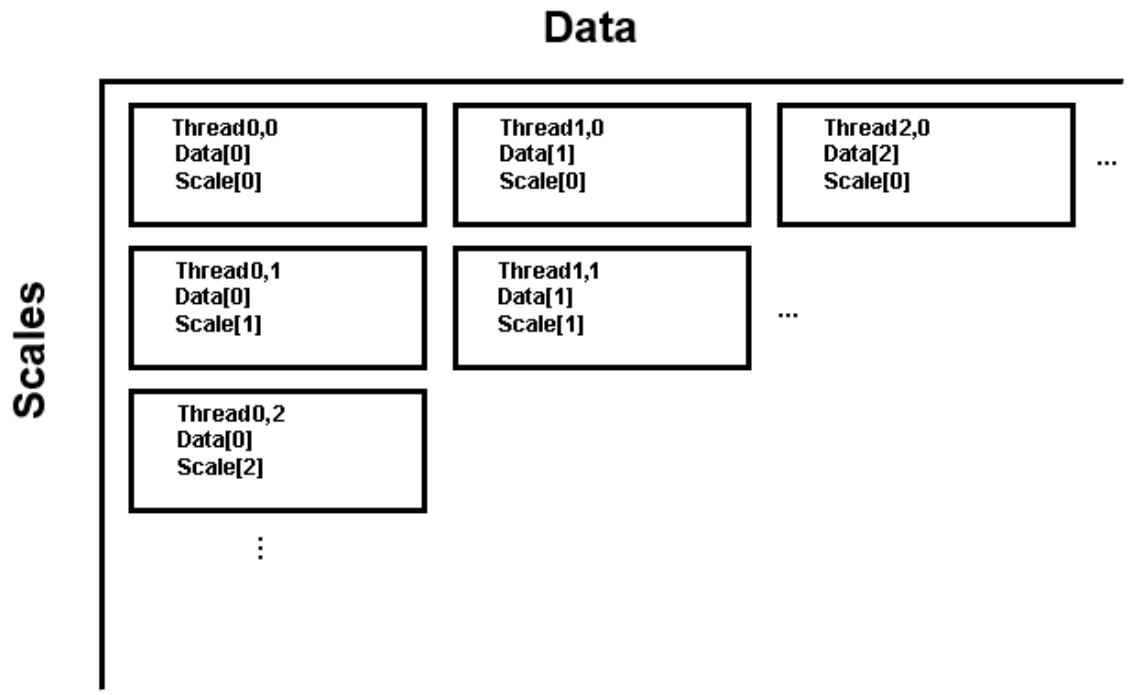

Abbildung 3.1: Verteilung der Werte und Scales auf die Threads

In Pseudo-Code sieht die FWT-Berechnung mit Threads dann beispielsweise so aus:

Listing 3.2: Beispielimplementierung FWT auf GPU

```
1 double constant = (2.0 * PI) / ( ndata * h);
2 int k = pos.x;
3 int j = pos.y
4
5 out[k][j] = fft_data[k]
6         * Wavelet( constant * ( scales[j] * k) )
7         * sqrt( scales[j] );
```

Als Wavelet-Funktion verwenden wir die Fourier-transformierte Funktion des Morlet-Wavelets (Siehe [12]), wobei man mit einigen Änderungen im Code auch die Fourier-transformierte Funktion eines anderen Wavelets verwenden kann.

Um den Code der FWT in den nächsten beiden Kapiteln nicht unnötig umfangreich zu machen, stellen wir hier folgende Bedingungen: Zum einen sollen die Werte-Vektoren bereits Fourier-transformiert sein und zum anderen sollen sie in den Grafikspeicher kopiert sein. Wir werden in Kapitel 5 eine FFT-Bibliothek für CUDA kennenlernen, die ebenfalls voraussetzt, dass Werte-Vektor und Ergebnis-Vektor im Grafikspeicher liegen, sodass wir den Ergebnis-Vektor der FFT als Werte-Vektor verwenden können.

4 CUDA C Grundlagen

Das CUDA-Modell teilt das System in zwei Teile ein, zum einen den Host, der die CPU und den Arbeitsspeicher des Systems umfasst, zum anderen ein oder mehrere Devices, die aus den im System verwendeten GPUs und dem jeweiligen Grafikspeicher bestehen. Um diese Aufteilung widerzuspiegeln wird auch der Sourcecode in auf der CPU ausgeführten Host-Code, der in C oder C++ verfasst wird, und in auf der GPU laufenden Device-Code, welcher mit CUDA C erstellt wird, getrennt. Man muss jedoch beachten, dass das Device keinen Host-Code ausführen kann und umgekehrt, was dazu führt, dass Funktionen der Standardbibliotheken und auch der Runtime API, auf dem Device nicht verwendet werden können.

Wie bereits erwähnt besteht CUDA C aus C mit einigen wenigen Erweiterungen, beispielsweise für die Reservierung von Speicher auf dem Device. Im folgenden Kapitel werden nun diese Erweiterungen und die CUDA Runtime API im Allgemeinen in Verbindung mit einem einzelnen Device angesprochen und anhand einiger Beispiele erklärt.

4.1 Host-, Kernel- und Device-Funktionen

Die erste Erweiterung gegenüber C betrifft die Art der Funktionen. CUDA C verwendet Qualifier um die Verwendung und die ausführende Plattform (CPU oder GPU) festzulegen. Dabei gibt es drei verschiedene Typen von Funktionen:

- **Host-Funktion:** Host-Funktionen entsprechen den bereits aus C bekannten Funktionstypen und bezeichnen alle Funktionen, die von der CPU ausgeführt werden. In CUDA werden alle Funktionen, die keinen anderen Qualifier verwenden als Host-Funktionen behandelt und optional mit dem Qualifier `__host__` bezeichnet.

Host-Funktionen können nur von anderen Host-Funktionen aufgerufen werden und ihrerseits Kernel-Funktionen und andere Host-Funktionen aufrufen.

Listing 4.1: Beispiel für eine Host-Funktion

```
1 __host__ int function()
2 {
3     return 0;
4 }
```

- **Kernel-Funktion:** Als Kernel-Funktion oder nur Kernel werden Funktionen bezeichnet, die den von Threads parallel ausgeführten Code enthalten. Kernel werden auf dem Device ausgeführt und mit dem Qualifier **`__global__`** bezeichnet. Sie haben den Rückgabewert `void`, können von Host-Funktionen aus aufgerufen werden und selbst Device-Funktionen aufrufen[1].

Listing 4.2: Beispiel für eine Kernel-Funktion

```
1 __global__ void kernel()
2 {
3 }
```

- **Device-Funktion:** Device-Funktionen werden, wie der Name bereits andeutet, auf dem Device ausgeführt und verwenden den Qualifier **`__device__`**. Sie können von einer Kernel-Funktion oder von anderen Device-Funktionen aufgerufen werden.

Listing 4.3: Beispiel für eine Device-Funktion

```
1 __device__ int dev_fuction()
2 {
3     return 0;
4 }
```

Da CUDA jede Funktion zunächst als Host-Funktion betrachtet, können wir so gut wie jedes in Standard-C (zum Beispiel C99) verfasste Programm als CUDA-Programm betrachten, das dann eben nur Host-Code enthält. Da der CUDA-Compiler NVCC Host- und Device-Code vor dem Übersetzen aufteilt und Host-Code an den System-Compiler weitergereicht wird, macht das keinen nennenswerten Unterschied (siehe auch Abschnitt 2.2).

1 Devices, die Compute Capability 3.0 oder höher unterstützen, können Kernel-Funktionen auch von anderen, auf dem Device ausgeführten Funktionen aufrufen lassen[3]

In unserer Implementierung der FWT verwenden wir die Funktionstypen folgendermaßen: Wir verwenden eine Host-Funktion `runCudaWaveletTransform()`, die von anderen Host-Funktionen zur Ausführung der Wavelettransformation eines Fourier-transformierten Werte-Vektors aufgerufen wird und selbst die Kernel-Funktion, die die FWT berechnet, initialisiert und aufruft. Diese Funktion erhält die folgenden Parameter: die Anzahl der Scales (`nscales`)und der Werte (`ndata`) als konstante Integer-Werte, einen Pointer auf den Vektor mit den Fourier-transformierten Werten im Device-Speicher (`*d_data`) und einen Pointer auf eine Ergebnismatrix im Device-Speicher (`*d_result`). Da sowohl Eingabe als auch Ergebnis im komplexen Zahlenbereich liegen, verwenden wir als Datentyp `cuDoubleComplex`, der in `cuComplex.h` definiert ist.

Listing 4.4: FWT Host-Funktion

```
__host__ void runCudaWaveletTransform( const unsigned int nscales,
                                       const unsigned int ndata,
                                       const cuDoubleComplex *d_data,
                                       cuDoubleComplex *d_result )
{
    //init and call FWT-kernel
}
```

In der Kernel-Funktion `waveletKernel()` wird die eigentliche Berechnung der FWT durchgeführt. Dafür benötigen wir die gleichen Parameter, die bereits der Host-Funktion `runCudaWaveletTransform()` übergeben wurden.

Listing 4.5: FWT Kernel-Funktion

```
__constant__ double d_constScales[1024];
__constant__ float constant;

__global__ void waveletKernel( const unsigned int nscales,
                               const unsigned int ndata,
                               const cuDoubleComplex *d_data,
                               cuDoubleComplex *d_result )
{
    int tid;
    int scaleID;

    cuDoubleComplex data;
    double scale;

// calculate FWT
        double r = WSMorlet( ( constant * tid ) * scale) * sqrt(scale);
        cuDoubleComplex s = make_cuDoubleComplex(r * data.x, r * data.y);

// save result in matrix
        d_result[(scaleID * ndata) + (tid)] = s;
}
```

Die Variablen `tid` und `scaleID` bezeichnen Zeilenindex des Threads und den Spaltenindex des jeweiligen Blocks (siehe Abschnitt 4.2.2). Diese werden später dazu verwendet, den Wert und die `Scale` aus dem Speicher zu laden. `d_constScales[]` und `constant` werden mithilfe von Host-Funktionen (siehe unten) zuvor in den Constant Memory geladen.
Die Berechnung der FWT selbst wird in mehrere Teile aufgeteilt: Zuerst wird der Term $\widehat{\psi}\left(\frac{2\pi}{Nh}ks\right)\sqrt{s}$ berechnet. Dieser wird dann mit Real- und Imaginärteil des (Fourier-transformierten) Eingabewertes multipliziert und das Ergebnis mithilfe der in `cuComplex.h` verfügbaren Funktion `make_cuDoubleComplex()` wieder in eine komplexe Zahl gespeichert, die im letzten Schritt an der entsprechenden Stelle in der Ergebnismatrix `d_result` geschrieben wird.

Des weiteren verwenden wir für das Wavelet eine Device-Funktion namens `WSMorlet()`, in der wir die Fourier-transformierte Funktion des Morlet-Wavelets implementieren. `d_constMorlet[]` enthält konstante Werte, die wir zuvor mit einer Host-Funktion (siehe unten) in den Constant Memory des Device kopieren müssen. Als Parameter erhält das Wavelet einen `double`-Wert `x`, der aus dem Term $ks\frac{2\pi}{Nh}$ besteht (Siehe Kapitel3), wobei k, eine Position im Werte-Vektor, s eine *Scale*, N die Anzahl der Werte im Werte-Vektor und h die *sampling distance* ist.

Listing 4.6: Morlet-Wavelet-Funktion (Fouriertransformiert)

```
__constant__ double d_constMorlet[2];
__constant__ double SQRTSQRT_PI = 1.33133536380039;

__device__ double WSMorlet( double x )
{
    return  SQRTSQRT_PI * ( exp( - ((x - d_constMorlet[0]) * (x - d_constMorlet[0])) / 2.0f )
        - exp( -(x * x) / 2.0f - d_constMorlet[1] ) );
}
```

Zuletzt verwenden wir noch drei Host-Funktionen, die zum Kopieren der `Scales`, der Morlet-Konstanten und der Konstante $\frac{2\pi}{Nh}$ in den Constant Memory verwendet werden. Diese Funktionen müssen vor dem Aufruf der eigentlichen FWT-Funktion ausgeführt werden, damit diese korrekt ausgeführt wird.

Listing 4.7: Host-Funktionen zum Kopieren der Konstanten Werte in den Constant Memory

```
1  void setMorletParams(double *morlParams)
2  {
3      //Set d_constMorlet[]
4  }
5
6  void setConstScales(double *fScales, unsigned int nscales)
7  {
8      //Set d_constScales[]
9  }
10
11 void setConstant(int ndata, double h)
12 {
13     float constant_h = ( 2.0 * M_PI ) / ( ndata * h );
14     //Set constant
15 }
```

Eine Besonderheit bilden Funktionen mit dem Qualifier `__host__ __device__`. In manchen Fällen werden bestimmte Funktionen sowohl von Host-Funktionen als auch von Device-Funktionen benötigt. Um sich nun den Aufwand zu sparen solche Funktionen doppelt zu implementieren kann man den Qualifier `__host__ __device__` verwenden, sofern die Funktion keine Funktionen aufruft, die nur auf dem Host oder dem Device ausgeführt werden, beispielsweise aus der Standardbibliothek von C bzw. C++. Wird eine solche Funktion kompiliert, erstellt der Compiler jeweils eine Version für den Host und das Device.

4.2 Parallele Ausführung mit Threads

Um die Multiprozessoren einer GPU auszulasten, werden zur Berechnung einer (parallelisierbaren) Aufgabe eine große Anzahl an Threads eingesetzt. Im folgenden Abschnitt werden wir sehen, wie man eine Kernel-Funktion aufruft, ein- und mehrdimensionale Grids erstellt und Threads synchronisiert.
Die Parallelisierung einer Aufgabe geschieht in der Kernel-Funktion. Diese wird so erstellt, dass die Funktion selbst nur ein einziges Element aus der Werte-Menge oder einen kleinen Teil der Elemente verarbeitet. Um auf diese Elemente zuzugreifen, werden die Indices der Threads bzw. der Thread-Blöcke verwendet um verschiedene Elemente zu laden.

