Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Studienarbeit – Hardware-Systeme

Studienrichtung
Informatik (Master)

Bernd Fecht (954020)
Julian Hillesheimer (954018)

Heterogenes Rechnen mit OpenCL – Praktische Anwendung

Prüfer: Prof. Dr. Gundolf Kiefer
Abgabe der Arbeit am: 30.06.2017

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Studienarbeit:
Bernd Fecht
Julian Hillesheimer

Inhaltsverzeichnis

1 Einleitung

Diese Studienarbeit beschäftigt sich mit dem Thema OpenCL. Diese Arbeit ist Teil einer Zusammenarbeit zwischen Julian Hillesheimer und Bernd Fecht und gliedert sich in zwei Dokumente auf. Im ersten Dokument wurden bereits die theoretischen Grundlagen und Aspekte von OpenCL erläutert. Dieses Dokument fokussiert sich auf eine Wärmeausbreitungssimulation mit OpenCL und bildet den Abschluss dieser Studienarbeit.

1.1 Motivation

OpenCL bietet eine vereinheitliche Schnittstelle, um auf verschiedenen uneinheitlichen Parallelrechnern parallelisierten Programmcode ausführen zu können. Die Motivation sich mit OpenCL zu beschäftigen, besteht deshalb vor allem darin, einen Programmcode zu schreiben, der bestimmte Algorithmen enthält, die parallel und somit beschleunigt auf einer Vielzahl von unterschiedlichen Parallelrechnern ausgeführt werden können.

1.2 Ziel der Arbeit

Ziel dieser Studienarbeit ist es, einen Einblick auf die Verwendung von OpenCL zu verschaffen. Dies wird anhand der praktischen Umsetzung der Wärmeausbreitung in OpenCL verdeutlicht.

1.3 Aufbau der Arbeit

Die Studienarbeit gliedert sich in fünf Kapitel auf. In dem ersten Kapitel wird das Ziel der Arbeit und die Motivation für das Thema dieser Arbeit erläutert.

Anschließend werden mathematische Grundlagen für die Wärmeausbreitungssimulation vermittelt. Diese bilden das Fundament zum Verständnis der praktischen Umsetzung.

Im nächsten Kapitel wird auf die Implementierung der Wärmeausbreitungssimulation eingegangen.

Danach werden verschiedene Zeitmessungen veranschaulicht und interpretiert.

Ein Fazit schließt die Arbeit ab.

2 Theoretische Grundlagen Wärmeausbreitung

In den nachfolgenden Abschnitten wird erläutert, was der Begriff Wärmeleitung bedeutet, was die Wärmeleitungsgleichung ist und wie die Wärmeleitung in einem zweidimensionalen Medium mit konstanter Randbedingung simuliert werden kann.

2.1 Was ist Wärmeleitung?

Unter dem Begriff Wärmeleitung versteht man einen Energietransport infolge atomarer und molekularer Wechselwirkung unter dem Einfluss ungleichförmiger Temperaturverteilungen. Dabei kann an jeder Stelle eines Körpers, der für diese Studienarbeit als homogen und isotrop angenommen wird, zu jedem Zeitpunkt ein Wärmestrom festgestellt werden. [1]

2.2 Wärmeleitungsgleichung

Die Verteilung der Wärme kann mittels der Wärmeleitungsgleichung, siehe Gleichung 1.1, berechnet werden. Die Wärmeleitungsgleichung ist eine parabolische Differentialgleichung, welche den Zusammenhang zwischen der räumlichen und zeitlichen Ausbreitung der Wärme in einem Körper beschreibt. [2]

$$\frac{\partial}{\partial t} u\left(\vec{x}, t\right) - \alpha \Delta\left(\vec{x}, t\right) = 0 \tag{1.1}$$

Die Funktion $u\left(\vec{x}, t\right)$ liefert die Temperatur an der Stelle $\vec{x}$ zum Zeitpunkt t, Δ ist der Laplace-Operator bezüglich $\vec{x}$ und α bildet die Konstante für Temperaturleitfähigkeit des Mediums. [2]

2.3 Wärmeleitung simulieren

Die Wärmeleitung bzw. die dafür verwendete Wärmeleitungsgleichung muss nun so transformiert werden, damit sie an einem Computer numerisch gelöst werden kann. Dies erfolgt durch Diskretisierung im Raum. Dazu wird das zweidimensionale Medium mit einem Gitter überzogen, welches durch gleichgroße Raster mit der Länge h zusammengesetzt ist. Somit kann eine Näherungslösung erzielt werden, da durch die Diskretisierung nun ein Gitter anstatt einem Intervall verwendet werden kann. [3]

Um dies zu erreichen, muss die Gleichung 1.1 nun schrittweise umgeformt werden. Da für die Simulation ein kartesisches Koordinatensystem verwendet wird, ergibt sich durch Umformen die Gleichung 1.2.

$$\frac{\partial}{\partial t}u\left(\vec{x},t\right)=\alpha\Delta\left(\vec{x},t\right)=\alpha\left(\frac{\partial^2}{\partial x^2}u\left(t,x,y\right)+\frac{\partial^2}{\partial y^2}u\left(t,x,y\right)\right) \tag{1.2}$$

Die Ableitungen lassen sich nun mittels der Taylorreihe, wie in Gleichung 1.3 und Gleichung 1.4 dargestellt, umformen.

$$\frac{\partial}{\partial t}u\left(t,x,y\right)\cong\frac{u\left(t+\Delta t,x,y\right)-u\left(t,x,y\right)}{\Delta t} \tag{1.3}$$

$$\begin{aligned}\frac{\partial^2}{\partial x^2}u\left(t,x,y\right)&\cong\frac{u\left(t,x+\Delta x,y\right)-2u\left(t,x,y\right)+u\left(t,x-\Delta x,y\right)}{\Delta x^2}\\ \frac{\partial^2}{\partial y^2}u\left(t,x,y\right)&\cong\frac{u\left(t,x,y+\Delta y\right)-2u\left(t,x,y\right)+u\left(t,x,y-\Delta y\right)}{\Delta x^2}\end{aligned} \tag{1.4}$$

Die vorherigen Umformungen können nun in der Gleichung 1.5 zusammengefasst werden und bilden die diskretisierte Wärmeleitungsgleichung, welche nun implementiert werden kann, um die Wärmeleitung in einem zweidimensionalen Medium zu simulieren. [4]

$$u\left(t+\Delta t,x,y\right)=u\left(t,x,y\right)+\Delta t\alpha\left(\frac{\partial^2}{\partial x^2}u\left(t,x,y\right)+\frac{\partial^2}{\partial y^2}u\left(t,x,y\right)\right) \tag{1.5}$$

Bei der Implementierung muss darauf geachtet werden, dass geeignete Werte für Δt und $h$also Δx und Δy gewählt werden, da ansonsten der Diskretisierungsfehler steigt und die Lösung des numerischen Verfahrens nicht zur kontinuierlichen Lösung konvergiert.