Um einen Kernel auszuführen, wird im Normalfall eine größere Anzahl an Threads gestartet, von denen jeder eine Kopie dieses Kernels ausführt. Die Gesamtheit aller Threads wird als *Grid* bezeichnet, welches entweder eindimensional oder mehrdimensional angelegt werden kann. Die maximal mögliche Dimension ist abhängig von der Compute Capability des ausführenden Device. Innerhalb eines Grids werden die Threads in sogenannte Thread-Blöcke oder einfach nur Blöcke zusammengefasst. Auch diese können sowohl ein- als auch mehrdimensional sein, wiederum abhängig vom ausführenden Device.

4.2.1 Kernel-Aufruf

Der Aufruf eines Kernels mit der Runtime API unterscheidet sich leicht von gewöhnlichen Funktionsaufrufen. Um einen Kernel aufzurufen wird folgendes Schema verwendet:

```
kernel_name <<< X , Y , ... >>> ( param1, param2, ... );
```

`kernel_name` ist hierbei der Name der Kernel-Funktion, in unserem Fall also `waveletKernel` und `(param1, param2, ...)` wie auch bei normalen Funktionsaufrufen die Parameter der Kernel-Funktion. Der Term `<<< X , Y , ... >>>` wird als *execution configuration parameters* bezeichnet[13]. Diese legen unter anderem die Anzahl der Threads fest, die bei einem Kernel-Aufruf gestartet werden sollen.

Im einfachsten Fall sieht ein Aufruf so aus:

```
kernel_name <<< int X , int Y >>> ( param1, param2 ... ); // X,Y > 0
```

Damit wird ein eindimensionales Grid erstellt, wobei `X` die Anzahl der Blöcke und `Y` die Anzahl der Threads in jedem Block angeben, sodass insgesamt also $X * Y =: N$ Threads in einem eindimensionalen Grid der Größe N gestartet werden.

4.2.2 Thread-, Block- und Grid-Variablen

Das Ziel bei der Verwendung eines Grids aus Threads ist in fast allen Fällen die Verarbeitung von Daten oder die parallele Ausführung einer Berechnung mit verschiedenen Parametern. Wir sind zwar in der Lage ein- und mehrdimensionale Grids zu erstellen, aber um diese auch sinnvoll nutzen zu können, müssen wir auch die Koordinaten der Threads in einem Grid kennen um beispielsweise verschiedene Daten aus einem Vektor zu laden.

CUDA stellt dafür eingebaute Variablen vom Typ `dim3`, einer dreidimensionalen Struktur, innerhalb einer Kernel-Funktion zur Verfügung, namentlich `threadIdx`, `blockIdx`, `blockDim` und `gridDim`

`threadIdx.x`, `threadIdx.y` und `threadIdx.z` geben dabei die X,Y und Z-Koordinate eines Threads innerhalb des Grids an. Diese beginnen in jedem Block mit , sodass die Threads in verschiedenen Blöcken die selben Werte für `threadIdx` besitzen.
Um auf die Koordinaten eines Thread-Blocks zuzugreifen verwendet man `blockIdx.x`, `blockIdx.y` und `blockIdx.z`. Für die Anzahl Threads in jede Dimension werden `blockDim.x` bzw. `blockDim.y` und `blockDim.z` benutzt.
Zusätzlich kann man mit `gridDim.x`, `gridDim.y` und `gridDim.z` die Anzahl der Blöcke im Grid in X-, Y- und Z-Richtung abrufen.
Diese Variablen können nur von Funktionen verwendet werden, die auf dem Device ausgeführt werden.

Um beispielsweise den globalen Index eines Threads in einem zweidimensionalen Grid mit eindimensionalen Blöcken, also seine eindeutige Index-Nummer, herauszufinden, können wir folgendes verwenden:

```
int tid = threadIdx.x
          + blockIdx.x * blockDim.x
          + blockIdx.y * blockDim.x * gridDim.x
```

Sehen wir uns ein Beispiel für die Verwendung dieser Variablen an. Wir verwenden folgenden Kernel zur Vektoraddition (Listing 4.8):

Listing 4.8: Vektoraddition-Kernel

```
#define N 10

__global__ void VectorAddKernel(int *vector_a, int *vector_b, int *result)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x

    if ( tid < N ) result[tid] = vector_a[tid] + vector_b[tid];
}
```

Die Eingabeparameter sind zwei Werte-Vektoren `vector_a`und `vector_b` der Länge N, die in einer Host-Funktion bereits vorher mit Integer-Werten gefüllt und in den Speicher des Device kopiert wurden, und ein leerer Ergebnis-Vektor `result`, für den nur Speicher im Device-Speicher reserviert wurde. N wurde mit `#define N` auf einen beliebigen Wert, beispielsweise auf 10, festgelegt.
In diesem Kernel werden nun die Werte an der Position `tid` der Eingabevektoren `vector_a` und `vector_b` addiert und in `result` gespeichert. `tid` ist hierbei der eindeutige Index des ausführenden Threads in einem eindimensionalen Grid mit eindimensionalen Blöcken.

Wir werden später sehen, dass, besonders für große N, die sinnvollste Verteilung von Threads in einem Grid von der Form *`P` Blöcke mit jeweils `Q` Threads* ist, gegenüber der hier vielleicht offensichtlicheren und einfacheren Form *Ein Block mit `N` Threads* oder *`N` Blöcke mit jeweils einem Thread*.
Werden in den letzten beiden Fällen genau N Threads gestartet, müssen wir bei der ersten Verteilung darauf achten, dass $PQ \geq N$ gilt um auf das richtige Ergebnis zu kommen.

```
VectorAddKernel <<< P , Q >>> ( vector_a, vector_b, result );
```

Da wir in diesem Fall auch mehr als N Threads startet können, müssen wir sicherstellen, dass nicht auf unerlaubte Speicherbereiche zugegriffen wird. Um dies zu verhindern setzen wir die Bedingung, dass die Berechnung nur durchgeführt wird, wenn $tid < N$ ist:

```
if ( tid < N ) result[tid] = vector_a[tid] + vector_b[tid];
```

So verhindern wir, dass Threads, deren Index größer als die Anzahl der Elemente in den Vektoren ist, die Berechnung durchführen und auf nicht reservierten oder fremden Speicherbereich zugreifen.

4.2.3 Mehrdimensionale Blöcke und Grids

Wir haben bereits gesehen, wie man ein eindimensionales Grid zur Bearbeitung einer Vektoraddition erstellt. In manchen Fällen kann es jedoch auch sinnvoll oder hilfreich sein, wenn man auch die Y-Koordinaten eines Threads oder Blocks verwenden kann um beispielsweise auf Daten im Speicher zuzugreifen. Ein Beispiel dafür ist unsere FWT-Implementierung. Wenn wir uns die Berechnung einmal bildlich vorstellen, sehen wir, dass diese die Form einer Matrix besitzt, bei der in jeder Zeile eine *Scale* auf den Eingabe-Vektor angewendet wird.

Es macht hier Sinn, ein zweidimensionales Grid mit `ndata` Threads in X-Richtung und `nscales` Threads in Y-Richtung zu verwenden, in dem alle Threads einer Zeile die gleiche `Scale` auf einen Eingabewert anwenden. Wir können den Zeilenindex eines Threads, der zwischen `0` und `ndata-1` liegt, verwenden um einen Wert aus dem Werte-Vektor `d_data` zu laden und, wenn wir eindimensionale Blöcke verwenden, den Y-Index des Blocks um eine *Scale* aus dem *Scale*-Array `d_constScales[]` zu holen. Dieses Array wird der Wavelet-Funktion nicht als Parameter übergeben, sondern muss vorher mit der Host-Funktion `setConstScales()` in den Constant Memory des Device geladen werden.
Um ein mehrdimensionales Grid zu erstellen, müssen wir den Kernel-Aufruf etwas verändern. Anstelle der Integer-Werte X und Y verwenden wir zwei `dim3`-Strukturen, um die Dimensionen des Grids und der Blöcke festzulegen.

Ein eindimensionales Grid mit eindimensionalen Blöcken wird beispielsweise so aufgerufen:

```
dim3 gridDim(2,1,1);
dim3 blockDim(4,1,1);

kernel<<<gridDim, blockDim>>>();
```

In diesem Fall wird ein Grid mit 2 Blöcken erstellt, die jeweils 4 Threads enthalten, es werden also insgesamt 8 Threads gestartet.
Dieser Aufruf ist äquivalent zu

```
kernel<<<2,4>>>();
```

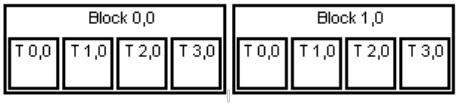

Abbildung 4.1: 2 Blöcke mit 4 Threads - eindimensional

Für ein zweidimensionales Grid mit zweidimensionalen Blöcken sieht das ganze dann so aus:

```
dim3 gridDim(2,2,1);
dim3 blockDim(4,4,1);

kernel<<<gridDim, blockDim>>>();
```

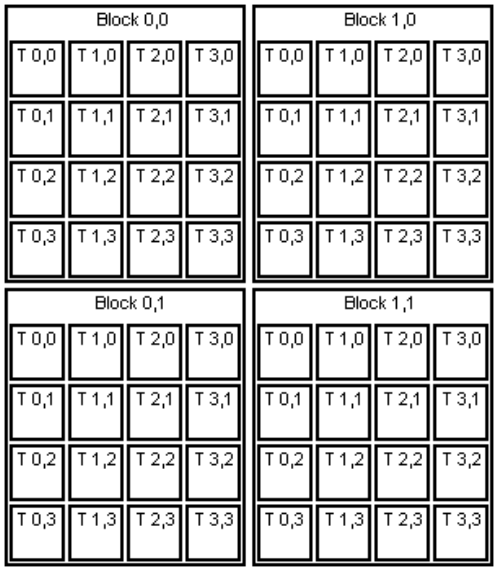

Abbildung 4.2: 2x2 Blöcke mit 4x4 Threads - zweidimensional

Hier enthält das Grid bei Ausführung insgesamt 2*2, also 4 Blöcke, die jeweils 4*4 = 16 Threads enthalten. Die Größe des Grids beträgt also 64 (4*16) Threads.

Wie schon bei eindimensionalen Grids und Blöcken werden auch hier die Threads und Blöcke durchnummeriert, jetzt allerdings auch in Y-Richtung. Man kann also für jeden Thread und jeden Block eine X- und eine Y-Koordinate im Grid abfragen. Für Threads geht das über die schon bekannte Variable `threadIdx.x` für die X-Position und zusätzlich über `threadIdx.y`, bei Blöcken wird für die Y-Position `blockIdx.y` verwendet. Eine weitere Möglichkeit sind dreidimensionale Grids und dreidimensionale Blöcke.

Für die FWT-Implementierung verwenden wir an dieser Stelle eindimensionale Blöcke mit einem Thread. Dies wird an späterer Stelle noch überarbeitet, würde hier aber den Code unnötig verkomplizieren. Unser Grid wird also aus $N := \texttt{ndata} * \texttt{nscales}$ Threads bestehen.

In unserer Host-Funktion `runCudaWaveletTransform()` wird der Kernel mit

```
dim3 gridDim(ndata, nscales , 1);
dim3 blockDim(1, 1, 1);

waveletKernel<<<gridDim, blockDim>>>( nscales,
                                      ndata,
                                      d_data,
                                      d_result );
```

aufgerufen.
In der Kernel-Funktion `waveletKernel()` werden ebenfalls Änderungen vorgenommen (siehe Listing 4.9)

Listing 4.9: FWT Kernel-Funktion (Erweiterung)

```
1 __constant__ double d_constScales[1024];
2 __constant__ float constant;
3
4 __global__ void waveletKernel( const unsigned int nscales,
5                                const unsigned int ndata,
6                                const cuDoubleComplex *d_data,
7                                cuDoubleComplex *d_result )
8 {
9
```

```
10      int tid = blockIdx.x * blockDim.x + threadIdx.x;
11      int scaleID = blockIdx.y;
12
13      cuDoubleComplex data;
14      double scale;
15
16      if (scaleID < nscales && tid < ndata)
17      {
18  // load values
19          scale = d_constScales[scaleID];
20          data = d_data[tid];
21
22  // calculate FWT
23          double r = WSMorlet( ( constant * tid ) * scale) * sqrt(scale);
24          cuDoubleComplex s = make_cuDoubleComplex(r * data.x, r * data.y);
25
26  // save result in matrix
27          d_result[(scaleID * ndata) + (tid)] = s;
28      }
29  }
```

Zum einen haben wir bereits erwähnt, dass die Variablen `tid` und `scaleID` für Zeilenindex des Threads und Spaltenindex des Blocks verwendet werden. Die zweite Änderung bzw. Erweiterung betrifft das Laden des zu verwendenden Wertes und der Scale aus dem Speicher. Dazu verwenden wir die Variablen `data` und `scale`:

```
scale = d_constScales[scaleID];
data = d_data[tid];
```

Zuletzt müssen wir an dieser Stelle wieder sichergehen, dass wir nicht auf unerlaubten Speicherbereich zugreifen. Dazu wird analog zu Abschnitt 4.2.2 ein `if-then`-Konstrukt verwendet:

```
if (scaleID < nscales && tid < ndata)
{
...
}
```

4.2.4 Thread-Synchronisation

Betrachten wir folgenden Kernel (Listing 4.10)

Listing 4.10: Berechnung ohne Thread Synchronisation

```
1  __global__ void kernel(int *data, int *result, int ndata)
2  {
3  int tid = threadIdx.x;
4
5  // double each value in data
6  if ( tid < ndata) data[tid] = data[tid] * 2;
7
8  // calculate
9  if ( tid < (ndata - 1) ) result[tid] = data[tid] + data[tid + 1];
10 else if ( tid == (ndata - 1) ) result[tid] = data[tid] + data[0];
11 }
```

Aufgerufen wird dieser mit

```
kernel<<<1,N>>>( data, result, N );
```

wobei `N` die Anzahl der Elemente in den Vektoren `data` und `result` ist. Wir starten ein Grid, das aus einem Block mit N Threads besteht.
Die Berechnung, die dieser Kernel durchführt, ist an sich nicht weiter von Bedeutung, allerdings können wir hier sehen, was bei mangelnder Synchronisation der Threads passieren kann.
Dieser Kernel erhält als Parameter einen mit Werten gefüllten Vektor `data` der Länge N, einen Ergebnis-Vektor `result` der Länge N und mit `ndata` die Anzahl N der Elemente. Die Berechnung selbst besteht aus zwei Teilen:

1. Jeder Thread soll zunächst den Wert im Werte-Vektor, der seinem Index entspricht, verdoppeln und
2. diesen Wert mit dem (bereits verdoppelten) Wert an der Position `tid+1`, bzw. wenn der Thread den maximalen Index besitzt, an der Stelle 0 addieren und im Ergebnis-Vektor speichern.

Führen wir diesen Kernel nun mit $N := 100$ aus[2], erhalten wir unter Umständen folgende Ausgabe:

```
...
kernel( 93 ) = 374, should be: 374
kernel( 94 ) = 378, should be: 378
kernel( 95 ) = 286, should be: 382
kernel( 96 ) = 386, should be: 386
kernel( 97 ) = 390, should be: 390
kernel( 98 ) = 394, should be: 394
kernel( 99 ) = 198, should be: 198
...
```

Was ist hier passiert?
Da wir im zweiten Teil der Berechnung auf Daten zugreifen, die von einem anderen Thread bearbeitet wurden, kann es passieren (muss aber nicht), dass dieser sich noch an einer anderen Position im Code befindet, sei es durch Zuteilung zu einem anderen, noch nicht ausgeführten Warp (siehe hierzu auch Abschnitt 5.2) oder durch eine Verzweigung.
In jedem Fall greifen wir hier auf einen noch nicht bearbeiteten Wert zu, was zu einem falschen Ergebnis führt. Die Lösung für dieses Problem ist es, die einzelnen Threads zu synchronisieren. CUDA verwendet dafür im Kernel eine eingebaute Funktion namens

```
__syncthreads();
```

die an der Stelle, an der sie im Kernel aufgerufen wird eine *barrier synchronisation* zwischen allen Threads in einem Block durchführt. Sobald ein Thread an eine Stelle im Code kommt, wo er `__syncthreads()` aufruft, wartet er an dieser Stelle, bis alle anderen Threads im selben Block ebenfalls das selbe `__syncthreads()` aufrufen. Danach fahren alle Threads in diesem Block an dieser Position mit der Ausführung fort.

2 Die main-Funktion dafür ist in Anhang A.2 zu finden

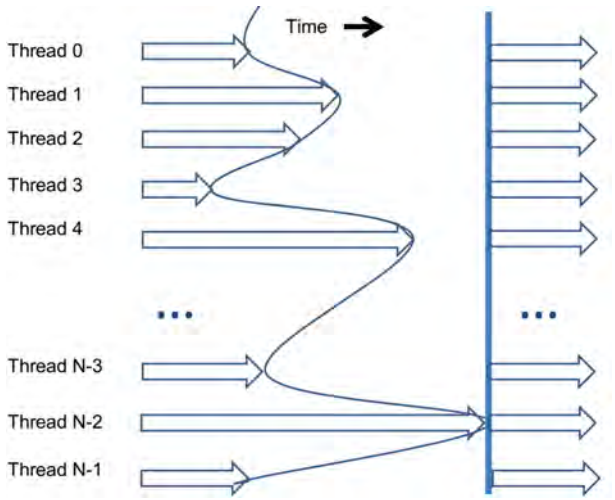

Abbildung 4.3: Synchronisation mithilfe einer Barriere (Quelle: [13])

Um in unserem Kernel also sicherzugehen, dass jeder Wert zuerst verdoppelt wird, bevor wir im zweiten Schritt zur Addition zweier Werte kommen, fügen wir nach

```
if ( tid < ndata) data[tid] = data[tid] * 2;
```

ein `__syncthreads()` ein (Listing 4.11).

Listing 4.11: Berechnung mit Thread-Synchronisation

```
1  __global__ void kernel2(int *data, int *result, int ndata)
2  {
3  int tid = threadIdx.x;
4
5  // double each value in data
6  if ( tid < ndata) data[tid] = data[tid] * 2;
7  __syncthreads();
8
9  // calculate
10 if ( tid < (ndata - 1) ) result[tid] = data[tid] + data[tid + 1];
11 else if ( tid == (ndata - 1) ) result[tid] = data[tid] + data[0];
12 }
```

Ein Punkt, der bei Thread-Synchronisation zu beachten ist, sind `__syncthreads()`-Statements in `if-then-else`-Konstrukten. Soll in einem Kernel eine Synchronisation in einem `if`-Block durchgeführt werden, muss sichergestellt sein, dass **alle** Threads innerhalb eines Blocks diese Bedingung erfüllen.

Listing 4.12: if-then-else mit Thread-Synchronisation

```
__global__ void kernel()
{
    if ( threadIdx.x % 2 == 0 )
    {
        // ...
        __syncthreads();
    }
    else
    {
        // ...
        __syncthreads();
    }
}
```

Führt man eine Kernel-Funktion aus, die wie in Listing 4.12 aufgebaut ist, wird ein Teil der Threads den `then`-Block ausführen, der Rest den `else`-Block. In beiden Blöcken befindet sich ein `__syncthreads()`-Statement. Da CUDA jeden Synchronisationspunkt einzeln behandelt, wird jeder Thread darauf warten, dass alle anderen den selben Synchronisationspunkt erreichen, was zu einem Stillstand führt. Das *CUDA Runtime System* stellt zwar sicher, dass es nicht zu einer endlosen Wartezeit bei einer Synchronisation kommt und die Threads weiter arbeiten, allerdings kann es dabei zu undefinierbaren Verhalten und Seiteneffekten kommen.

Abschließend sei hier noch erwähnt, dass CUDA keine Synchronisation verschiedener Blöcke ermöglicht. Der Grund dafür ist, dass Blöcke als Arbeitseinheiten angesehen werden, welche unabhängig voneinander ausgeführt werden können. Dadurch, dass man nicht auf andere Blöcke warten muss, kann man diese in beliebiger Reihenfolge ausführen und auch beliebig den SMs zuteilen, was eine bessere Skalierbarkeit des Programms ermöglicht, ohne dass etwas im Code geändert werden muss.

4.3 CUDA Speichertypen

Wie Eingangs erwähnt, besteht ein Device aus der GPU und dem dazugehörenden Speicher. Im folgenden Abschnitt beschäftigen wir uns nun mit den verschiedenen Speichertypen, die auf dem Device zur Verfügung stehen.

Da wir in CUDA zwischen Host- und Device-Speicher unterscheiden, müssen wir auch die Pointer auf die jeweiligen Speicherbereiche unterscheiden. Ein Host-Pointer zeigt auf

reservierten Speicher auf dem Host, während ein Device-Pointer auf Device-Speicher zeigt. Host- und Device-Pointer können sowohl Host-Funktionen als auch Kernel- oder Device-Funktionen als Parameter übergeben werden, allerdings können diese nur in Funktionen, die auf der jeweiligen Plattform ausgeführt werden, dereferenziert werden, was bedeutet, dass auf Host-Pointer nur in Host-Funktionen zugegriffen werden kann und analog dazu auf Device-Pointer nur in Funktionen, die auf der GPU ausgeführt werden.

Es gibt mehrere Arten von Speicher, die jeweils unterschiedliche Eigenschaften und Gültigkeitsbereiche haben:

- **Global Memory:** Variablen in diesem Speicher sind für alle Threads in einem Grid sichtbar und können sowohl gelesen als auch geschrieben werden. Sie werden mit dem Modifier `__device__` markiert und sind während der gesamten Programmlaufzeit gültig.
- **Constant Memory:** Konstante Werte auf dem Device werden im Constant Memory gespeichert und verwenden bei der Deklaration die Modifier `__device__ __constant__` oder einfach nur `__constant__`. Werte im Constant Memory sind ähnlich wie im Global Memory für alle Threads in einem Grid sichtbar, können aber auf dem Device nur ausgelesen werden. Die Gültigkeitsdauer einer solchen Variablen ist die gesamte Laufzeit des Programms.
- **Shared Memory:** Auf Variablen im Shared Memory haben nur Threads innerhalb eines Blocks Lese- und Schreibzugriff. Diese Variablen werden mit dem Modifier `__shared__` markiert und sind nur während der Laufzeit des jeweiligen Kernels gültig.
- **Register und Local Memory:** Variablen die innerhalb einer Kernel-Funktion erstellt werden, speichert man in den On-Chip-Registern der GPU. Sie sind nur für den jeweiligen Thread sichtbar. Innerhalb einer Kernel-Funktion erstellte Vektoren werden hingegen im Local Memory gespeichert. Es handelt sich dabei um einen Bereich des Global Memory, der für jeden Thread einzeln reserviert wird und nur von diesem Thread gesehen werden kann.
- **Texture Memory:** Texture Memory wird, ähnlich dem Constant Memory, als nur-lesbarer Speicher verwendet, was es wiederum ermöglicht, ihn effektiv zu cachen.

Die Verwendung ist besonders dann effizient, wenn gleichzeitig auf mehrere nah zusammen liegende Speicherbereiche zugegriffen wird.

`__device__`- und `__constant__`- Variablen werden außerhalb eines Funktionskörpers deklariert, wohingegen `__shared__`-Variablen in der Kernel-Funktion deklariert werden.

In den folgenden Abschnitten werden nun einige Speichertypen genauer vorgestellt.

4.3.1 Global Memory

Wollen wir im Host-Speicher liegende Daten auf dem Device bearbeiten, müssen wir diese zunächst in den Global Memory kopieren.

Verwenden wir als Beispiel den Kernel zur Vektoraddition aus Abschnitt 4.2.2. Wir wollen nun 2 Vektoren mit jeweils 4 Werten addieren[3].

Erstellt werden zwei Vektoren, die mit Werten gefüllt werden und ein Vektor, in dem das Ergebnis gespeichert werden soll. Diese Vektoren werden im Speicher des Hosts angelegt. Um nun eine Vektoraddition auf dem Device durchzuführen müssen diese zunächst in den Global Memory kopiert werden.

Dazu werden zunächst drei Pointer wie in C/C++ bekannt angelegt:

```
1 int *d_vec_a;
2 int *d_vec_b;
3 int *d_result;
```

Im nächsten Schritt soll nun Speicher im Global Memory reserviert werden, auf den die angelegten Pointer zeigen sollen. Dazu wird die API Funktion `cudaMalloc()` verwendet:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

`cudaMalloc()` funktioniert ähnlich wie `malloc()` auf dem Host. Es werden `size` Byte an Speicher im Global Memory reserviert, auf die wir mit dem Device-Pointer `devPtr` zugreifen können.

3 Vollständiges Listing A.3

Will man dagegen Speicher freigeben verwendet man , analog zu `free()` in C

```
cudaError_t cudaFree ( void* devPtr ).
```

Nun werden mit `cudaMemcpy()` die Inhalte der beiden Vektoren jeweils in die reservierten Speicherbereiche kopiert

```
cudaError_t cudaMemcpy ( void* dst,
                         const void* src,
                         size_t count,
                         cudaMemcpyKind kind )
```

Diese Funktion kopiert den Inhalt des Speichers, auf den der Pointer `src` zeigt an die Speicherposition im Global Memory, die vom Pointer `dst` markiert wird. `count` gibt dabei die Menge des Speichers an, die kopiert werden soll, in Bytes und `kind` die Kopierrichtung (Host->Device (`cudaMemcpyHostToDevice`), Device->Host (`cudaMemcpyDeviceToHost`), Host->Host (`cudaMemcpyHostToHost`) und Device->Device (`cudaMemcpyDeviceToDevice`)) an.

```
1  //For the sake of completeness
2  //#define SIZE 4
3
4  //Memory Allocation on Device
5  cudaMalloc((void **) &d_vec_a, SIZE*sizeof(int));
6  cudaMalloc((void **) &d_vec_b, SIZE*sizeof(int));
7  cudaMalloc((void **) &d_result, SIZE*sizeof(int));
8
9  //Copy vectors to device
10 cudaMemcpy(d_vec_a, vec_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
11 cudaMemcpy(d_vec_b, vec_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);
```

Diese Vektoren können jetzt der dem Kernel zu Verarbeitung übergeben werden. Dieser berechnet nun die Vektoraddition und speichert das Ergebnis im Vektor `d_result`. Damit das Ergebnis auf dem Host verwendet werden kann, muss es wieder in den Host-Speicher kopiert werden.

```
1  //Run Kernel
2  int threads = 192;
3  int blocks = (int) ( (SIZE + threads - 1) / threads );
4  VectorAddKernel<<<blocks, threads>>>(d_vec_a, d_vec_b, d_result);
5
6  //Copy result back to host
7  cudaMemcpy(result, d_result, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
```

4.3.2 Constant Memory

Im Gegensatz zu Global Memory ist Constant Memory von Device- und Kernel-Funktionen nur lesbar, was bedeutet, das Variablen in diesem Speicher als Konstanten behandelt werden. Solche Variablen werden entweder im Code selbst definiert oder können zur Laufzeit von einer Host-Funktion mit `cudaMemcpyToSymbol()` festgelegt werden. `cudaMemcpyToSymbol()` unterscheidet sich von `cudaMemcpy()` dadurch, dass letzteres nur auf den Global Memory zugreifen kann, wohingegen `cudaMemcpyToSymbol()` nur auf den Constant Memory zugreifen kann.

```
cudaError_t cudaMemcpyToSymbol ( const void* symbol,
                                 const void* src,
                                 size_t count,
                                 size_t offset = 0,
                                 cudaMemcpyKind kind =
                                     cudaMemcpyHostToDevice )
```

Variablen im *Constant Memory* werden aufgrund der Tatsache, dass sie während der Kernel-Laufzeit nicht geändert werden, durch das *CUDA Runtime System* aggressiv gecached, was dazu führt, dass Zugriffe darauf wesentlich schneller sind, als Zugriffe auf Variablen im *Global Memory* [13]

Da bei unserer FWT-Implementierung verschiedene Werte über die Laufzeit hinweg konstant bleiben, können wir diese in den Constant Memory laden. im einzelnen sind das

```
1 __constant__ double d_constScales[1024];
2 __constant__ double d_constMorlet[2];
3 __constant__ float constant;
4 __constant__ double SQRTSQRT_PI = 1.33133536380039;
```

wobei `d_constScales[]` ein Vektor mit den einzelnen *Scales*, `d_constMorlet[]` Parameter für das Morlet-Wavelet, `constant` $\frac{2\pi}{Nh}$ und `SQRTSQRT_PI` $\sqrt[4]{\pi}$ sind.