3 Durchführung und Implementierung

3.1 Einrichtung einer OpenCL Umgebung

Für das Ausführen von OpenCL Code wird zunächst eine OpenCL Runtime benötigt. Diese kann von dem jeweiligen Hardware-Hersteller des Devices bezogen werden, auf dem das OpenCL Programm letztendlich ausgeführt wird.

Des Weiteren wird das OpenCL SDK für das jeweilige Device benötigt. Auch das SDK ist über den Hardware-Hersteller beziehbar. In diesem Fall soll das Programm auf einer NVIDIA GPU und als Referenz auf einer Intel CPU ausgeführt werden. NVIDIA liefert das SDK innerhalb ihres CUDA Toolkits. Als Betriebssystem wird in dieser Arbeit Microsoft Windows 10 genutzt. Die Installation der Treiber und der SDKs gestalten sich dabei sehr einfach.

Bei der Verwendung von Visual Studio muss daraufhin lediglich das Include Verzeichnis und die OpenCL.lib und OpenCL.dll im Linker angegeben werden.

3.2 Implementierung Wärmeausbreitung

3.2.1 Definieren der Kernel-Parameter und der Work-Item Größe

Zunächst ist es erforderlich, die Daten zu definieren, die an den Kernel übergeben werden. Neben den Parametern für die Berechnung, die näher in Abschnitt 2 erläutert sind, sind dies drei 2D-Matrizen. Die erste 2D-Matrix enthält dabei die Wärmewerte für die vorhergehende Iteration, also die Ausgangsmatrix. In der zweiten 2D-Matrix werden die berechneten Werte zurückgeschrieben. Diese Matrix dient dann in der nächsten Iteration als Ausgangsmatrix. Die dritte 2D-Matrixist ist die Ergebnismatrix, in der das Ergebnis zurückgeschrieben wird. Jede Matrix hat dabei die gleiche Höhe und Breite.

Wir definieren die Ausgangsmatrix in dem Programm so, dass in den Rändern der 2D-Matrix konstant die Wärme anliegt. Deshalb wird die Ausgangsmatrix in der ersten Spalte mit dem Wert 255 initialisiert. Alle anderen Randwerte erhalten den konstanten Faktor 0.

Ausgangsmatrix

255	0	0	0
255	x1y1	x2y1	0
255	x1y2	x2y2	0
255	x1y3	x2y3	0
255	x1y4	x2y4	0
255	x1y5	x2y5	0
255	x1y6	x2y6	0
255	x1y7	x2y7	0
255	0	0	0

Legende: grün: anliegende Wärme
orange: zu berechnende Wärme

Tabelle 1: Ausgangsmatrix

Im C Code werden diese Matrizen als einfache Arrays dargestellt (mit *Länge = Höhe x Breite* der 2D-Matrix) und daraufhin als Buffer auf das Device geschrieben. Schließlich wird der Buffer dem Kernel noch als Parameter übergeben, sodass der Kernel auf die Daten zugreifen kann. Die nötigen Funktionen, um die Buffer auf dem Device zu initialisieren und als Parameter an den Kernel zu übergeben, sind im Abschnitt 3.2.4 zu finden.

Dadurch, dass in der Berechnung die Wärme immer über die benachbarten Werte berechnet wird, dürfen die Randwerte nicht mitberechnet werden, da es hier keine unmittelbaren Nachbarn gibt. Deshalb, und weil die Werte an den Rändern sowieso konstant anliegen, wird die Skalierung der Work-Items nach der Skalierung der zu berechneten Werte definiert. Die Größe der Work-Items kann demnach folgendermaßen deklariert werden: *Work-Item-Größe = (Höhe der 2D-Matrix – 2) x (Breite der 2D-Matrix - 2)*. In Tabelle 1 entspricht dies dem orangenen Anteil.

3.2.2 Definieren der Work-Group Größe

Die Work-Items können in sogenannte Work-Groups unterteilt werden. Dabei ist es wichtig, dass die Anzahl der Work-Items durch die Datengröße ohne Rest teilbar ist, da sonst das Problem nicht vollständig aufgeteilt werden kann. Zulässig wären beispielsweise 256x256 oder 512x256 für die Datengröße bei einer Work-Group Skalierung von 64x1.

3.2.3 CL Kernel

Der OpenCL Kernel beinhaltet den Programm-Code, der tatsächlich auf dem OpenCL Device ausgeführt wird. Alle Berechnungen innerhalb des Kernels werden parallelisiert ausgeführt, da mehrere Kernel-Instanzen gleichzeitig auf dem OpenCL Device erstellt und ausgeführt werden. Die Anzahl ist dabei abhängig von dem eingesetzten Device. Die nötigen Schritte, um den Kernel auf dem Device auszuführen, werden in Kapitel 3.2.4 beschrieben. Die Berechnungen für die Wärmeausbreitung werden alle innerhalb des Kernels ausgeführt.

```
__kernel
void calcHeatConduction(__constant float* params, __constant int*
dimensions, __global float* heatPrevious, __global float* heatValues,
__global float* finalResult)
{
int x = get_global_id(0);
int y = get_global_id(1);
```

Listing 1: Definieren des zu berechnenden Wertes

Listing 1 zeigt, wie innerhalb des Kernels, über die eindeutige ID des Work-Items die Position des zu berechnenden Werts herausgefunden wird.

```
float cOrigVal = heatPrevious[x + 1 + ind + dimensions[0]];
float cVal = -2.0f * cOrigVal;
float uxx = cVal + heatPrevious[x + ind + dimensions[0]] + heatPrevious[x + 2
+ ind + dimensions[0]];
float uyy = cVal + heatPrevious[x + 1 + ind] + heatPrevious[x + 1 + ind +
dimensions[0] + dimensions[0]];

float temp = 1.0f / params[0];
temp = temp * temp;

float val = cOrigVal + params[1]*params[2]*(uxx + uyy)*temp;
heatValues[x + 1 + ind + dimensions[0]] = val;

finalResult[x + 1 + ind + dimensions[0]] = val;
}
```

Listing 2: Berechnung und Zurückschreiben der Wärmewerte

Anschließend wird die eigentliche Berechnung, wie in Abschnitt 2.3 definiert, durchgeführt und die Ergebnisse in die Buffer (heatValues und finalResult) zurückgeschrieben.