Um nun Werte aus dem Host-Speicher in den Device-Speicher zu kopieren, verwenden wir die in Abschnitt 4.1 erstellten Funktionen:

Listing 4.13: Setter-Funktionen für Scales und Parameter

```
1  void setMorletParams(double *morlParams)
2  {
3      cudaMemcpyToSymbol(d_constMorlet, morlParams, sizeof(double) * 2);
4  }
5
6  void setConstScales(double *fScales, unsigned int nscales)
7  {
8      cudaMemcpyToSymbol(d_constScales, fScales, sizeof(double) * nscales);
9  }
10
11 void setConstant(int ndata, double h)
12 {
13     float constant_h = ( 2.0 * M_PI ) / ( ndata * h );
14     cudaMemcpyToSymbol(constant, &constant_h, sizeof(float));
15 }
```

4.3.3 Shared Memory

Shared Memory besitzt gegenüber den anderen Speichertypen einige Unterschiede. Zum einen werden Variablen, anders als bei Global Memory oder Constant Memory im Funktionskörper einer Kernel- oder Device-Funktion angelegt, zum anderen besitzen solche Variablen eine andere Sichtbarkeit für Threads. Zusätzlich dürfen `__shared__`-Variablen nicht mit einem Wert initialisiert werden.

Wird ein Kernel mit einer `__shared__`-Variable aufgerufen, erstellt CUDA für jeden Block eine eigene Kopie dieser Variable. Diese kann dann nur von Threads innerhalb ihres Blocks gesehen werden, die Variablen der anderen Blöcke bleiben unsichtbar.

Der Vorteil von Shared Memory gegenüber Global Memory ist der schnellere Lese-Zugriff darauf, was daher rührt, dass dieser sich im Gegensatz zu Global und Constant Memory direkt auf der GPU befindet. Das hat aber auch den Nachteil, dass die Größe des Speichers geringer ist. Verwendet wird Shared Memory üblicherweise dann, wenn häufiger auf Speicher im Global Memory zugegriffen werden muss, um den Zugriff auf diesen zu verringern.

Verwenden wir folgenden Kernel als Beispiel:

```
__global__ void kernel(int *data, int *vector)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < N)
    {
        for (int i = 0; i < ROUNDS; i++)
        {
            data[tid] += vector[threadIdx.x];
        }
    }
    __syncthreads();
}
```

Was dieser Kernel berechnet, ist hier wieder nebensächlich. Interessanter ist die Feststellung, dass dadurch, dass die Berechnung

`data[tid] += vector[threadIdx.x];`

in einer Schleife durchgeführt wird, mehrfach auf einen Eingabevektor `vector` im Global Memory zugegriffen werden muss, was durch die langsame Zugriffszeit vergleichsweise ineffizient ist. Für `N` = 1000 und `ROUNDS` = 10000, sowie mit dem Compilerparameter `-arch=sm_21` für die Compute Capability 2.1[4], benötigt dieser Kernel auf einer Geforce GTX 460m mit 448 Threads pro Block etwa 5,86 ms zur Bearbeitung.

Eine schnellere Alternative ist es, für jeden Block die verwendeten Werte aus `vector` in den Shared Memory zu laden. Dazu gehen wir in 2 Schritten vor:

1. Wir nutzen die Threads in jedem Block um die Werte in einen Vektor `sh_vec[]` im Shared Memory zu kopieren.

```
    __shared__ int  sh_vec[N];

//load to shared memory
    if (threadIdx.x < N) sh_vec[threadIdx.x] = vector[tid];
    __syncthreads();
```

4 siehe Kapitel 5.1

2. Die Berechnung greift nun nicht mehr auf den Vektor im Global Memory zu sondern auf `sh_data`.

```
//process data
    if (tid < N)
    {
        for (int i = 0; i < ROUNDS; i++)
        {
            data[tid] += sh_vec[threadIdx.x];
        }
    }
    __syncthreads();
```

Wenn wir die Laufzeit unter Verwendung des Shared Memory betrachten, sehen wir, dass diese mit 0,15 ms wesentlich schneller ist.

Da Variablen im Shared Memory für alle Threads in einem Block sichtbar sind, sollte darauf geachtet werden, dass die Threads synchron laufen. Das ist in diesem Beispiel nicht unbedingt nötig, da jeder Thread einen eigenen Wert bearbeitet. Sobald aber mehrere Threads den oder die selben Wert(e) bearbeiten, müssen diese synchronisiert werden.
Alternativ kann man auch mit dem Kernel-Aufruf festlegen, wie viel Speicher ein Vektor im Shared Memory verwenden darf. In diesem Fall wird der Vektor mit dem Modifier `extern` erweitert:

```
extern __shared__ int sh_data[];
```

Die Menge des benötigten Speichers in Bytes wird dann mit dem Kernel-Aufruf übergeben:

```
size_t mem = 1024 //Size in bytes

kernel<<<X , Y , mem >>>();
```

In diesem Fall also `mem` Bytes.

Man muss bei der Verwendung von Constant Memory und Shared Memory aber darauf achten, dass dieser begrenzt ist. Ein SM hat mit Compute Capability 2.0 beispielsweise 48 KB an Shared Memory zur Verfügung.

5 CUDA C Erweitert

Nachdem im letzten Kapitel die Grundlagen für die Programmierung mit CUDA C erklärt wurden, werden in diesem Kapitel unter anderem verschiedene API-Funktionen und verschiedene Eigenschaften der Parallelisierung mit CUDA besprochen.

5.1 Device-Eigenschaften auslesen

Wir sind bisher davon ausgegangen, dass, wenn wir ein CUDA-Programm ausführen, ein Device mit ausreichenden Fähigkeiten zur Verfügung steht. Soll der Code nun aber in einem unbekannten System ausgeführt werden, stellen sich zwei Fragen:

1. Steht ein CUDA-fähiges Device zur Verfügung und
2. Welche Eigenschaften besitzt es.

Die erste Frage lässt sich mit einer Funktion der CUDA Runtime API leicht beantworten. Um die Anzahl der verfügbaren Devices herauszufinden verwenden wir

```
cudaError_t cudaGetDeviceCount ( int* count )
```

Diese Funktion speichert die Anzahl der gefundenen Devices mit *compute capability* ≥ 1.0 in einer Variable `count` (Listing 5.1).

Listing 5.1: CUDA Device Properties - Beispiel 1

```
__host__ int function()
{
    int count;

    cudaGetDeviceCount(&count);

    printf("devices␣with␣compute␣capability␣>=␣1.0␣found:␣%d\n", count);

    // ...
```

Nachdem wir nun die Anzahl der verwendbaren Devices herausgefunden haben wollen wir wissen, welche Ressourcen auf den Devices verfügbar sind. Als Ressourcen werden unter anderem die Anzahl der Multiprozessoren und der verfügbare Speicher bezeichnet, aber auch Informationen wie die maximale Anzahl der Threads, die einem Multiprozessor gleichzeitig zugewiesen werden können, fallen darunter.

Für den Zugriff auf die Device-Informationen zur Laufzeit verwenden wir eine weitere API-Funktion namens

```
cudaError_t cudaGetDeviceProperties ( cudaDeviceProp* prop, int  device )
```

Diese liefert uns die Informationen über das Device mit der Nummer `device` in Form eines `structs` namens `cudaDeviceProp`[1]. Um sich nun beispielsweise bestimmte Informationen aller verfügbaren Devices ausgeben zu lassen, kann man mit einer Schleife über die Devices iterieren und mit `printf()` ausgeben lassen.

Listing 5.2: CUDA Device Properties - Beispiel 2

```
    // ...

//Get device properties
    cudaDeviceProp properties[count];

    for (int i = 0; i < count; i++)
    {
        cudaGetDeviceProperties( &(properties[i]), i );
    }

```

1 Der Aufbau von `cudaDeviceProp` findet sich in der *CUDA Runtime API Documentation* unter `http://docs.nvidia.com/cuda/cuda-runtime-api/index.html`. In diesem Kapitel werden nur einige Inhalte davon angesprochen

```
11 //print some informations
12     printf( "devices␣with␣compute␣capability␣>=␣1.0␣found:␣%d\n", count );
13
14     for ( int i = 0; i < count; i++ )
15     {
16         //Device name
17         printf( "Device␣%d:␣name␣=␣%s\n", i, properties[i].name );
18
19         //Device number of available multiprocessors
20         printf( "Device␣%d:␣number␣of␣multiprocessors␣=␣␣␣␣␣%d\n", i,
21                             properties[i].multiProcessorCount );
22
23         //Device compute capability version
24         printf( "Device␣%d:␣compute␣capability␣=␣%d.%d\n", i,
25                             properties[i].major, properties[i].minor );
26
27         //Device maximum allowed threads per block
28         printf( "Device␣%d:␣max␣threads␣per␣multiprocessor␣=␣%d\n", i,
29                             properties[i].maxThreadsPerBlock );
30
31         //Global memory size (in Bytes)
32         printf( "Device␣%d:␣Global␣memory␣in␣bytes␣=␣%ld\n", i,
33                              properties[i].totalGlobalMem );
34
35         //...
36     }
37     return 0;
38 }
```

Wird das Programm aus Listing 5.2 auf einer **Geforce GTX 460m** ausgeführt, erhalten wir folgende Ausgabe:

```
devices with compute capability >= 1.0 found: 1
Device 0: name = GeForce GTX 460M
Device 0: number of multiprocessors =   4
Device 0: compute capability = 2.1
Device 0: max threads per multiprocessor = 1024
Device 0: Global memory in bytes = 1609760768
```

Eine vollständige Auflistung der Device Properties findet sich in der Dokumentation der *CUDA Runtime API* [5].

Compute Capability

Oben wurde bereits die Compute Capability eines Device erwähnt. Dabei handelt es sich um einen Indikator für die Spezifikationen und Eigenschaften einer GPU ([3]). Beispielsweise stehen *Double-precision floating-point numbers* nur auf Devices zur Verfügung, die mindestens Version 1.3 unterstützen. Die vom Device unterstütze Version der *compute capability* lässt sich mit Hilfe der in den *Device Properties* enthaltenen Variablen `major` und `minor` für die Form `major.minor` auslesen. Einen Ausschnitt der Funktionen die für die verschiedenen Versionen der *compute capability* findet sich in Tabelle A.1.

In den folgenden Abschnitten werden wir sehen, wie man Device Properties und Compute Capability verwenden kann.

5.2 Thread-Verarbeitung

Wir haben in Kapitel 4 gesehen, dass ein Kernel-Aufruf eine vorab definierte Anzahl an Blöcken startet, die jeweils aus einer festgelegten Menge an Threads bestehen. Wenn wir zum Beispiel 5 Blöcke mit jeweils 20 Threads starten wollen, erhalten wir insgesamt $5 * 20 = 100$ Threads. Zur Berechnung dieser Threads werden die Blöcke nun den Multiprozessoren auf der GPU zugewiesen.

An dieser Stelle kommen die vorher erwähnten Device Properties und die Compute Capability zur Anwendung. Unglücklicherweise kann ein SM keine unbegrenzte Anzahl an Blöcken gleichzeitig verarbeiten, da seine Ressourcen beschränkt sind. Die vom Device unterstützte Compute Capability gibt eine maximale Menge an, die einem SM gleichzeitig zugewiesen werden können, auf Devices mit Compute Capability 2.1 sind das zum Beispiel 8 Blöcke, auf einem Device mit Compute Capability 3.0 können dagegen 16 Blöcke pro SM zugewiesen werden [3].
Es gibt allerdings noch weitere Faktoren, die die Anzahl der zuweisbaren Blöcke verringern können. Einem SM können nur eine bestimmte Menge an Threads gleichzeitig zugewiesen werden, die sich auf alle zugewiesenen Blöcke verteilen. Bei der GTX 460m aus dem vorherigen Abschnitt sind das 1536 Threads, was bedeutet, dass einem SM zum Beispiel 1 Block mit 1024 Threads, 2 Blöcke mit 768, 4 Blöcke mit 384 Threads oder auch 8 Blöcke mit 192 Threads zugewiesen werden können. Ein anderer begrenzender Faktor ist beispielsweise der verfügbare Shared Memory. Jeder SM kann mit Compute Capability

2.1 auf maximal 48 KB im Shared Memory zugreifen, die auf die zugewiesenen Blöcke aufgeteilt werden.
Wenn ein Block einem SM zur Bearbeitung zugewiesen wurde, wird er in kleinere Ausführungseinheiten, sogenannte *Warps* unterteilt. Im CUDA Modell besteht ein *Warp* aus 32 Threads[2], die gleichzeitig bearbeitet werden. Sind in einem Block weniger als 32 Threads, oder lässt sich die Anzahl nicht ohne Rest durch 32 teilen, kann ein *Warp* auch weniger als 32 Threads enthalten. Der SM bearbeitet alle Threads innerhalb eines *Warps* nach dem *single-instruction, multiple-thread*-Modell (SIMT)[15]. In diesem Modell wird an jeder Stelle, an der ein neuer Befehl ausgeführt werden soll, von einem sogenannten *thread scheduler* ein *Warp* ausgewählt und der jeweils aktuelle Befehl dieses *Warps* auf allen Threads ausgeführt. Innerhalb des *Warps* wird also für jeden Thread der selbe Befehl abgearbeitet.
Bei Verzweigungen, beispielsweise durch ein `if-then-else`-Konstrukt werden alle Zweige nacheinander abgearbeitet, was bedeuten kann, dass einige Threads in einem *Warp* teilweise inaktiv sind, wenn sie den jeweiligen Zweig nicht aufrufen [15].

Die **Geforce GTX 460m**, deren *Device Properties* wir in Abschnitt 5.1 haben ausgeben lassen, unterstützt *compute capability 2.1*, was bedeutet, dass einem SM maximal 8 Blöcke gleichzeitig zugewiesen werden können. Weiter kann man in Tabelle A.1 sehen, dass pro SM maximal 1536 Threads gleichzeitig bearbeitet werden können, was bedeutet, dass jeder SM bis zu $1536/32 = 48$ *Warps* parallel verarbeiten kann.

Verwenden wir hier wieder unsere FWT-Funktion. In der derzeitigen Form werden bei Aufruf des FWT-Kernels insgesamt $ndata * nscales$ Blöcke mit jeweils einem Thread gestartet, was zur selben Menge an *Warps* mit einem Thread führt.

```
1 dim3 dimBlock(1, 1);
2 dim3 dimGrid(ndata, nscales);
3
4 waveletKernel<<<dimGrid, dimBlock>>>( nscales,
5                                       ndata,
6                                       d_data,
7                                       d_result );
```

Gesucht ist also die optimale Größe eines Thread-Blocks, damit alle 4 SM optimal ausgelastet werden. Zur Vereinfachung gehen wir davon aus, dass alle anderen Beschränkungen, etwa der verfügbare Speicher, hier keine Rolle spielen.

2 Die Größe eines *Warps* ist in den *Device Properties* in `warpSize` gespeichert. Alternativ kann man in Kernel- und Device-Funktionen auch die eingebaute Variable `warpSize` verwenden[3]

Für die Thread- und Block-Anzahl besitzt unser Device folgende Beschränkungen (Tabelle A.1): Einem SM können maximal 8 Blöcke oder 1536 Threads (also 48 Warps) gleichzeitig zugewiesen werden, je nachdem was als erstes eintrifft. Eine optimale Auslastung haben wir also, wenn wir die 1536 Threads auf die 8 Blöcke aufteilen, sodass jeder Block $1536/8 = 192$ Threads verwendet.

Um diese Berechnung im Code möglichst allgemein zu halten, verwenden wir hier die *Device Properties*. Wir suchen zuerst die Menge der verfügbaren Devices mit `cudGetDeviceCount()` und ermitteln dann die jeweiligen *Device Properties* mit `cudaGetDeviceProp()` wie in Abschnitt 5.1 beschrieben. Wir benötigen den Wert für die maximale Anzahl an Threads pro SM (`maxThreadsPerMultiProcessor`) und der Blöcke, die ein SM bearbeiten kann. Für letzten gibt es in den *Device Properties* leider keinen Wert, deswegen müssen wir ihn anhand der *compute capability* (`major` reicht hier) bestimmen.

```
1  // number of available devices
2      int count = 0;
3      cudaGetDeviceCount(&count);
4
5  //device pproperties
6      cudaDeviceProp properties[count];
7
8      for (int i = 0; i < count; i++) cudaGetDeviceProperties( &(properties[i]), i );
9
10 // use device with ID 0
11     int deviceID = 0;
12
13 //get blocks per sm
14     int blocks_per_sm = 8;
15
16     if (properties[deviceID].major == 3) blocks_per_sm = 16;
17
18 //get threads per block
19     int threads_per_sm = properties[deviceID].maxThreadsPerMultiProcessor;
20     int threads_per_block = threads_per_sm / blocks_per_sm;
21
22 //create grid
23     int data_blocks = ( ndata + ( threads_per_block - 1 ) ) / threads_per_block;
24
25     dim3 dimBlock(threads_per_block, 1);
26     dim3 dimGrid(data_blocks , nscales);
27
28     waveletKernel<<<dimGrid, dimBlock>>>( nscales,
29                                           ndata,
30                                           d_data,
31                                           d_result );
```

Zu beachten ist hierbei, dass durch die Verwendung von

```
// use device with ID 0
    int deviceID = 0;
```

das erste verfügbare Device verwendet wird, ungeachtet der Leistung.

Mit

```
//get blocks per sm
    int blocks_per_sm = 8;

    if (properties[deviceID].major == 3) blocks_per_sm = 16;
    else blocks_per_sm = 8;
```

suchen wir die Anzahl der maximal an einen SM zuweisbaren Blöcke. Nach Tabelle A.1 sind das bis *compute capability 2.X* 8 Blöcke und ab Version 3.0 16 Blöcke. Um nun die Zahl der Threads in einem Block zu erhalten, teilen wir die Zahl der Threads pro SM durch die Zahl der Blöcke.

Nun müssen wir noch die Größe des Grids festlegen. Da unser Kernel so aufgebaut ist, dass er über die Y-Koordinate des Blocks im Grid die *Scale* auswählt und wir nach wie vor nur eindimensionale Blöcke erstellen wollen, reicht es aus, die Anzahl der Blöcke in X-Richtung zu ändern. Die Größe des Grids in Y-Richtung bleibt weiterhin `nscales`.

Die Anzahl der benötigten Blöcke in X-Richtung erhalten wir mittels

```
    int data_blocks = ( ndata + ( threads_per_block - 1 ) ) / threads_per_block;
```

Zuletzt wird die Blockgröße und Größe des Grids als `dim3` festgelegt und der Kernel aufgerufen.

```
    dim3 dimBlock(threads_per_block, 1);
    dim3 dimGrid(data_blocks , nscales);

    waveletKernel<<<dimGrid, dimBlock>>>( nscales,
                                          ndata,
                                          d_data,
                                          d_result );
```

Um dieses Beispiel einmal in Zahlen zu fassen: Gehen wir von einem Input-Vektor mit 1000 Werten und einem Vektor mit 200 *Scales* aus. Zuvor wurden bei Aufruf des Kernels $1000 * 200 = 200000$ Blöcke mit jeweils einem Thread erstellt. Da die SMs der **Geforce**

GTX 460m jeweils nur 8 Blöcke gleichzeitig bearbeiten können, würden so nur $8 * 4 = 32$ Threads auf allen 4 SMs gleichzeitig bearbeitet. Durch die Erhöhung der Blockgröße auf $1536/8 = 192$ Threads werden in X-Richtung

$$\lfloor (1000 + (192 - 1))/192 \rfloor = \lfloor 1191/192 \rfloor = \lfloor 6.203125 \rfloor = 6 \text{ Blöcke}$$

erstellt. Wir erhalten also ein Grid mit $6 * 200 = 1200$ Blöcken. Es können zwar nach wie vor nur 32 Blöcke gleichzeitig zugewiesen werden, allerdings werden jedem SM nun $8 * 192 = 1536$ Threads zur Berechnung übergeben, was der maximal möglichen Anzahl der GPU entspricht.

In der Realität übersteigt die Anzahl der einem SM zugewiesenen Threads die Menge der *Streaming Processors* (SP) meist, so dass immer nur ein kleiner Teil davon parallel bearbeitet werden kann. Dass dennoch so viele Threads zugewiesen werden, hängt mit der Bearbeitung von Operationen mit hoher Latenz, beispielsweise Speicherzugriffe, in CUDA zusammen [13]. Wartet ein *Warp* nun auf das Ergebnis einer solchen Operation, kann der SM einen anderen *Warp* auswählen, der keine Wartezeit auf offene Ergebnisse hat und diesen ausführen um die Wartezeit des ersten *Warps* zu überbrücken. Dies wird als *latency hiding* bzw. *latency tolerance* bezeichnet [13].

5.3 Verschiedene Funktionen der CUDA Runtime API

Wir haben bisher verschiedene Funktionen der CUDA Runtime API kennengelernt, beispielsweise `cudaMalloc()` und `cudaFree()` zur Allokation von Speicher im *Global Memory* oder die *Device Properties*. Im folgenden Abschnitt werden einige weitere interessante API-Funktionen vorgestellt, darunter das Auslesen von aufgetretenen Fehlern oder die Messung der Laufzeit eines Programms. Dieser Abschnitt soll verschiedene Funktionen nur kurz vorstellen. Ausführlichere Informationen über Anwendungsszenarien und Funktion bietet hier wieder der **CUDA C Programming Guide**[3] und die **CUDA Runtime API Documentation**[5]

5.3.1 Fehlerbehandlung

Bei Aufruf von API-Funktionen kann es durchaus zu Fehlern kommen, sei es, dass nicht genug Speicher zum Reservieren vorhanden ist (`cudaMalloc()` oder beispielsweise ein Pointer auf Host-Memory statt auf Device-Memory übergeben wurde. Viele API-Funktionen geben einen `cudaError_t`-Wert zurück, wenn sie aufgerufen werden.

Fehlertyp und ID	Bedeutung
cudaSuccess = 0	Kein Fehler aufgetreten
cudaErrorInvalidValue = 11	Ein oder mehrere Werte die als Parameter übergeben wurden sind illegal
cudaErrorInvalidDevicePointer = 17	Mindestens ein übergebener Device-Pointer ist ungültig
cudaErrorInvalidMemcpyDirection = 21	Die Kopierrichtung (Host -> Device o.ä.) ist nicht von CUDA spezifiziert

Tabelle 5.1: Mögliche Fehler, die bei der Ausführung von `cudaMemcpy()` auftreten können

5.3.2 Memory-Mapping und Zero-Copy Host Memory

Zusätzlich zur bekannten C-Funktion `malloc()` bietet die Runtime API eigene Funktionen um Speicher auf dem Host zu allozieren, beispielsweise

```
cudaError_t cudaMallocHost ( void** ptr, size_t size )
```

und

```
cudaError_t cudaHostAlloc ( void** pHost, size_t size, unsigned int  flags )
```

Beide Funktionen haben gemeinsam, dass der von ihnen reservierte Speicher *page-locked* ist, was bedeutet, dass das System ihn garantiert im RAM hält und nicht auf die Festplatte auslagert[23]. Pointer, die auf Speicher zeigen, der von `cudaMallocHost()` reserviert wurde, werden von CUDA als Device-Pointer behandelt, was es dem Device erlaubt, diese Speicherbereiche zu lesen und zu beschreiben.

Im Gegensatz zu `cudaMallocHost()` erlaubt es `cudaHostAlloc()` den so allozierten Speicher mit dem Parameter `flag` verschiedene Eigenschaften zuzuweisen. Als `flag` können folgende Werte verwendet werden[5].

- **`cudaHostAllocDefault`:** Mit dieser `flag` verhält sich `cudaAllocHost()` wie `cudaMallocHost()`, der erstellte Pointer kann also ohne weitere Bearbeitung vom Device verwendet werden.
- **`cudaHostAllocMapped`:** `cudaHostAlloc()` bildet den Speicher im Adressbereich der GPU ab. In diesem Fall wird der Pointer als Host-Pointer behandelt, zur Verwendung des Speichers auf der GPU muss also zuerst mit `cudaHostGetDevicePointer()` (siehe unten) ein Device-Pointer auf diesen Speicher erstellt werden.
- **`cudaHostAllocPortable`:** Der Speicher wird von allen CUDA-Prozessen als `page-locked` oder `pinned` behandelt, nicht nur im aktuellen
- **`cudaHostAllocWriteCombined`:** Der Speicher wird als *write-combined* alloziert. Der Lesezugriff der GPU auf diesen Speicher wird verbessert, allerdings kann die CPU diesen nicht effektiv lesen. Die Verwendung ist daher dann sinnvoll, wenn die CPU diesen Speicher nur beschreibt und die GPU nur daraus liest[23].

Wie bereits erwähnt, können bei der Verwendung von `cudaHostAlloc()` je nach verwendeter `flag` Host-Pointer erstellt werden. Um diesen Speicher auf der GPU zu verwenden wird mit

```
cudaError_t cudaHostGetDevicePointer ( void** pDevice,
 void* pHost,
 unsigned int flags )
```

ein Device-Pointer `pDevice` erstellt. `flags` ist zum Zeitpunkt von CUDA 5.5 noch nicht implementiert und muss daher `0` gesetzt werden[5]. Der erstellte Pointer kann dann beispielsweise von einem Kernel verwendet werden.

Um den von `cudaMallocHost()` und `cudaHostAlloc()` reservierten Speicher wieder freizugeben, wird analog zu `free()` in C oder `cudaFree()` für Device Memory

```
cudaError_t cudaFreeHost ( void* ptr )
```

verwendet. Da dieser Host-Speicher direkt vom Device gelesen und beschrieben werden kann und daher keine weiteren Kopiervorgänge benötigt werden, spricht man auch von *zero-copy host memory*. Sinnvoll ist die Anwendung vor allem auf Systemen, in denen sich GPU und CPU den Speicher teilen um unnötige Kopiervorgänge zu vermeiden. Auch in Fällen, in denen der Speicher nur einmal gelesen und/oder beschrieben wird, kann die Verwendung von Host-Memory Performance-Vorteile bringen[23].

Trotzdem sollte die Verwendung von *page-locked* Memory mit Bedacht erfolgen. Da der Speicher vom System bis zur Freigabe im RAM gehalten wird, steht für das restliche System weniger Speicher zur Verfügung, was dazu führen kann, dass andere Anwendungen aufgrund zu wenig Speicher Laufzeit- oder Performance-Probleme bekommen können[3, 23].

5.3.3 CUDA Streams

Ein Stream ist eine Abfolge von Operationen, die in einer bestimmten Reihenfolge ausgeführt werden. Standardmäßig werden alle Operationen im `0`-Stream ausgeführt.

Um einen eigenen Stream für bestimmte Operationen zu verwenden, wird zunächst mit

```
cudaError_t cudaStreamCreate ( cudaStream_t* pStream )
```

ein `cudaStream_t` erstellt

```
1 cudaStream_t stream;
2 cudaCreateStream( &stream );
```

Um einen Kernel einem Stream zuzuweisen, wird dieser nun mit

```
kernel<<<X, Y, 0, stream>>>();
```

aufgerufen, wobei `X` und `Y` die bereits bekannten Blöcke und Threads, `0` die Menge des Shared Memory, die dynamisch zugewiesen werden soll und `stream` der Stream ist, in dem der Kernel ausgeführt werden soll.
Um innerhalb eines Streams Daten zu kopieren wird an Stelle des bereits bekannten `cudaMemcpy()`

```
cudaError_t cudaMemcpyAsync ( void* dst,
                              const void* src,
                              size_t count,
                              cudaMemcpyKind kind,
                              cudaStream_t stream = 0 )
```

verwendet. Im Unterschied zu `cudaMemcpy()` wird `cudaMemcpyAsync()` asynchron ausgeführt, was bedeutet, dass es zu dem Zeitpunkt, an dem die Ausführung des Hauptprogramms fortgesetzt wird, nicht sicher ist, dass die Funktion bereits ausgeführt wurde. Allerdings ist innerhalb eines Streams garantiert, dass die nächste Operation erst nach Beendigung der vorherigen gestartet wird[23]. Ein weiterer Unterschied ist, dass `cudaMemcpyAsync()` nur von und in *page-locked* Memory kopieren kann.

Um einen nicht mehr benötigten Stream freizugeben wird

```
cudaError_t cudaStreamDestroy ( cudaStream_t stream )
```

verwendet.