Der Kernel und somit die Berechnung der Wärmeausbreitung werden dabei iterativ ausgeführt. Dabei wird der Buffer mit dem Ergebnis (heatValues) mit dem Buffer Ausgangsbuffer (heatPrevious) nach jeder Iteration getauscht, sodass die berechneten Werte als neue Ausgangswerte für die nächste Iteration dienen.

3.2.4 OpenCL C API

Um den Kernel letztendlich auf dem Gerät zu erstellen und auszuführen werden zusätzlich noch einige Schritte benötigt, die über die OpenCL C API bereitgestellt werden. Dabei orientiert sich das Programm an den Schritten der typischen OpenCL Runtime. [5]

```
// 1. Discovering platform and devices

cl_uint numPlatforms = 0;
status = clGetPlatformIDs(0, NULL, &numPlatforms);

cl_platform_id *platforms = NULL;
platforms = (cl_platform_id*)malloc(numPlatforms * sizeof(cl_platform_id));

status = clGetPlatformIDs(numPlatforms, platforms, NULL);

cl_uint numDevices = 0;
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL,
&numDevices);

cl_device_id *devices;
devices = (cl_device_id*)malloc(numDevices * sizeof(cl_device_id));

status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices,
devices, NULL);
```

Listing 3: Herausfinden der Plattformen und Devices

Zunächst müssen wie in Listing 3 über Anfragen an die OpenCL API die Plattform und das Device herausgefunden und ausgewählt werden. Eine Plattform wäre beispielsweise NVIDIA und das zugehörige Device für diese Plattform eine NVIDIA Grafikkarte.

```
// 2. Creating a context
cl_context context = clCreateContext(NULL, numDevices, devices, NULL, NULL,
&status);

// 3. Creating a command-queue for the first device
cl_command_queue cmdQueue = clCreateCommandQueue(context, devices[0],
CL_QUEUE_PROFILING_ENABLE, &status);
```

Listing 4: Erstellen des Kontext und der Command-Queue

Als Nächstes muss der Kernel erzeugt werden, der die Host zu Device Koordination übernimmt und die Command-Queue erzeugt werden, die für das Senden der Befehle an das Device zuständig ist.

```
// 4. Creating buffers to hold data
cl_mem bufPrevious = clCreateBuffer(context, CL_MEM_READ_WRITE, datasize,
NULL, &status);

[...]

// 5. Copying the input data onto the device

status = clEnqueueWriteBuffer(cmdQueue, bufPrevious, CL_TRUE, 0, datasize,
heatPrevious, 0, NULL, NULL);

[...]
```

Listing 5: Buffer erstellen und initialisieren

In den nächsten Schritten werden für die Parameter, die an den Kernel übergeben werden sollen, Buffer angelegt und die Daten in den entsprechenden Buffer kopiert.

```
// 6. Creating and building the kernel

cl_program program = clCreateProgramWithSource(context, 1, (const
char**)&programSource, NULL, &status);

status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
```

Listing 6: Erstellen des Kernels und Builden des Kernels

Danach wird der eigentliche Kernel erstellt. Der Programmcode für den Kernel liegt dabei als ein char Array vor. Danach wird der Kernel gebaut und ist somit fast für die Ausführung bereit. Zunächst müssen jedoch noch die Parameter an den Kernel übergeben werden.

Nach der Ausführung der Kernel müssen noch die Ergebnisse aus dem Buffer des Device an den Host zurück geschrieben werden. Anschließend müssen alle allokierten Ressourcen wieder freigegeben werden und die Ausführung ist somit abgeschlossen.

4 Messung und Ergebnisse

In diesem Kapitel wird auf die, im Rahmen dieser Arbeit, durchgeführten Messungen und Ergebnisse eingegangen.

4.1 Testsysteme, Messmethode und Messparameter

Die Messungen wurden auf verschiedenen Geräten ausgeführt mit verschiedenen Messparametern. Deshalb werden die verwendeten Testsysteme vorab vorgestellt und erklärt, wie die Messwerte entnommen wurden und welche Parameter verwendet wurden.

4.1.1 Testsysteme

Als Testsysteme wurden insgesamt zwei Computer benutzt, deren Spezifikationen hier näher erläutert werden.

1. System: Desktop PC
   - CPU: Intel Xeon E3-1231
     - 4 Kerne a 3.40 GHz, Hyperthreading, 8MB Cache, Turbo-Clock bis 3.80 GHz
   - RAM: 8GB RAM

- GPU: NVIDIA GeForce GTX 750 Ti
  - Basis Taktung 1020MHz, Boost-Taktung 1085 MHz
  - GPU RAM: 2048 MB
  - 640 Processing-Elements

2. System: Acer Aspire M5-581TG Notebook
   - CPU: Intel Core i5-3317U
     - 2 Kerne a 1,70 GHz, Hyperthreading, 3MB Cache, Turbo-Clock bis 2,60 GHz
   - RAM: 4GB RAM
   - Prozessor-GPU: Intel HD Graphics 4000
     - 350MHz Taktung
     - GPU RAM: Shared Memory
     - 16 Processing-Elements
   - GPU: NVIDIA GeForce GT 640M
     - 625MHz Taktung
     - GPU RAM: 2048MB
     - 364 Processing-Elements

4.1.2 Messmethode

Gemessen wird die reine Ausführungszeit auf dem Device. Dazu werden die Events von OpenCL über die Funktion clGetEventProfilingInfo() nach ihrem Status abgefragt (siehe Listing 7).

```
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(startTime), &startTime, NULL);
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(endTime),
&endTime, NULL);

time = (endTime - startTime) * 10e-6f;
totalTime += time;
}

printf("Total time: %0.3f ms\n", totalTime);
printf("Average time per step: %0.3f ms\n", totalTime / steps)
```

Listing 7: Zeitmessung mit OpenCL

Gemessen wird die Zeit in Millisekunden.

4.1.3 Messparameter

In dieser Arbeit wurde die Laufzeit für folgende Parameter gemessen:

- Verschiedene Devices
- Verschiedene Work-Group Größen

- Unterschiedliche Datengrößen

4.2 Durchführung und Interpretation der Zeitmessung

Zunächst wurde die Wärmeausbreitung auf der GPU des 1. Systems, also mit der NVIDIA GeForce GTX 750 Ti berechnet. Dabei wurde eine Datengröße von 640x320 gewählt, bei einer Anzahl von 10.000 Iterationen.

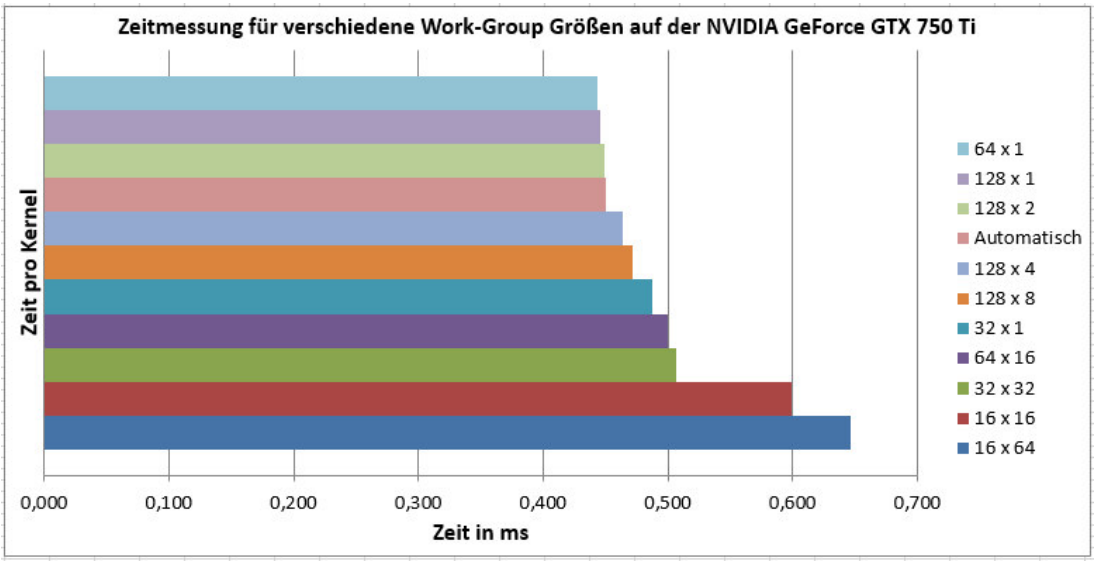

Abbildung 1: Zeitmessung für verschiedene Work-Group Skalierungen

Für die Zeitmessung wurden, wie in Abbildung 1 zu sehen, die Laufzeit für verschiedene Work-Group Skalierungen gemessen, um die optimale Work-Group Size für das Programm herauszufinden. Die Zeit wurde dabei im Durchschnitt für 10.000 Iterationen in Millisekunden angegeben. Wie in dem Graph zu sehen läuft das Programm am schnellsten für eine Work-Group Skalierung von 64x1, dicht gefolgt von 128x1. Je weiter die Work-Groups auf zwei Dimensionen aufgeteilt wurde, wurde das Programm immer unperformanter. Mit einer Größe von 16x64 läuft das Programm am langsamstem. Vermutlich liegt das daran, wie die Daten im Speicher liegen. Bei einer eindimensionalen Verteilung werden die Daten einfach als Array, hintereinander in den Speicher geschrieben. Somit können die Speicherbereiche nacheinander abgegriffen werden, ohne im Speicher hin- und herspringen zu müssen.

Bei der Berechnung der optimalen Work-Group-Size spielt die Größe der Daten keine Rolle. Getestet wurde dabei mit unterschiedlichen Größen, immer mit dem selbem Ergebnis. Auch wie die Work-Items für die Größen skaliert wurden, spielt keine Rolle.

Interessant war jedoch das Verhalten für die automatische Berechnung der Work-Group Size, denn mit steigender Datengröße näherte sich die Laufzeit immer mehr an den Referenzwert für die optimale Work-Group-Size von 64x1 an. Daher ist zu vermuten, dass OpenCL ab einer bestimmten Größe von ca. 1280x1280 ebenfalls die 64x1 als optimale Größe für die Work-Group Skalierung errechnet hat.

Nachdem die optimale Work-Group Skalierung herausgefunden wurde, wurde gemessen, wie sich unterschiedliche Datengrößen auf die Laufzeit auswirken. Dabei wurden wieder 10.000 Iterationen verwendet und die durchschnittliche Laufzeit eines Kernels in Millisekunden angegeben.

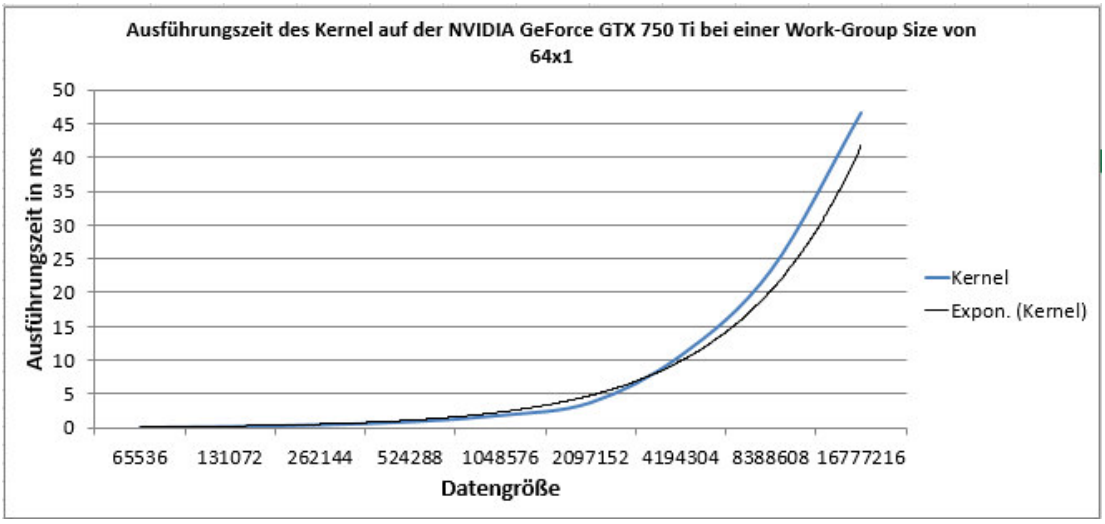

Abbildung 2: Ausführungszeit für verschiedene Datengrößen

Wie in Abbildung 2 zu sehen, steigt die Ausführungszeit der Kernel exponentiell zu der Datengröße an. Vergleichsweise wurde zur Veranschaulichung in der Abbildung noch eine exponentielle Kurve miteingefügt.