Eine Anwendung für Streams ist beispielsweise die Verarbeitung eines Datensatzes, der zu groß ist, um ihn vollständig in den Speicher des Device zu kopieren. In diesem Fall kann man einem oder mehreren Streams immer wieder kleinere Ausschnitte des Datensatzes zuweisen, die auf das Device kopiert, dort verarbeitet und dann wieder zurück kopiert werden.
Da die Abarbeitung von Streams asynchron ist, muss nach dem letzten Aufruf sichergegangen werden, dass alle Operationen in einem Stream beendet wurden. Dazu wird

```
cudaError_t cudaStreamSynchronize ( cudaStream_t stream )
```

verwendet.

5.3.4 Laufzeitmessung mit Events

Zur Messung der Laufzeit eines Programms können mit der Runtime API sogenannte Events verwendet werden. Ein Event ist eine Art Zeitstempel, die im Code gesetzt werden kann[23].

Wir erstellen zunächst zwei `cudaEvent_t`, jeweils einen für Startpunkt und Endpunkt der Messung. Dazu verwenden wir

```
cudaError_t cudaEventCreate ( cudaEvent_t* event ) .
```

Um einen Messpunkt zu setzen wird die Funktion

```
cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t stream = 0 )
```

verwendet.
Da CUDA-Funktionen zum Teil asynchron ausgeführt werden, müssen wir sichergehen, dass das Event im letzten Messpunkt beendet wurde, bevor wir die Laufzeit berechnen können. Dazu wird

```
cudaError_t cudaEventSynchronize ( cudaEvent_t event )
```

aufgerufen. Diese Funktion wartet bis das Event `event` beendet wurde, bevor der Code weiter ausgeführt wird.
Die Berechnung der Laufzeit besteht aus einem weiteren Funktionsaufruf

```
cudaError_t cudaEventElapsedTime ( float* ms,
                                   cudaEvent_t start,
                                   cudaEvent_t end )
```

Diese Funktion berechnet die Laufzeit zwischen den Events `start` und `end` und speichert das Ergebnis in `ms`. Die Laufzeit wird in Millisekunden angegeben.

Werden Events nicht mehr benötigt, müssen sie mit

```
cudaError_t cudaEventDestroy ( cudaEvent_t event )
```

wieder freigegeben werden. Eine vollständige Messung sieht also so aus:

Listing 5.3: Laufzeitmessung mit CUDA-Events (vgl [23])

```
    cudaEvent_t start, stop;
    float elapsedTime;

//create Events
    cudaCreateEvent( &start );
    cudaCreateEvent( &stop );

//set starting point
    cudaEventRecord( start );

//execute something
    kernel<<<X,Y>>>();

//set endpoint
    cudaEventRecord( stop );

//Synchronize
    cudaEventSynchronize( stop );

//calculate and print execution time
    cudaEventElapsedTime( &elapsedTime, start, stop );
    printf( "Time: %3.1f ms\n", elapsedTime );

//Destroy events
    cudaDestroyEvent( start );
    cudaDestroyEvent( stop );
```

5.4 CUDA Bibliotheken

CUDA bietet neben der Runtime API noch weitere Funktionsbibliotheken. Eine davon ist die CUDA Driver API. Die Driver API bietet einen ähnlichen Umfang an Funktionen, erlaubt aber mehr Kontrolle über das Device und erfordert mehr Management. Während die Runtime API vieles automatisch erledigt, muss die Driver API im Programm beispielsweise mit `cuInit()` initialisiert werden. Ein weiterer Unterschied ist, dass die Module, die die Kernel- und Device-Funktionen enthalten manuell geladen werden müssen um einen Kernel auszuführen. Wir werden die Driver API an dieser Stelle nicht weiter verfolgen.

5.4.1 CUFFT

Eine der wichtigsten Anwendungen in der Signalverarbeitung ist die FFT (Fast Fourier Transform). Diese wird dazu verwendet, Daten vom Zeit- in den Frequenzbereich abzubilden. Für Grundlagen zu diesem Standardverfahren sei auf die Fachliteratur verwiesen, beispielsweise [22] oder [25]

CUDA bietet mit cuFFT[6] eine bereits fertige Implementierung der FFT für die GPU an. Um diese zu verwenden muss zum einen zum kompilieren der Parameter `-lcufft` angegeben werden, zum anderen muss im Code die Datei `cufft.h` eingebunden werden

```
#include <cufft.h>
```

Sind diese Voraussetzungen erfüllt, kann man die cuFFT-Bibliothek verwenden.

Um mit cuFFT eine FFT auszuführen sind drei Schritte notwendig (ohne das Kopieren der Daten miteinzubeziehen):

1. Einen Plan erstellen
2. die FFT mit dem erstellten Plan ausführen
3. den Plan freigeben

Je nachdem, welche Dimension die Eingabewerte haben, verwendet man zur Erstellung des FFT-Plans

```
cufftResult cufftPlan1d( cufftHandle *plan,
                         int nx,
                         cufftType type,
                         int batch )

cufftResult cufftPlan2d( cufftHandle *plan,
                         int nx,
                         int ny,
                         cufftType type )
```

```
cufftResult cufftPlan3d( cufftHandle *plan,
                         int nx,
                         int ny,
                         int nz,
                         cufftType type )
```

wobei `plan` ein `cufftHandle`-Objekt, `nx`, `ny` und `nz` die Größe des Input-Vektors in x-, y- und z-Richtung und `type` die Art der Transformation ist. `type` ist dabei ein Wert aus dem `enum cufftType_t` (siehe Listing 5.4).

Listing 5.4: Aufbau `cufftType_t` (Quelle: [6])

```
typedef enum cufftType_t {
    CUFFT_R2C = 0x2a,  // Real to complex (interleaved)
    CUFFT_C2R = 0x2c,  // Complex (interleaved) to real
    CUFFT_C2C = 0x29,  // Complex to complex (interleaved)
    CUFFT_D2Z = 0x6a,  // Double to double-complex (interleaved)
    CUFFT_Z2D = 0x6c,  // Double-complex (interleaved) to double
    CUFFT_Z2Z = 0x69   // Double-complex to double-complex (interleaved)
} cufftType;
```

Die FFT wird nun abhängig vom erstellten Plan mit folgenden Funktionen durchgeführt:

```
cufftResult cufftExecC2C( cufftHandle plan,
                          cufftComplex *idata,
                          cufftComplex *odata,
                          int direction );

cufftResult cufftExecZ2Z( cufftHandle plan,
                          cufftDoubleComplex *idata,
                          cufftDoubleComplex *odata,
                          int direction);
```

Diese Funktionen führen einen *single-* bzw. *double-precision complex-to-complex*-Plan aus. Input- wie Output-Vektoren bestehen aus komplexen Werten.

```
cufftResult cufftExecR2C( cufftHandle plan,
                          cufftReal *idata,
                          cufftComplex *odata );

cufftResult cufftExecD2Z( cufftHandle plan,
                          cufftDoubleReal *idata,
                          cufftDoubleComplex *odata );
```

Erstellt man einen *single-precision* (*double-precision*) *real-to-complex*-Plan, werden `cufftExecR2C()`, bzw. `cufftExecD2Z` verwendet. Diese Funktionen implizieren eine normale FFT, also von Zeit- in Frequenz-Bereich[6].

```
cufftResult cufftExecC2R( cufftHandle plan,
                          cufftComplex *idata,
                          cufftReal *odata );

cufftResult cufftExecZ2D( cufftHandle plan,
                          cufftComplex *idata,
                          cufftReal *odata );
```

Mit einem *single-precision* (*double-precision*) *complex-to-real*-Plan werden `cufftExecC2R()` bzw. `cufftExecZ2D()` verwendet, was normalerweise der inversen FFT entspricht.

Mit Unterschied des Parameters `direction` für die Funktionen, die innerhalb der Komplexen Zahlen bleiben, haben alle sechs Funktionen die gleichen Parameter. `plan` ist das im vorherigen Schritt erstellte Plan, `idata` der Input-Vektor (komplex oder real) und `odata` der Output-Vektor (ebenfalls komplex bzw. real). `cufftExecC2C()` und `cufftExecZ2Z()` haben noch einen weiteren Parameter, da sie sowohl für FFT als auch für inverse FFT verwendet werden, wenn sowohl Input als auch Output ein Vektor mit komplexen Werten ist. `direction` kann entweder `CUFFT_FORWARD` für die FFT oder `CUFFT_INVERSE` für die inverse FFT sein.

Nach Ausführung der FFT wird der erstellte Plan mit

```
cufftResult cufftDestroy(cufftHandle plan)
```

wieder freigegeben.

5.4.2 Weitere Bibliotheken

Neben der FFT-Bibliothek cuFFT bietet CUDA noch weitere Funktionsbibliotheken an, die hier nur kurz vorgestellt werden.

- **CUBLAS:** *cuBLAS* ist eine Implementierung der *Basic Linear Algebra Subprograms*[1] für NVIDIA-GPUs. BLAS stellt grundlegende Vektor-Matrix-Operationen, beispielsweise Skalar-Vektor- oder Vektor-Vektor-, Vektor-Matrix- oder Matrix-Matrix-Operationen zur Verfügung[1].
- **CURAND:** Die *cuRAND*-Bibliothek dient zur Generierung von Pseudozufallszahlen[7].
- **CUSPARSE:** Enthält Operationen für den effizienten Umgang mit dünn-besetzten Matrizen und Vektoren[8].
- **NPP:** *NPP* (NVIDIA Performance Primitives) ist eine Sammlung von Funktionen zur Signal-, Bild- und Videobearbeitung auf der GPU[19].
- **Thrust:** Template-Bibliothek, die auf der C++-STL (Standard Template Library) basiert[27]. Thrust stellt parallele Vektor-Typen (für Host und Device) und Algorithmen, beispielsweise Sortier-Algorithmen, zur Verfügung.

6 Vergleich von CUDA und OpenCL

Eine andere Möglichkeit, parallelisierten Code zu erstellen ist die **OpenCL**-API[20]. Diese hat den Vorteil, dass sie im Gegensatz zu CUDA herstellerunabhängig ist und neben GPUs auch CPUs und andere Beschleuniger zur Parallelisierung unterstützt[13]. In diesem Kapitel untersuchen wir die Unterschiede und Gemeinsamkeiten von CUDA und OpenCL und die Portierbarkeit von CUDA-Code.

6.1 Hintergrund zu OpenCL

OpenCL ist eine *cross-platform parallel computing API*, die auf C basiert[13]. Wie bereits erwähnt, ist sie herstellerunabhängig. Das bedeutet, dass damit erstellter Code (theoretisch) unabhängig vom verwendeten Device-Typ und -Hersteller verwendet werden kann. In der Realität sieht dies jedoch etwas anders aus. Viele OpenCL-Features sind optional, was zur Folge hat, dass sie nicht zwangsläufig von jedem Device unterstützt werden[13]. Will man den Code möglichst unabhängig gestalten, muss man auf diese Features verzichten, was wiederum zu Performance-Einbußen führen kann, oder ausführlich überprüfen, welche Features unterstützt werden, was zu längerem und komplexerem Code führen kann.

OpenCL ist ein offener Standard, der unter anderem von AMD, Intel oder Nvidia implementiert wird[9, 26].

6.2 Programming Model

Wie schon bei CUDA besteht ein OpenCL-System aus einem Host und einem oder mehreren Devices, wobei ein Device hier nicht unbedingt eine GPU sein muss, im weiteren Verlauf

aber als solche verwendet wird. Ein Device besteht aus mehreren *Computing Units* (CUs), die den *Streaming Multiprocessors* (SMs) im CUDA-Modell entsprechen. Jede CU besteht wiederum aus mehreren *Processing Elements* (PEs), das CUDA-Äquivalent hierzu sind die *Streaming Processors* (SPs).

Grundsätzlich sind sich CUDA- und OpenCL-Modell sehr ähnlich, beispielsweise bei den Speichertypen. OpenCL kennt *Global Memory*, *Constant Memory*, *Local Memory* (CUDA-Äquivalent: Shared Memory) und *Private Memory* (Register und Local Memory in CUDA). Die Speichertypen werden wie ihre CUDA-Äquivalente verwendet, so ist der OpenCL-Constant Memory vom Device aus nur-lesbar.

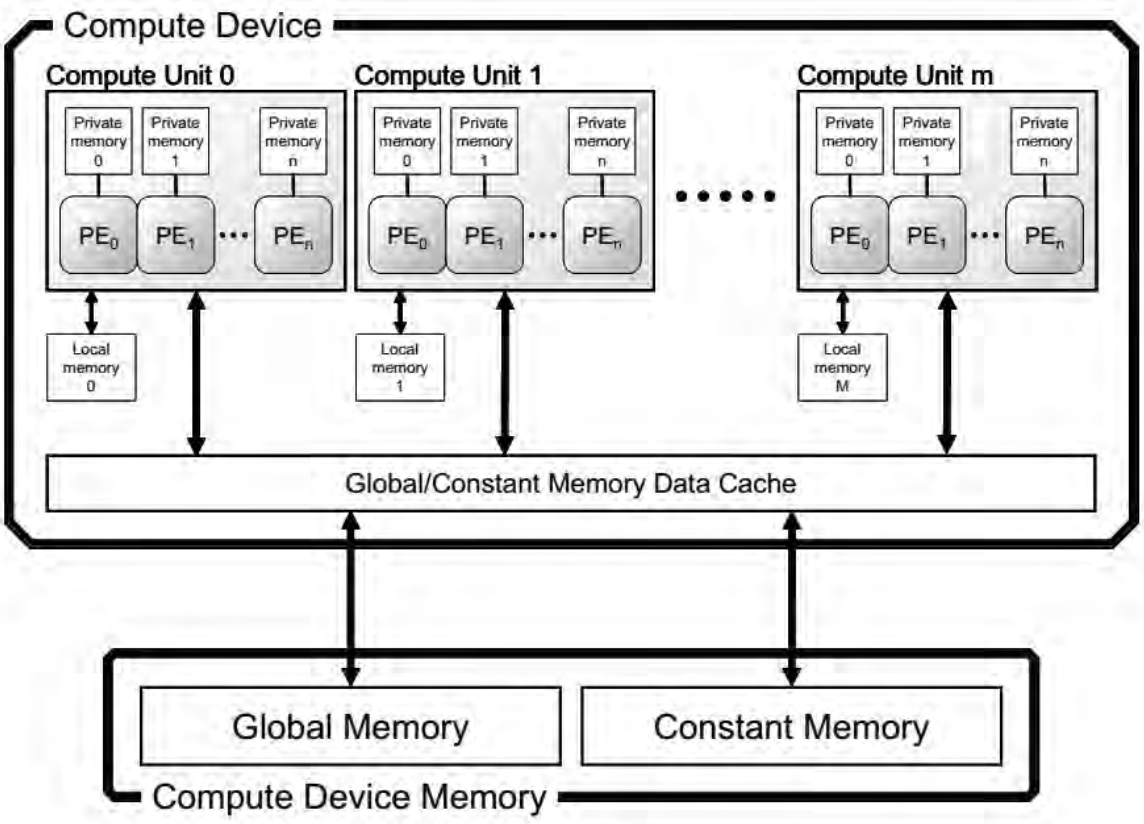

Abbildung 6.1: OpenCL-Architektur (Quelle: [14])

Auch die Parallelisierung von Kernel-Funktionen ist ähnlich aufgebaut. Jeder Kernel wird in einem *work item* (CUDA-Äquivalent: Thread) ausgeführt. Work items können in mehrdimensionalen *work groups* (Blöcken) zusammengefasst werden. Die Gesamtheit aller work items wird als *NDRange* (Grid) bezeichnet (Vergleiche auch Tabelle 6.1)

OpenCL	CUDA
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Register und Local Memory
Computing Units	Streaming Multiprocessors
Processing Elements	Streaming Processors
Work Item	Thread
Work Group	Block
NDRange	Grid

Tabelle 6.1: OpenCL-Terme und ihre CUDA-Äquivalenz [13]

Um Devices zur Berechnung eines Kernels zu verwenden werden diese in sogenannten *Contexts* gebündelt. Ein Context besteht aus einem Speicherbereich, der den gesamten adressierbaren Speicher der beinhalteten Devices umfasst. Speicherobjekte, die innerhalb eines Contexts verwendet werden, sind für alle Devices in diesem Context sichtbar[13, 16].

Um einem Device innerhalb eines Contexts eine Aufgabe, beispielsweise einen Kernel, zuzuweisen, wird eine *Command Queue* verwendet, die mit einem CUDA-Stream vergleichbar ist. Sobald das Device eine Aufgabe erledigt hat, wird die nächste in der Queue von oben verendet und ausgeführt.

6.3 Vektoraddition in CUDA und OpenCL

Vergleichen wir den Code, der mit der CUDA API und der OpenCL API erstellt wird. Wir werden an dieser Stelle noch einmal auf die CUDA Driver API zurückkommen, da diese, im Vergleich zur Runtime API, mehr Ähnlichkeiten zu OpenCL aufweist.

Runtime API

Für unsere Vektoraddition verwenden wir den leicht angepassten Kernel aus A.3 (Listing 6.1).

Listing 6.1: Vektoraddition-Kernel CUDA

```
extern "C" __global__ void VecAdd_kernel(int *vector_a, int *vector_b, int *result, int size)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < size)
    {
        result[tid] = vector_a[tid] + vector_b[tid];
    }

    __syncthreads();
}
```

Verwenden wir zum Aufrufen nun die Runtime API reichen uns die folgenden Schritte:

1. Erstellen von Device-Pointern
2. Speicherreservierung auf dem Device
3. Kopieren der Daten vom Host auf das Device
4. Ausführung des Kernels
5. Zurück kopieren des Ergebnisses auf den Host
6. Allozierten Speicher auf dem Device wieder freigeben

Listing 6.2: Vektoraddition mit CUDA Runtime API

```
void vectorAdd_RT( int *vec_a, int *vec_b, int *vec_result)
{
// Device pointer [1]
    int *d_vec_a;
    int *d_vec_b;
    int *d_result;

// Memory Allocation on Device [2]
    cudaMalloc((void **) &d_vec_a, SIZE*sizeof(int));
    cudaMalloc((void **) &d_vec_b, SIZE*sizeof(int));
    cudaMalloc((void **) &d_result, SIZE*sizeof(int));

// Copy data to device [3]
    cudaMemcpy(d_vec_a, vec_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_vec_b, vec_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);

// Run Kernel [4]
    size_t threads = 192;
    size_t blocks = (SIZE + threads - 1) / threads;
    VecAdd_kernel<<< blocks , threads>>>(d_vec_a, d_vec_b, d_result, SIZE);
```

```
21
22 // Copy result back to host [5]
23     cudaMemcpy(vec_result, d_result, SIZE * sizeof(int), cudaMemcpyDeviceToHost);
24
25 // Free device memory [6]
26     cudaFree(d_vec_a);
27     cudaFree(d_vec_b);
28     cudaFree(d_result);
29 }
```

Die Driver API und die OpenCL API dagegen arbeiten auf einem niedrigeren Level, was zur Folge hat, man mehr Code für die gleiche Funktionalität benötigt, da verschiedene Aufgaben, die die Runtime API automatisch erledigt, selbst aufgerufen werden müssen.

OpenCL API

Die OpenCL API erfordert etwas mehr Code, um die obige Vektoraddition auszuführen. Sehen wir uns zunächst den Kernel der OpenCL-Version an (Listing 6.3).Der Kernel in einem zusätzlichen Source-File *Kernel.cl* angelegt.

Listing 6.3: Kernel.cl

```
1  __kernel void VectorAddKernel(__global int *vector_a,
2                                __global int *vector_b,
3                                __global int *result,
4                                int size)
5  {
6      int tid = get_global_id(0);
7
8      if (tid < size)
9      {
10         result[tid] = vector_a[tid] + vector_b[tid];
11     }
12
13     barrier(0);
14 }
```

Wie wir sehen, halten sich die Unterschiede zwischen CUDA und OpenCL hier in Grenzen. OpenCL verwendet anstelle des `__global__`-Modifiers für Kernel-Funktion `__kernel`. Außerdem erhalten die im Global Memory liegenden Parameter den Modifier `__global`. Ein weiterer Unterschied ist die Methode zur Ermittelung der ID des jeweiligen Threads. Mit CUDA haben wir für die globale ID folgenden Term verwendet:

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

OpenCL bietet hier eine einfachere Möglichkeit. `get_global_id(x)` mit $x \in 0, 1, 2$ liefert uns die x, y und z Koordinate im Grid. Zuletzt synchronisieren wir die Threads ebenfalls am Ende der Berechnung. Mit CUDA verwenden wir hier `__syncthreads()`, OpenCL verwendet dazu `barrier()`.

Etwas komplizierter ist hier der Code, der zum Aufruf dieses Kernels verwendet wird.

1. Einlesen des Sourcecode-Codes des Kernels.
2. Device-Pointer anlegen.
3. Systeminformationen erhalten. Um einen Context zum Ausführen des Kernels zu erstellen, benötigen wir die Anzahl der verwendbaren Devices im System.
4. Context erstellen.
5. Command Queue erstellen
6. Programm erstellen. An dieser Stelle verwenden wir den Sourcecode-Code, den wir in Punkt [1] eingelesen haben um ein OpenCL-Programm zu erstellen.
7. Programm kompilieren. Nachdem wir den Sourcecode-Code zur Laufzeit laden, müssen wir ihn kompilieren um ihn zu verwenden. OpenCL liefert dazu bestimmte Funktionen mit.
8. Kernel erstellen. Aus dem im vorherigen Schritt kompilierten Programm wird nun eine Kernel-Funktion geladen.
9. Speicher im Device reservieren.
10. Kopieren der Daten vom Host auf das Device.
11. Parameter für den Kernel festlegen. Wir rufen den Kernel hier nicht direkt auf, deswegen können wir die Parameter nicht übergeben und müssen sie vorab festlegen.
12. Kernel ausführen.
13. Zurück-kopieren des Ergebnisses auf den Host.
14. Allozierten Speicher auf dem Device wieder freigeben.

15. Aufräumen. In diesem Schritt wird unter anderem der Context aufgelöst.

Listing 6.4: Vektoraddition-Kernel-Aufruf mit OpenCL

```
1  void vectorAdd_OpenCL( int *vec_a, int *vec_b, int *vec_result)
2  {
3  //Init [1]
4      char *src = NULL;
5      size_t src_size;
6      readFile( "Kernel.cl", src, &src_size );
7
8  // Device pointer [2]
9      cl_mem dev_a;
10     cl_mem dev_b;
11     cl_mem dev_result;
12
13 // get system information [3]
14     cl_platform_id platform;
15     cl_device_id device;
16     clGetPlatformIDs(1, &platform, NULL);
17     clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
18
19 // Create context [4]
20     cl_context context;
21     context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
22
23 //Create command queue [5]
24     cl_command_queue queue;
25     queue = clCreateCommandQueue( context,
26                                   device,
27                                   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
28                                   NULL );
29
30 // Create module from binary file [6]
31     cl_program program;
32     program = clCreateProgramWithSource(context, 1, (const char**) &src, &src_size, NULL);
33
34 // Compile source [7]
35     const char options[] = "-Werror -cl-std=CL1.1";
36     clBuildProgram(program, 1, &device, options, NULL, NULL);
37
38 // Create kernel [8]
39     cl_kernel kernel;
40     kernel = clCreateKernel(program, "VectorAddKernel", NULL);
41
42 //Memory Allocation on Device [9]
43     dev_a = clCreateBuffer( context, CL_MEM_READ_WRITE, SIZE * sizeof(int), NULL, NULL );
44     dev_b = clCreateBuffer( context, CL_MEM_READ_WRITE, SIZE * sizeof(int), NULL, NULL );
45     dev_result = clCreateBuffer( context, CL_MEM_READ_WRITE, SIZE * sizeof(int), NULL, NULL );
46
47 //Copy vectors to device [10]
48     clEnqueueWriteBuffer( queue, dev_a, CL_TRUE, 0, SIZE *sizeof(int), vec_a, 0, NULL, NULL );
49     clEnqueueWriteBuffer( queue, dev_b, CL_TRUE, 0, SIZE *sizeof(int), vec_b, 0, NULL, NULL );
```

```
50
51 //Set Args [11]
52     cl_int size = SIZE;
53     clSetKernelArg( kernel, 0, sizeof(cl_mem), &dev_a );
54     clSetKernelArg( kernel, 1, sizeof(cl_mem), &dev_b );
55     clSetKernelArg( kernel, 2, sizeof(cl_mem), &dev_result );
56     clSetKernelArg( kernel, 3, sizeof(cl_int), &size );
57
58 //Run Kernel [12]
59     size_t gridDim[2] = {SIZE, 1};
60     size_t blockDim[2] = {1,1};
61     clEnqueueNDRangeKernel( queue, kernel, 2, NULL, gridDim, blockDim, 0, NULL, NULL );
62
63 //Copy result back to host [13]
64     clEnqueueReadBuffer( queue,
65                          dev_result,
66                          CL_TRUE,
67                          0,
68                          SIZE * sizeof(int),
69                          vec_result,
70                          0,
71                          NULL ,
72                          NULL );
73
74 //Free device memory [14]
75     clReleaseMemObject(dev_a);
76     clReleaseMemObject(dev_b);
77     clReleaseMemObject(dev_result);
78
79 //Cleanup [15]
80     clReleaseContext(context);
81 }
```

Im Vergleich zur Runtime API sind also mehr Funktionsaufrufe notwendig, was sich auch in der Länge des Codes widerspiegelt. Die Hauptunterschiede liegen unter anderem im Erstellen des Contexts, dem Laden und Kompilieren des Kernels zur Laufzeit und im Aufruf dieses Kernels. Die Runtime API arbeitet im Vergleich zur OpenCL API aber auf einem höheren Abstraktionslevel, so dass viele der bei OpenCL notwendigen Schritte automatisiert erledigt werden. Wir werden gleich anhand der CUDA Driver API sehen, dass sich die Unterschiede zu OpenCL aber in Grenzen halten.

CUDA Driver API

Wir verwenden wieder den Kernel aus Listing 6.1, mit dem Unterschied, dass dieser jetzt mit dem Compiler in ein `ptx`-Format (Kernel.ptx) gebracht wurde[1].

Bei Verwendung der Driver API werden folgende Schritte benötigt um einen Kernel zu auszuführen[2].

1. Driver API initialisieren. Um die Driver API zu verwenden, muss diese erst initialisiert werden.
2. Device-Pointer anlegen.
3. Systeminformationen erhalten.
4. Device finden. Auch mit der Driver API wird ein Context erstellt, dem ein Device zugewiesen wird.
5. Context erstellen.
6. Ein Modul aus dem `ptx`-File mit dem Kernel erstellen.
7. Kernel erstellen. Ähnlich wie in OpenCL wird auch hier die Kernel-Funktion, die wir verwenden wollen, aus einem Modul geladen.
8. Speicher im Device reservieren.
9. Kopieren der Daten vom Host auf das Device.
10. Parameter für den Kernel festlegen. Wieder analog zu OpenCL, mit dem Unterschied, dass hierfür keine Funktion verwendet wird, sondern die Parameter mit dem Kernel-Aufruf übergeben werden.
11. Kernel ausführen. Auch hier wird der Kernel nicht direkt aufgerufen, sondern durch eine Funktion, der die Argumente, die Größe und der Aufbau des Grids, sowie die Parameter des Kernels übergeben werden.
12. Zurück-kopieren des Ergebnisses auf den Host.
13. Allozierten Speicher auf dem Device wieder freigeben.
14. Aufräumen. Darunter fällt auch hier das Auflösen des Contexts.