Zuletzt wurde die Zeit für die verschiedenen Geräte gemessen. Dabei wurde wieder die optimale Work-Group Skalierung von 64x1 verwendet. Die Datengröße wurde dabei bei jeder Messung verändert. Gemessen wurde auf allen in Abschnitt 4.1.1 erläuterten Devices.

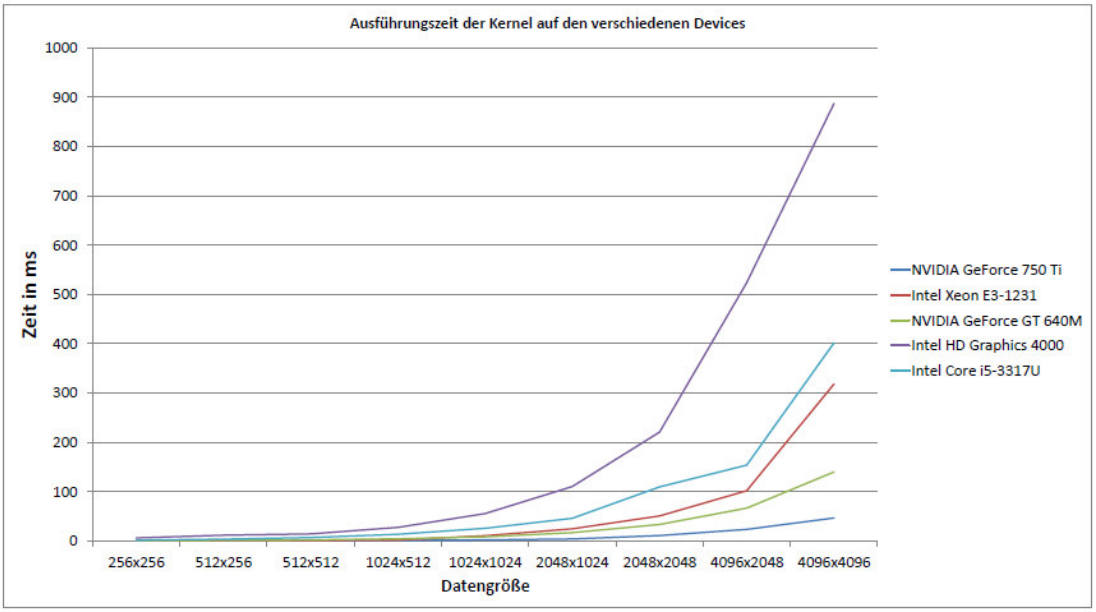

Abbildung 3: Zeitmessung auf verschiedenen Geräten

Zu sehen ist, dass die Intel HD Graphics 4000 mit seiner Ausführungszeit nicht überzeugend ist und egal bei welcher Datengröße am schlechtesten abschneidet. Das liegt vermutlich an den wenigen Processing-Elements, der niedrigen Taktrate und vor allem an dem Shared Memory der Prozessor-GPU. Dieses Device ist also für die Aufgabenstellung kein geeignetes Gerät.

Am besten schneiden die zwei NVIDIA GPUs ab. Vor allem wenn die Datengröße steigt. Hier kann die GPU durch seine Vielzahl an Recheneinheiten punkten.

Auch ist zu beobachten, dass die CPUs und die NVIDIA GPUs für eine niedrige Datengröße zunächst sehr ähnliche Laufzeiten haben. Das nächste Diagramm zeigt dies noch im Detail:

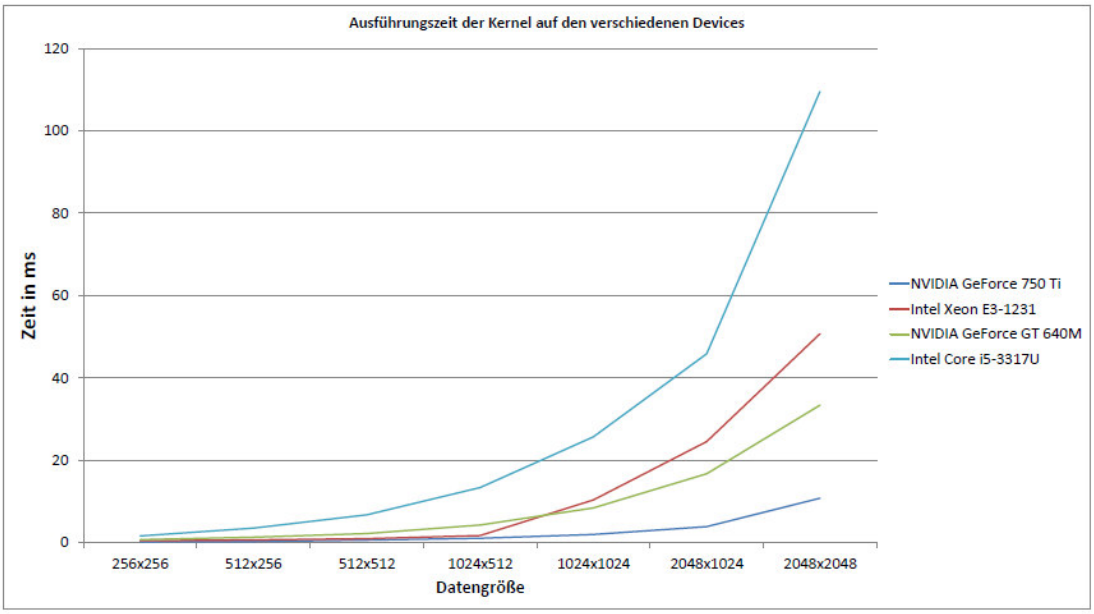

Abbildung 4: Zeitmessung für eine Auswahl der Devices

In der Abbildung 4 sieht man den Vergleich der Laufzeit der zwei CPUs zu der NVIDIA GeForce GT 640M im Detail. Auffällig ist, dass der Intel Xeon E3-1231 bis zu einer gewissen Datengröße sogar schneller ist als die NVIDIA GeForce GT 640M GPU. Woran liegt das?

Zum einen kann es durchaus sein, dass die CPU hier durch ihre hohe Taktfrequenz der GPU zunächst überlegen ist. Die Daten werden trotz der Aufteilung auf die 4 Kerne schneller berechnet. Auch liegt die Vermutung nahe, dass ab einem gewissen Zeitpunkt die Datengröße signifikant größer ist als die Größe des Caches der CPU. Da die CPU die Daten aus dem RAM in den Cache lädt, braucht die CPU mit steigender Datengröße auch mehr Laufzeit, da sie ständig den Cache wieder aktualisieren muss.

Das selbe Phänomen lässt sich auch beim Vergleich der NVIDIA GeForce 750Ti GPU zu der Intel Xeon E3-1231 CPU beobachten. Ab einer Datengröße von 1024x512 steigt die Laufzeit der CPU im Vergleich zur GPU signifikant an.

5 Fazit

Wie in diesem praktischen Beispiel anhand der Messwerte gezeigt wird, kann es sich durchaus lohnen, OpenCL für eine Beschleunigung einer bestimmten Problemstellung zu verwenden.

Selbstverständlich richtet sich die Performance des mit OpenCL beschleunigten Programms an die richtige Implementierung und korrekte Ausführung des Codes.

Hier bietet die Implementierung von OpenCL viele Stolperfallen, vor allem wegen und durch die jeweiligen Besonderheiten der genutzten Hardware, wie zum Beispiel verwendete Caches. Das hat sich auch in den in dieser Arbeit erfassten Messwerten verdeutlicht, bei der die Intel GPU aufgrund ihres geteilten Speichers wesentlich unperformanter war als die CPU. Hier müssen vor allem die Spezifikationen und Funktionsweisen der jeweiligen Hardware im Voraus analysiert werden, um letztendlich eine performante und optimierte Lösung für die Implementierung des Programms zu finden.

Des Weiteren muss die Problemstellung eingehend analysiert werden, da sich nicht alle Probleme für eine parallele Umsetzung eignen. Zudem sollte man auch bei einem, für eine Parallelisierung geeigneten Programm, die Problemgröße ansehen. Denn ist die Laufzeit sowieso geringer als einige Millisekunden, lohnt sich eine parallele Abarbeitung des Problems eher nicht. Grundsätzlich gilt, dass vor allem bei steigender Datenmenge ist eine parallele Abarbeitung eines Problems durchaus effizienter im Vergleich zu der sequentiellen Abarbeitung.

Auch die Fehlersuche im Programmcode gestaltet sich schwierig, da man OpenCL Code nicht klassisch debuggen kann. Dies führt zu einer Steigerung der Entwicklungszeit. Anfänglich sollte also auch abgeschätzt werden, ob sich der Aufwand bei der Implementierung im Vergleich zu der gewonnenen Laufzeit rentiert.

6 Literaturverzeichnis

[1] Grigull U and Sandner H 1990 *Waermeleitung (Waerme- und Stoffuebertragung)* 2nd edn (Berlin: Springer)

[2] Wikipedia 2017 *Wärmeleitungsgleichung* https://de.wikipedia.org/w/index.php?oldid=155535722 (accessed 29 Jun 2017)

[3] *TUM INFO V - Vorlesungen* https://www5.in.tum.de/lehre/vorlesungen/ (accessed 29 Jun 2017)

[4] Douglas Andrade *Case study: heat transfer simulation using CLGL interop.* http://www.cmsoft.com.br/opencl-tutorial/case-study-heat-transfer-simulation-using-clgl-interop/ (accessed 29 Jun 2017)

[5] Kaeli D R, Mistry P, Schaa D and Zhang D P 2015 *Heterogeneous Computing with OpenCL 2.0* (Elsevier Science)

7 Anhang

7.1 Hauptprogramm

```
#include <stdio.h>
#include <stdlib.h>
#include <CL\cl.h>

char* readFile(const char* filename) {
	FILE *fp;
	char *fileData;
	long fileSize;

	// open
	fp = fopen(filename, "rb");
	if (!fp) {
		printf("Could not open file \n");
		//exit(-1);
	}

	// file size

	if (fseek(fp, 0, SEEK_END)) {
		printf("Error reading file 1 \n");
		//exit(-1);
	}

	fileSize = ftell(fp);

	if (fileSize < 0) {
		printf("Error reading file 2 \n");
		//exit(-1);
	}

	if (fseek(fp, 0, SEEK_SET)) {
		printf("Error reading file 3 \n");
		//exit(-1);
	}

	// read

	fileData = (char*)malloc(fileSize + 1);
	if (!fileData) {
		//exit(-1);
	}

	fread(fileData, fileSize, 1, fp);

	// terminate string

	fileData[fileSize] = '\0';

	// close file
	if (fclose(fp)) {
		printf("Error closing file \n");
		exit(-1);
	}
```

```
	//for (int i = 0; i < fileSize; i++) {
	//	printf("%c", fileData[i]);
	//}

	return fileData;
}

void printOpenCLInfos() {
	int i, j;
	char* value;
	size_t valueSize;
	cl_uint platformCount;
	cl_platform_id* platforms;
	cl_uint deviceCount;
	cl_device_id* devices;
	cl_uint maxComputeUnits;

	// get all platforms
	clGetPlatformIDs(0, NULL, &platformCount);
	platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) *
platformCount);
	clGetPlatformIDs(platformCount, platforms, NULL);

	for (i = 0; i < platformCount; i++) {

		// get all devices
		clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0,
NULL, &deviceCount);
		devices = (cl_device_id*)malloc(sizeof(cl_device_id) *
deviceCount);
		clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL,
deviceCount, devices, NULL);

		// for each device print critical attributes
		for (j = 0; j < deviceCount; j++) {

			// print device name
			clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0,
NULL, &valueSize);
			value = (char*)malloc(valueSize);
			clGetDeviceInfo(devices[j], CL_DEVICE_NAME,
valueSize, value, NULL);
			printf("%d. Device: %s\n", j + 1, value);
			free(value);

			// print hardware device version
			clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, 0,
NULL, &valueSize);
			value = (char*)malloc(valueSize);
			clGetDeviceInfo(devices[j], CL_DEVICE_VERSION,
valueSize, value, NULL);
			printf(" %d.%d Hardware version: %s\n", j + 1, 1,
value);
			free(value);

			// print software driver version
			clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, 0,
NULL, &valueSize);
			value = (char*)malloc(valueSize);
```

```
                clGetDeviceInfo(devices[j], CL_DRIVER_VERSION,
valueSize, value, NULL);
                printf(" %d.%d Software version: %s\n", j + 1, 2,
value);
                free(value);

                // print c version supported by compiler for
device
                clGetDeviceInfo(devices[j],
CL_DEVICE_OPENCL_C_VERSION, 0, NULL, &valueSize);
                value = (char*)malloc(valueSize);
                clGetDeviceInfo(devices[j],
CL_DEVICE_OPENCL_C_VERSION, valueSize, value, NULL);
                printf(" %d.%d OpenCL C version: %s\n", j + 1, 3,
value);
                free(value);

                // print parallel compute units
                clGetDeviceInfo(devices[j],
CL_DEVICE_MAX_COMPUTE_UNITS,
                        sizeof(maxComputeUnits), &maxComputeUnits,
NULL);
                printf(" %d.%d Parallel compute units: %d\n", j +
1, 4, maxComputeUnits);

            }

            free(devices);

        }

        free(platforms);
}

void printOpenCLInfos2() {
        int i, j;
        char* info;
        size_t infoSize;
        cl_uint platformCount;
        cl_platform_id *platforms;
        const char* attributeNames[5] = { "Name", "Vendor",
            "Version", "Profile", "Extensions" };
        const cl_platform_info attributeTypes[5] = { CL_PLATFORM_NAME,
CL_PLATFORM_VENDOR,
            CL_PLATFORM_VERSION, CL_PLATFORM_PROFILE,
CL_PLATFORM_EXTENSIONS };
        const int attributeCount = sizeof(attributeNames) /
sizeof(char*);

        // get platform count
        clGetPlatformIDs(5, NULL, &platformCount);

        // get all platforms
        platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id) *
platformCount);
        clGetPlatformIDs(platformCount, platforms, NULL);

        // for each platform print all attributes
        for (i = 0; i < platformCount; i++) {
```

```
            printf("\n %d. Platform \n", i + 1);

            for (j = 0; j < attributeCount; j++) {

                  // get platform attribute value size
                  clGetPlatformInfo(platforms[i], attributeTypes[j],
0, NULL, &infoSize);
                  info = (char*)malloc(infoSize);

                  // get platform attribute value
                  clGetPlatformInfo(platforms[i], attributeTypes[j],
infoSize, info, NULL);

                  printf("  %d.%d %-11s: %s\n", i + 1, j + 1,
attributeNames[j], info);
                  free(info);

            }

            printf("\n");

      }

      free(platforms);
}

int main() {
      const bool debug = false;
      const int width = 4098;
      const int height = 2050;
      const float deltaX = 0.5;
      const float deltaT = 0.01f;
      const float alpha = 1.0f;
      const int steps = 10000;
      size_t datasize = sizeof(float) * width * height;

      const float heatParams[3] = { deltaX, deltaT, alpha };
      const int heatDimensions[2] = { width, height };
      float* heatPrevious = (float*)malloc(datasize);
      float* heatValues = (float*)malloc(datasize);
      float* finalResult = (float*)malloc(datasize);
      if (debug) {
            printOpenCLInfos();
            printOpenCLInfos2();
      }

      for (int i = 0; i < width * height; i++) {
            if (i % width == 0)
                  heatPrevious[i] = heatValues[i] = finalResult[i] =
250;
            else
                  heatPrevious[i] = heatValues[i] = finalResult[i] =
0;
      }

      // 1. Discovering platform and devices

      cl_int status;
      cl_uint numPlatforms = 0;
      status = clGetPlatformIDs(0, NULL, &numPlatforms);
```

```
	if (debug)
		printf("%zd: ", status);

	cl_platform_id *platforms = NULL;
	platforms = (cl_platform_id*)malloc(numPlatforms *
sizeof(cl_platform_id));

	status = clGetPlatformIDs(numPlatforms, platforms, NULL);
	if (debug)
		printf("%zd: ", status);

	cl_uint numDevices = 0;
	status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
NULL, &numDevices);
	if (debug)
		printf("%zd: ", status);

	cl_device_id *devices;
	devices = (cl_device_id*)malloc(numDevices *
sizeof(cl_device_id));

	status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL,
numDevices, devices, NULL);
	if (debug)
		printf("%zd: ", status);

	// 2. Creating a context

	cl_context context = clCreateContext(NULL, numDevices,
devices, NULL, NULL, &status);
	if (debug)
		printf("%zd: ", status);

	// 3. Creating a command-queue for the first device

	cl_command_queue cmdQueue = clCreateCommandQueue(context,
devices[0], CL_QUEUE_PROFILING_ENABLE, &status);
	if (debug)
		printf("%zd: ", status);

	// 4. Creating buffers to hold data

	cl_mem bufPrevious = clCreateBuffer(context,
CL_MEM_READ_WRITE, datasize, NULL, &status);
	if (debug)
		printf("%zd: ", status);
	cl_mem bufValues = clCreateBuffer(context, CL_MEM_READ_WRITE,
datasize, NULL, &status);
	if (debug)
		printf("%zd: ", status);
	cl_mem bufResult = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
datasize, NULL, &status);
	if (debug)
		printf("%zd: ", status);
	cl_mem bufParams = clCreateBuffer(context, CL_MEM_READ_ONLY,
sizeof(heatParams), NULL, &status);
	if (debug)
		printf("%zd: ", status);
	cl_mem bufDimensions = clCreateBuffer(context,
CL_MEM_READ_ONLY, sizeof(heatDimensions), NULL, &status);
```

```
	if (debug)
		printf("%zd: ", status);

	// 5. Copying the input data onto the device

	status = clEnqueueWriteBuffer(cmdQueue, bufPrevious, CL_TRUE,
0, datasize, heatPrevious, 0, NULL, NULL);
	if (debug)
		printf("%zd: ", status);
	status = clEnqueueWriteBuffer(cmdQueue, bufValues, CL_TRUE, 0,
datasize, heatValues, 0, NULL, NULL);
	if (debug)
		printf("%zd: ", status);
	status = clEnqueueWriteBuffer(cmdQueue, bufParams, CL_TRUE, 0,
sizeof(heatParams), heatParams, 0, NULL, NULL);
	if (debug)
		printf("%zd: ", status);
	status = clEnqueueWriteBuffer(cmdQueue, bufDimensions,
CL_TRUE, 0, sizeof(heatDimensions), heatDimensions, 0, NULL, NULL);
	if (debug)
		printf("%zd: ", status);

	// 6. Creating the kernel and creating + building the prorgam

	char* programSource = readFile("HeatConductionKernel2.txt");

	/*const char* programSource =
	"__kernel \n"
	"void calcHeatConduction(__global float* A, __global float* B,
__global float* C) \n"
	"{ \n"
	"\n"
	"int idx = get_global_id(0); \n"
	"C[idx] = A[idx] + B[idx]; \n"
	"} \n";*/

	cl_program program = clCreateProgramWithSource(context, 1,
(const char**)&programSource, NULL, &status);
	if (debug)
		printf("%zd: ", status);
	status = clBuildProgram(program, numDevices, devices, NULL,
NULL, NULL);
	if (debug)
		printf("%zd: ", status);

	// 7. Extracting the kernel from the program

	cl_kernel kernel = clCreateKernel(program,
"calcHeatConduction", &status);
	if (debug)
		printf("%zd: ", status);

	// 8. Executing the kernel

	//status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
&bufPrevious);
	//printf("%zd: ", status);
	//status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
&bufValues);
	//printf("%zd: ", status);
```

```
	//status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
&bufResult);
	//printf("%zd: ", status);

	//size_t indexSpaceSize[2], workGroupSize[2];
	//indexSpaceSize[0] = width - 2;
	//indexSpaceSize[1] = height - 2;
	//workGroupSize[0] = width - 2;
	//workGroupSize[1] = height - 2;

	//status = clEnqueueNDRangeKernel(cmdQueue, kernel, 2, NULL,
indexSpaceSize, workGroupSize, 0, NULL, NULL);
	//printf("%zd: ", status);

	size_t indexSpaceSize[2], workGroupSize[2];
	indexSpaceSize[0] = width - 2;
	indexSpaceSize[1] = height - 2;
	workGroupSize[0] = 64;
	workGroupSize[1] = 1;

	cl_event event;
	cl_ulong startTime, endTime;
	double totalTime = 0.0;
	double time;

	for (int step = 1; step <= steps; step++) {
		if (step % 1000 == 0)
			printf("Calculated steps %d ...\n", step);
		if (step % 2 == 0) {
			status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
&bufPrevious);
			if (debug)
				printf("%zd: ", status);
			status = clSetKernelArg(kernel, 3, sizeof(cl_mem),
&bufValues);
			if (debug)
				printf("%zd: ", status);
		}
		else {
			status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
&bufValues);
			if (debug)
				printf("%zd: ", status);
			status = clSetKernelArg(kernel, 3, sizeof(cl_mem),
&bufPrevious);
			if (debug)
				printf("%zd: ", status);
		}
		status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
&bufParams);
		if (debug)
			printf("%zd: ", status);
		status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
&bufDimensions);
		if (debug)
			printf("%zd: ", status);
		status = clSetKernelArg(kernel, 4, sizeof(cl_mem),
&bufResult);
		if (debug)
			printf("%zd: ", status);
```

```
            status = clEnqueueNDRangeKernel(cmdQueue, kernel, 2,
NULL, indexSpaceSize, workGroupSize, 0, NULL, &event);
            if (debug)
                  printf("%zd: ", status);

            clFinish(cmdQueue);

            clGetEventProfilingInfo(event,
CL_PROFILING_COMMAND_START, sizeof(startTime), &startTime, NULL);
            clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
sizeof(endTime), &endTime, NULL);
            time = (endTime - startTime) * 10e-6f;
            totalTime += time;
      }

      printf("Total time: %0.3f ms\n", totalTime);
      printf("Average time per step: %0.3f ms\n", totalTime /
steps);

      // 9. Copying output data back to the host

      status = clEnqueueReadBuffer(cmdQueue, bufResult, CL_TRUE, 0,
datasize, finalResult, 0, NULL, NULL);
      if (debug)
            printf("%zd: ", status);

      // 10. Releasing resources

      clReleaseKernel(kernel);
      clReleaseProgram(program);
      clReleaseCommandQueue(cmdQueue);
      clReleaseMemObject(bufPrevious);
      clReleaseMemObject(bufValues);
      clReleaseMemObject(bufResult);
      clReleaseContext(context);
/*
      for (int i = 0; i < width * height; i++) {
            if (i % width == 0)
                  printf("\n");
            printf("%f ", finalResult[i]);
      }
*/
      if (debug)
            printf("%f", finalResult[width * 150 + 1]);

      free(heatPrevious);
      free(heatValues);
      free(finalResult);
      free(platforms);
      free(devices);
      free(programSource);

      system("pause");

      return 0;
}
```

7.2 OpenCL Kernel

```
__kernel
void calcHeatConduction(__constant float* params, __constant int*
dimensions, __global float* heatPrevious, __global float*
heatValues, __global float* finalResult)
{
int x = get_global_id(0);
int y = get_global_id(1);

int ind = dimensions[0] * y;

float cOrigVal = heatPrevious[x + 1 + ind + dimensions[0]];
float cVal = -2.0f * cOrigVal;
float uxx = cVal + heatPrevious[x + ind + dimensions[0]] +
heatPrevious[x + 2 + ind + dimensions[0]];
float uyy = cVal + heatPrevious[x + 1 + ind] + heatPrevious[x + 1 +
ind + dimensions[0] + dimensions[0]];

float temp = 1.0f / params[0];
temp = temp * temp;

float val = cOrigVal + params[1]*params[2]*(uxx + uyy)*temp;
heatValues[x + 1 + ind + dimensions[0]] = val;

finalResult[x + 1 + ind + dimensions[0]] = val;
}
```