1 Siehe auch Abschnitt 2.2
2 Hinweis: Ein Teil des Codes wurde hier aus [3] übernommen

Listing 6.5: Vektoraddition-Kernel-Aufruf Driver API

```
1  void vectorAdd_Driver( int *vec_a, int *vec_b, int *vec_result)
2  {
3  // Init
4      cuInit(0);
5
6  // Device Pointer
7      CUdeviceptr d_A;
8      CUdeviceptr d_B;
9      CUdeviceptr d_C;
10
11 // get system information
12     int deviceCount = 0;
13     cuDeviceGetCount(&deviceCount);
14
15 // get Device
16     CUdevice cuDevice;
17     cuDeviceGet(&cuDevice, 0);
18
19 // Create context
20     CUcontext cuContext;
21     cuCtxCreate(&cuContext, 0, cuDevice);
22
23 // Create module from binary file
24     CUmodule cuModule;
25     cuModuleLoad(&cuModule, "Kernel.ptx");
26
27 // Get function handle from module
28     CUfunction VecAdd_kernel;
29     cuModuleGetFunction(&VecAdd_kernel, cuModule, "VecAdd_kernel");
30
31 // Memory Allocation on Device
32     cuMemAlloc(&d_A, SIZE * sizeof(int) );
33     cuMemAlloc(&d_B, SIZE * sizeof(int) );
34     cuMemAlloc(&d_C, SIZE * sizeof(int) );
35
36 // Copy vectors to device
37     cuMemcpyHtoD(d_A, vec_a, SIZE * sizeof(int) );
38     cuMemcpyHtoD(d_B, vec_b, SIZE * sizeof(int) );
39
40 // Set Args
41     int size = SIZE;
42     void* args[] = { &d_A, &d_B, &d_C , &size };
43
44 // Run Kernel
45     int threadsPerBlock = SIZE;
46     int blocksPerGrid = 1;
47     cuLaunchKernel(VecAdd_kernel, blocksPerGrid, 1, 1, threadsPerBlock, 1, 1, 0, 0, args, 0);
48
49 // Copy result back to host
50     cuMemcpyDtoH(vec_result, d_C, SIZE * sizeof(int) );
51
```

```
52 // Free device memory
53     cuMemFree( d_A );
54     cuMemFree( d_B );
55     cuMemFree( d_C );
56 
57 // Cleanup
58     cuCtxDestroy(cuContext);
59 }
```

Fazit

Wir sehen also, dass sich die Unterschiede zwischen der Driver API und der OpenCL API durchaus in Grenzen halten. Dadurch, dass die Programmiermodelle von CUDA und OpenCL sich ähneln, sollte es einen vergleichsweise geringen Aufwand erfordern, ein CUDA-Programm in OpenCL umzusetzen.

7 Zusammenfassung

Sinnvoll ist die Verwendung von GPUs für Berechnungen vor allem dann, wenn wir eine gut parallelisierbare Aufgabe vorliegen haben, die von der Verwendung einer großen Anzahl Threads profitiert. Ein Beispiel hierfür ist die im Rahmen dieser Arbeit verwendete FWT.
Wir haben zur Implementierung der FWT CUDA C verwendet, eine Sprache, die für NVIDIA Grafikkarten entwickelt wurde und C um einige wenige Elemente erweitert. Dazu gehören beispielsweise Modifier für Funktionen, die auf der Grafikkarte ausgeführt werden, oder für Variablen im Grafikspeicher.
Parallelisierter Code wird in sogenannten Kernel-Funktionen ausgeführt, bei deren Aufruf angegeben wird, wie viele Threads diese ausführen sollen, was dazu führt, dass alle Threads in einem Aufruf den gleichen Code ausführen. Threads werden in einem sogenannten Grid ausgeführt und dort in mehrere gleich große Blöcke eingeteilt. Damit nun verschiedene Daten verarbeitet werden, kann man die Koordinaten der Blöcke im Grid und die Koordinaten der Threads in einem Block verwenden um einen eindeutigen Thread-Index zu erstellen.
Einem CUDA-Programmierer stehen mehrere Speichertypen zur Verfügung. Zum einen der Global Memory zum anderen auch spezielle Typen, beispielsweise den nur-lesbaren Constant Memory, der besonders aggressiv gecached wird oder der Shared Memory, in dem von jeder Variable für jeden Block in einem Grid eine Kopie angelegt wird. Jeder Speichertyp hat Vor- und Nachteile, sodass man die besten Ergebnisse meistens mit einer Kombination der Speichertypen erzielen kann.
Die Eigenschaften des ausführenden Device sind in den sogenannten Device Properties enthalten. Diese können zur Laufzeit ausgelesen werden und liefern Informationen, beispielsweise über die Menge des verfügbaren Global Memory, oder die vom Device unterstützte Version der Compute Capability. Letztere dient als Indikator für die Features und Spezifikationen, die eine Device-Modellreihe beherrscht.
CUDA bietet zwei APIs zur Erstellung von GPU-Programmen, zum einen die Runtime

API, die eine hohe Abstraktionsebene bietet und daher einfacher zu verwenden ist und die, mit OpenCL vergleichbare Driver API, die zwar umfangreicheren Code zur Folge hat, aber auch eine tiefer gehende Kontrolle über das Device erlaubt.

Abschließend betrachtet bietet CUDA durch die Tatsache, dass eine Erweiterung der Programmiersprache C verwendet wird, eine einfache Möglichkeit, Code zu schreiben, der die Vorteile einer Grafikkarte nutzen kann. Der Nachteil dabei ist allerdings, dass CUDA nur für NVIDIA-Grafikkarten entwickelt wurde. Wird beispielsweise ein heterogenes System mit AMD- und NVIDIA-Grafikkarten verwendet, bietet sich daher die Verwendung von OpenCL als Herstellerübergreifende API an.

A Anhang

A.1 Quelltexte

A.1.1 Testprogramm für Threadsynchronisation

Listing A.1: constr_sync.cu Kernels

```
#include <stdio.h>

#define N 100

//Kernel without thread synchronisation
__global__ void kernel(int *data, int *result, int ndata)
{
    int tid = threadIdx.x;

// double each value in data
    if ( tid < ndata) data[tid] = data[tid] * 2;

// calculate
    if ( tid < (ndata - 1) ) result[tid] = data[tid] + data[tid + 1];
    else if ( tid == (ndata - 1) ) result[tid] = data[tid] + data[0];
}

//Kernel with thread synchronisation
__global__ void kernel2(int *data, int *result, int ndata)
{
    int tid = threadIdx.x;

// double each value in data
    if ( tid < ndata) data[tid] = data[tid] * 2;
    __syncthreads();

// calculate
    if ( tid < (ndata - 1) ) result[tid] = data[tid] + data[tid + 1];
    else if ( tid == (ndata - 1) ) result[tid] = data[tid] + data[0];
    __syncthreads();
}
```

Listing A.2: constr_sync.cu Main

```
int main()
{
    int a[N];
    int result[N];
    int result2[N];

    int *d_a;
    int *d_result;
    int *d_a2;
    int *d_result2;

// Fill a
    for (int ii = 0; ii < N; ii++)
    {
        a[ii] = ii;
    }

//Allocate Memory on Device
    cudaMalloc((void **) &d_a, N*sizeof(int));
    cudaMalloc((void **) &d_result, N*sizeof(int));
    cudaMalloc((void **) &d_a2, N*sizeof(int));
    cudaMalloc((void **) &d_result2, N*sizeof(int));

// Use kernel for calculation, without ThreadSync
    cudaMemcpy(d_a,a, N*sizeof(int), cudaMemcpyHostToDevice);
    kernel<<<1,N>>>(d_a,d_result,N);
    cudaMemcpy(result, d_result, N*sizeof(int), cudaMemcpyDeviceToHost);

// Use kernel for calculation, with ThreadSync
    cudaMemcpy(d_a2,a, N*sizeof(int), cudaMemcpyHostToDevice);
    kernel2<<<1,N>>>(d_a2,d_result2,N);
    cudaMemcpy(result2, d_result2, N*sizeof(int), cudaMemcpyDeviceToHost);

//Compare results
    for (int i = 0; i < N; i++)
    {
        printf("kernel( %d ) = %d, should be: %d\n", i, result[i], result2[i]);
    }

// Free Memory on Device
    cudaFree(d_a);
    cudaFree(d_result);
    cudaFree(d_a2);
    cudaFree(d_result2);

    return 0;
}
```

A.1.2 Vektoraddition

Listing A.3: Addition zweier Vektoren (siehe auch [23])

```
#include <stdlib.h>
#include >stdio.h>

#define SIZE 4

__global__ void VectorAddKernel(int *vector_a, int *vector_b, int *result)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x

    if ( tid < SIZE ) result[tid] = vector_a[tid] + vector_b[tid];
}

int main()
{
    int vec_a[SIZE];
    int vec_b[SIZE];
    int result[SIZE];

//Fill vectors
    for (int i = 0; i < SIZE; i++)
    {
        vec_a[i] = i;
        vec_b[i] = 2 * i + 1;
    }

// Device pointer
    int *d_vec_a;
    int *d_vec_b;
    int *d_result;

//Memory Allocation on Device
    cudaMalloc((void **) &d_vec_a, SIZE*sizeof(int));
    cudaMalloc((void **) &d_vec_b, SIZE*sizeof(int));
    cudaMalloc((void **) &d_result, SIZE*sizeof(int));

//Copy vectors to device
    cudaMemcpy(d_vec_a, vec_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_vec_b, vec_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);

//Run Kernel
    int threads = 192;
    int blocks = (int) ( (SIZE + threads - 1) / threads );
    VectorAddKernel<<<blocks, threads>>>(d_vec_a, d_vec_b, d_result);

//Copy result back to host
    cudaMemcpy(result, d_result, SIZE * sizeof(int), cudaMemcpyDeviceToHost);

//Print result
```

```
49      for (int i = 0; i <3; i++)
50      {
51          printf("%d␣+␣%d␣=␣%d\n", vec_a[i], vec_b[i], result[i]);
52      }
53
54  //Free device memory
55      cudaFree(d_vec_a);
56      cudaFree(d_vec_b);
57      cudaFree(d_result);
58
59      return 0;
60  }
```

A.2 Tabellen

Folgende Tabelle enthält einen Ausschnitt der Spezifikationen, die aus der Compute Capability ausgelesen werden können[1]

Feature Support	Compute Capability						
Technical Specifications & Features	**1.0**	**1.1**	**1.2**	**1.3**	**2.X**	**3.0**	**3.5**
Maximum dimensionality of grid of thread blocks	2				3		
Maximum x-dimension of a grid of thread blocks	65535					$2^{31}-1$	
Maximum y- or z-dimension of a grid of thread blocks	65535						
Maximum x- or y-dimension of a block	512				1024		
Maximum z-dimension of a block	64						
Maximum number of threads per block	512				1024		
Warp size	32						
Maximum number of resident blocks per multiprocessor	8					16	
Maximum number of resident warps per multiprocessor	24		32		48	64	
Maximum number of resident threads per multiprocessor	768		1024		1536	2048	
Number of 32-bit registers per multiprocessor	8K		16K		32K	64K	
Maximum number of 32-bit registers per thread	128				63	255	
Maximum amount of shared memory per multiprocessor	16KB				48KB		
Number of shared memory banks	16				48		
Double-precision floating-point numbers	No			Yes			

Tabelle A.1: Technical Specifications per Compute Capability

1 Quelle: [3]

Literatur

[1] *Basic Linear Algebra Subprograms (BLAS)*. Feb. 2014. URL: http://www.netlib.org/blas/index.html.

[2] *CUBLAS LIBRARY*. v5.5. User Guide. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/cublas/index.html.

[3] *CUDA C PROGRAMMING GUIDE*. v5.5. Design Guide. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] *CUDA COMPILER DRIVER NVCC*. v5.5. Reference Guide. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[5] *CUDA RUNTIME API*. v5.5. API Reference Manual. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[6] *CUFFT LIBRARY USER'S GUIDE*. 5.5. NVIDIA Corporation. Aug. 2013. URL: http://docs.nvidia.com/cuda/cufft/index.html.

[7] *CURAND LIBRARY*. v5.5. Programming Guide. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/curand/index.html.

[8] *CUSPARSE LIBRARY*. v5.5. NVIDIA Corporation. Juli 2013. URL: http://docs.nvidia.com/cuda/cusparse/index.html.

[9] Christophe Dubach u. a. „Compiling a high-level language for GPUs:(via language support for architectures and compilers)“. In: *ACM SIGPLAN Notices*. Bd. 47. 6. ACM. 2012, S. 1–12.

[10] Rob Farber. *CUDA application design and development*. Elsevier, 2011.

[11] *GNU Compiler Collection*. URL: http://gcc.gnu.org/.

[12] M. Holschneider. *Wavelets: an analysis tool*. Oxford mathematical monographs. Clarendon Press, 1995. ISBN: 9780198534815.

[13] David B Kirk und W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. 2. Aufl. Newnes, 2012.

[14] Kazuhiko Komatsu u. a. „Evaluating performance and portability of OpenCL programs“. In: *The fifth international workshop on automatic performance tuning*. 2010.

[15] Erik Lindholm u. a. „NVIDIA Tesla: A unified graphics and computing architecture“. In: *IEEE micro* 28.2 (2008), S. 39–55.

[16] Aaftab Munshi u. a. *OpenCL programming guide*. Pearson Education, 2011.

[17] John Nickolls und William J Dally. „The GPU computing era“. In: *Micro, IEEE* 30.2 (2010), S. 56–69.

[18] *NVCC NVIDIA CUDA COMPILER*. URL: `https://developer.nvidia.com/cuda-downloads`.

[19] *NVIDIA Performance Primitives (NPP)*. v5.5. NVIDIA Corporation. März 2013. URL: `http://docs.nvidia.com/cuda/npp/index.html`.

[20] *OpenCL*. Feb. 2014. URL: `http://www.khronos.org/opencl/`.

[21] *OpenCL 1.1 Reference Pages*. Feb. 2014. URL: `http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/`.

[22] Kamisetty Rao, Do Nyeon Kim und Jae Jeong Hwang. *Fast Fourier Transform-Algorithms and Applications: Algorithms and Applications*. Springer, 2011.

[23] Jason Sanders und Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[24] Thomas Sauer. „The Continuous Wavelet Transform: Fast Implementation and Pianos“. In: *Monografías Matemáticas García de Galdeano* VV (2013), S. 1–8.

[25] Tomas Sauer. *Einführung in die Signal- und Bildverarbeitung*. Techn. Ber. Vorlesungsskript. Universität Passau, Juli 2012.

[26] John E Stone, David Gohara und Guochun Shi. „OpenCL: A parallel programming standard for heterogeneous computing systems“. In: *Computing in science & engineering* 12.3 (2010), S. 66.

[27] *THRUST QUICK START GUIDE*. v5.5. NVIDIA Corporation. Juli 2013. URL: `http://docs.nvidia.com/cuda/thrust/index.html`.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift