technische universität
dortmund

GPGPU-Programmierung mit OpenACC

Andreas Blume

2. Oktober 2018

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für praktische Informatik (LS4)
http://ls4-www.cs.tu-dortmund.de

fakultät für
informatik
Lehrstuhl für praktische Informatik

Inhaltsverzeichnis

Kapitel 1

Einleitung

Da die physikalischen Grenzen von sequentiellen Architekturen mehr oder weniger erreicht sind, stellt die parallele Programmierung eine Schlüsseltechnologie für die Performancesteigerung von Software dar. Denn nur so können Ergebnisse schneller berechnet und (noch) komplexere Probleme gelöst werden. [Hag16, Folie 6] Dies spiegelt sich besonders bei Anwendungen aus dem Bereich der Simulation naturwissenschaftlicher Phänomene z.B. die Wettervorhersage, das Design von Medikamenten oder computergraphischen Anwendungen aus Film-, Spiel- und Werbeindustrie wieder, wo eine steigende Nachfrage an immer höherer Rechenleistung erkennbar ist. [RR12, Seite 1]

Um mit Hilfe der parallelen Programmierung Software zu beschleunigen, muss der verwendete, sequentielle Algorithmus für eine parallele Abarbeitung vorbereitet werden. Dazu muss die sequentielle Anwendung mit einer parallelen Programmiersprache formuliert werden oder durch den Einsatz von Programmierumgebungen mit zusätzlichen Direktiven oder Anweisungen versehen werden. Je nach Algorithmus kann so ein recht großer Aufwand für die Erstellung eines effizienten, parallelen Programms entstehen, der im Erfolgsfall mit einer schnelleren Ausführung auf einer geeigneten Plattform belohnt wird. [RR12, Seite 2]

Der große Vorteil von portablen Programmierumgebungen ist die Möglichkeit ein paralleles Programm auf einer Vielzahl unterschiedlicher Plattformen ausführen zu können. Diese Eigenschaft ist die Grundlage für den breiten Einsatz in unterschiedlichen Bereichen. [RR12, Seite 2] Ein Beispiel für eine portable Programmierumgebung ist OpenACC, dass in dieser Dokumentation vorgestellt wird.

Das nächste Kapitel 2 gibt zunächst einen kurzen Überblick über die Aufgabe einer Grafikkarte (GPU) im Computersystem und die heutige GPU-Architektur. Anschließend wird die GPGPU-Programmierung, wobei die Abkürzung GPGPU für *General Purpose Computation on Graphics Processing Unit* steht, allgemein vorgestellt und ein erster Blick auf OpenACC geworfen. Abschließend wird OpenACC mit OpenMP, dass einen ähnlichen Ansatz für die parallele CPU-Programmierung verfolgt, verglichen.

Im dritten Kapitel der Dokumentation werden zwei einfache OpenACC-Beispiele vorgestellt. Dabei wird zunächst anhand einer Approximation der Kreiszahl Pi gezeigt, wie ein sequentielles Programm mit Hilfe von OpenACC auf einer GPU parallel ausgeführt werden kann. Außerdem wird mit einem zweiten Beispiel gezeigt, welche Möglichkeiten es in OpenACC gibt, um einen Speichertransfer zwischen Host und Beschleuniger durchzuführen. Beide Beispiele werden mit aussagekräftigen Messungen abgerundet.

Das vierte Kapitel stellt OpenACC im Detail vor. Die Konzentration liegt zunächst auf den Umgebungsvariablen und Makros von OpenACC. Danach werden die wichtigsten OpenACC-Funktionen beschrieben und jeweils anhand eines kurzen Beispiels die Verwendung gezeigt. Am Ende des Kapitels rücken die OpenACC-Pragmas in den Vordergrund, die beispielsweise zur Parallelisierung einer for-Schleife genutzt werden. Auch hier werden die wichtigsten Pragmas beschrieben und jeweils ein Beispiel angeben.

In Kapitel 5 wird anhand von drei Varianten gezeigt wie die Approximation von Pi aus Kapitel 2 unter paralleler Nutzung aller verfügbaren GPUs eines Rechners berechnet werden kann. Die erste Variante, eine reine OpenACC-Implementierung, basiert auf der asynchronen Kommunikation zwischen Host und den GPUs des Rechners. Bei den anderen beiden Implementierungen handelt es sich um eine Kombination aus pthreads und OpenACC sowie OpenMP und OpenACC. Dabei werden mit Hilfe von pthreads und OpenMP der parallele Aufruf der verfügbaren GPUs implementiert und OpenACC zur GPU-Programmierung genutzt. Den Abschluss des Kapitels bildet ein Vergleich der Ausführungszeiten aller vorgestellten Varianten.

Das letzte Kapitel 6 zeigt wie OpenACC-Programme mit Hilfe von passenden Profilingtools (nvprof und NVIDIA® Visual Profiler) untersucht werden können und so die Ausführungszeit weiter optimiert werden kann. Außerdem werden zwei Möglichkeiten zum Remote Profiling vorgestellt.

Kapitel 2

Die Grafikkarte und ihre Programmierung

In diesem Kapitel wird zunächst der Aufbau und die Funktionsweise einer Grafikkarte beschrieben. Danach wird ein kurzer Einblick in die GPGPU-Programmierung gegeben und OpenACC allgemein vorgestellt. Abschließend wird OpenACC mit OpenMP, dass einen ähnlichen Ansatz für die parallele CPU-Programmierung verfolgt, verglichen.

2.1 Die Grafikkarte

Grafikprozessoren (GPU = *Graphical Processing Unit*) waren ursprünglich Prozessoren, die speziell an die Bedürfnisse der Computergrafik - der Generierung eines Bildes aus einer abstrakten Objektbeschreibung [NFH07, Seite 6] - angepasst waren. Demzufolge konnten sie den Hauptprozessor (CPU = *Central Processing Unit*) entlasten und eine bessere graphische Ausgabe ermöglichen. Heute werden GPUs, aufgrund ihrer parallelen Many-Core-Architektur, immer häufiger für Berechnungen im wissenschaftlich-technischen Bereich (GPGPU = *General Purpose Computation on Graphics Processing Unit*) eingesetzt. Natürlich unterstützen auch die einfache Zugänglichkeit und die kostengünstige Anschaffung gegenüber anderen parallelen Systemen den Einsatz. Ein weiterer Vorteil ist das spezielle Design von GPUs, dass einen hohen Datendurchsatz erlaubt. Dadurch kann ein hoher Effizienzgewinn bei vielen parallelen Programmen erreicht werden.

2.1.1 Die Grafikkarte im Computersystem

Die Hauptaufgabe der ersten Grafikkarten war die Ausgabe von Daten auf dem Monitor. Demzufolge besaßen die GPUs sehr unterschiedliche Kommunikationskanäle für den Datentransfer vom Hauptspeicher zum GPU-Speicher bzw. zurück. Heutzutage ist eine GPU über PCI Express mit dem Motherboard, speziell der Northbridge, eines Computers verbunden. PCI Express (*Peripheral Component Interconnect Express*, Kurz: PCIe)

ist der Nachfolger von AGP (Accelerated Graphics Port) und bietet eine deutlich höhere Datenübertragerate pro Pin auf Hin- und Rückweg. Das 2012 eingeführte PCIe 3.0 erreicht je Lane und Richtung eine Datenübtragungsrate von ∼1GB/s (8GT/s, Gigatransfer pro Sekunde) [Bor17]. Der Nachfolger PCIe 4.0 (erschienen 2017) schafft sogar ∼2GB/s (16GT/s) [Bor17]. Diese Kommunikationsbandbreite ist häufig der größte Engpass bei der GPU-Programmierung. [KVBC12, Seite 3] Demzufolge sollte eine Berechnung nur dann auf die Grafikkarte ausgelagert werden (engl. *offload*), wenn die Vorteile eines GPU-Einsatzes die Kommunikationskosten übertreffen.

Heutige Computersysteme bestehen aus CPU, GPU und Arbeitsspeicher (RAM), sowie der North- und Soutbridge. Zu den Aufgaben der Northbridge gehöhrt die Steuerung des Datenflusses zwischen CPU, GPU und RAM, während sich die Southbridge auf die Verwaltung von Peripherie-Schnittstellen für unterschiedlichste Geräte (z.B. Maus, Tastatur und Monitor) konzentriert. Der vereinfachte Aufbau eines heutigen Computersystems ist in Abbildung 2.1 zu sehen.

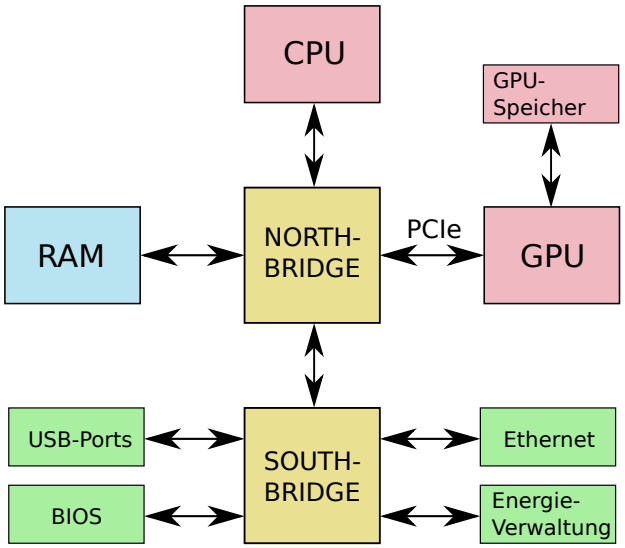

Abbildung 2.1: Vereinfachter Aufbau eines heutigen Computersystems.

2.1.2 Die Architektur einer Grafikkarte

Da CPUs nicht besonders gut für komplexe Graphikanwendungen ausgelegt waren (auch heute noch) und deshalb eine unzureichende Performance lieferten, wurde sehr früh der Entschluss gefasst einen dedizierten Graphikbeschleuniger zu entwickeln. Infolgedessen konnte die Architektur der CPU und der GPU unabhängig voneinander an die jeweiligen Anforderungen angepasst und optimiert werden. Da Grafikkarten eine große Menge an Gleitkommazahlen parallel verarbeiten müssen, besteht eine GPU aus vielen unabhängigen SIMD-Prozessoren. [RR12, Seite 387 bis 395] SIMD steht für **S**ingle **I**nstruction **M**ultiple

Data und bedeutet, dass eine Aktion zeitgleich auf mehrere Daten angewendet wird (siehe [Fly72]). Am Besten verdeutlicht dies eine Matrix- oder Vektorberechnung, wo dieselbe Operation auf unterschiedliche Elemente der Matrix oder des Vektors angewendet wird. Jeder SIMD-Prozessor besteht aus mehreren SIMD-Funktionseinheiten (FE). Jede SIMD-FE besitzt separate, lokale Register und es ist somit ein Datentransfer zwischen dem Speicher und den Registern notwendig. Die genaue Anzahl an SIMD-Prozessoren und Funktionseinheiten ist von der Architektur der jeweiligen GPU abhängig. Die NVIDIA® Quadro® P6000 der Workstation ls4gpu2 (weitere Details siehe Anhang A) besteht beispielsweise aus 30 Multiprozessoren, wobei jeder Prozessor über 65536 Register verfügt und ein Maximum von 2048 Threads besitzt. Die Größe des globalen Speichers beträgt 23,871 GiB.

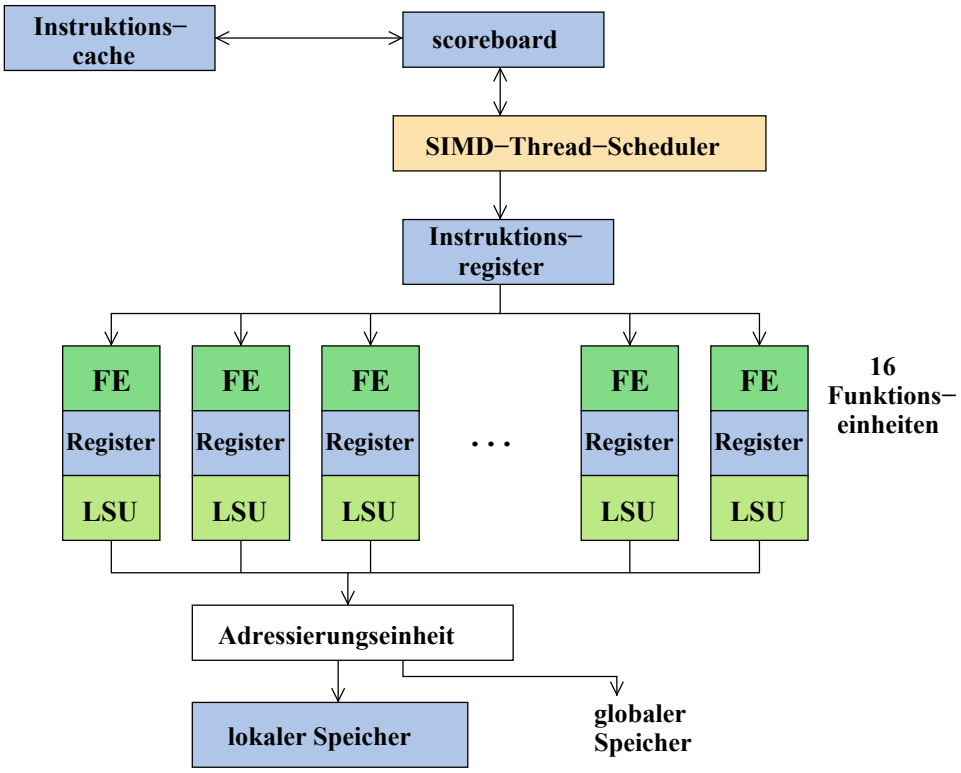

Abbildung 2.2: Blockdiagramm eines SIMD-Prozessors mit 16 Funktionseinheiten (FE), von denen jede eine separate Menge von Registern und eine eigene Transfereinheit (engl. *Load-Store-Unit*, LSU) hat. [RR12, Seite 390]

Die heutigen SIMD-Prozessoren unterstützen mehrere, unabhängige SIMD-Threads. Die einzelnen Threads werden von einem SIMD-Threads-Scheduler ausgewählt, dessen Basis eine Tabelle (engl. *Scoreboard*) mit aktuellen Informationen zu den einzelnen Threads ist, und zur Ausführung gebracht.

Das Blockdiagramm in Abbildung 2.2 zeigt den Aufbau eines SIMD-Prozessors mit 16 SIMD-Funktionseinheiten. Die LSUs (*Load-Store-Units*) stellen die Transfereineinheiten zwischen dem Speicher und den Registern der SIMD-Funktionseinheiten dar.

2.2 GPGPU-Programmierung

Seit vielen Jahren werden GPUs nicht nur zur Lösung von graphischen Problemen eingesetzt, sondern auch bei allgemeinen, rechenintensiven Berechnungen (GPGPU = *General Purpose Computation on Graphics Processing Unit*). Auf diese Weise kann die CPU entlastet werden und die Berechnungszeit bei großen homogenen Daten, aufgrund der parallelen Arbeitsweise einer GPU, deutlich verkürzt werden. Deshalb wird in diesem Abschnitt zunächst ein Blick auf die Entstehung der GPGPU-Programmierung geworfen und danach mit OpenACC die neuste Möglichkeit der GPU-Programmierung vorgestellt. Anschließend wird OpenACC mit OpenMP, das einen ähnlichen Ansatz für die parallele CPU-Programmierung verfolgt, verglichen.

2.2.1 Geschichte

Die ursprüngliche Aufgabe einer Grafikkarte war die Beschleunigung der Grafikwiedergabe und -anzeige. Demzufolge basierte die Programmschnittstelle auf Shader-Sprachen wie DirectX, OpenGL (*Open Graphics Library*) und Cg (*C for Graphics*). [KVBC12, Seite 1] Programme konnten als nur in Form von Operationen der Grafikpipeline ausgedrückt und anschließend auf der GPU ausgeführt werden. Später erfolgte die Einführung von Sprachen zum Stream Processing (ab 2004), zum Beispiel BrookGPU und Sh, die die GPU-Programmierung etwas komfortabler machten. BrookGPU [Bro18] wurde an der Stanford Universität entwickelt und erlaubt die Nutzung von GPUs als Coprozessor und Beschleuniger. Außerdem ist es plattformunabhängig und besitzt einen komplexen Kompiliervorgang. Da auch andere große Firmen die Entwickelung der GPU-Programmierung verfolgten, entwickelten sie eigene Frameworks für Stream Processing. Darunter das auf NVIDIA-Grafikkarten beschränkte Framework CUDA (*Compute Unified Device Architecture*, seit 2007, aktuelle Version: 9.2.88 [CUD18a]) sowie das ursprünglich von Apple entwickelte OpenCL (*Open Computing Language*, seit 2008, aktuelle Version: 2.2-7 [Ope18b]), dass heutzutage von der Khronos Group [Kro18] , einem im Jahre 2000 gegründetem Industriekonsortium mit über 100 Mitgliedern (u.a. AMD, Intel, NVIDIA, SGI, Apple, Microsoft, Google sowie Oracle), standardisiert und weiterentwickelt wird.

2.2.2 OpenACC

Um die GPU-Nutzung noch weiter zu verbessern und zu vereinfachen, steht seit 2011 mit OpenACC eine weitere Möglichkeit zur GPU-Programmierung zur Verfügung. Die OpenACC-Spezifikation [Ope18a] wurde ursprünglich von CAPS Enterprise, Cray Inc., The Portland Group (PGI) und NVIDIA sowie der Unterstützung von mehreren Universitäten und Forschungseinrichtungen entwickelt. [KH16, Seite 414]. Seitdem sind weitere Hersteller, Universitäten und Firmen hinzugekommen, so dass sich, wie der Name schon vermuten

lässt, OpenACC zu einem offenen Standard entwickelt hat. Der zweite Teil von OpenACC "ACC" steht für das englische Wort *accelerators*, zu deutsch "Beschleuniger".

OpenACC wurde für moderne Hochleistungsrechner (engl. *high-performance computing (HPC) systems*) entwickelt, die aus Multicore-Prozessoren und verschiedenen Parallelbeschleunigern (z.B. GPUs) bestehen. Dabei kann ähnlich wie bei OpenMP (vgl. Abschnitt 2.2.3.1) mit Hilfe von minimalen Anpassungen auf Basis von Compileranweisungen, Bibliotheksfunktionen und Umgebungsvariablen ein sequentielles C, C++ oder Fortran-Programm parallelisiert werden und beispielsweise auf einem Multicore-Prozessor und/oder GPU ausgeführt werden. Das heißt der Programmierer muss nicht explizit den sequentiellen Programmcode in Kernel umformen und Register anlegen, wie es in CUDA [CUD18b] der Fall ist. Dies übernimmt bei OpenACC komplett der Compiler, der die notwendigen Kernel generiert, Register erstellt und Shared-Memory-Variablen anlegt. [KH16, Seite 414-415]

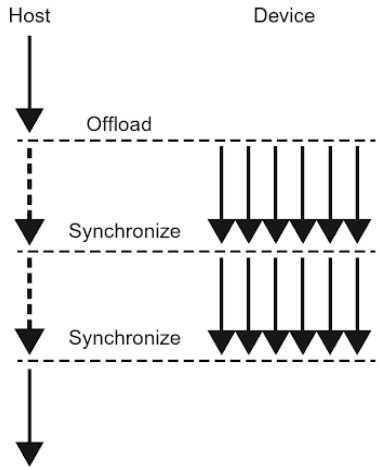

Abbildung 2.3: Das Ausführungsmodell von OpenACC auf Basis von Auslagern (engl. *offload*). [KH16, Seite 415]

Die Basis von OpenACC ist ein einfaches Programmiermodell, dass eine hohe Performance auf unterschiedlichen parallelen Plattformen ermöglicht. Die Ausführung eines OpenACC-Programms startet immer auf einer sogenannten Host-CPU, die anschließend die Daten und die Ausführung an einen *Beschleuniger* auslagern (engl. *offload*) kann (siehe Abbildung 2.3). Der Beschleuniger kann das gleiche physische Gerät wie der Host sein, im Falle eines Multicore-Prozessors, oder ein daran angeschlossenes Gerät wie z.B. eine GPU, die über den PCI Express (PCIe) Bus mit der Host-CPU verbunden ist. Dabei können Host und Beschleuniger über einen separaten oder gemeinsamen Speicher verfügen. Um jedoch die Portabilität eines OpenACC-Programms zu erhöhen, sollte angenommen werden, dass ein physikalisch getrennter Beschleuniger nur einen eigenen, separaten Speicher besitzt. Demzufolge müssen benötigte Daten erst zum Beschleuniger übertragen werden, bevor mit der Berechnung begonnen werden kann (vgl. Abbildung 2.4). [KH16, Seite 414-415] OpenACC bietet hierzu unterschiedliche Möglichkeiten an. Dabei sollte der Programmie-

rer immer berücksichtigen, dass der Datenaustausch zwischen Host und Beschleuniger Zeit in Anspruch nimmt und somit ein optimales Resultat nur bei einem entsprechend hohen Rechenaufwand mit gut überlegter Kommunikation erzielt werden kann.

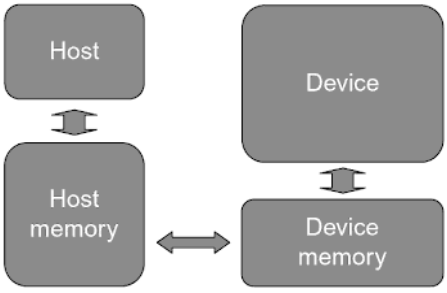

Abbildung 2.4: Das abstrakte Maschinenmodell von OpenACC. [KH16, Seite 415]

Um einen ersten Eindruck von OpenACC zu bekommen, wird im folgenden eine Art „Hallo Welt!“-Beispiel für OpenACC (siehe Code 2.1) vorgestellt. Der C-Code kopiert zunächst

```
1  #include <stdio.h>
2
3  #ifdef _OPENACC
4      #include <openacc.h>
5  #endif
6
7  int main(int argc, char* argv[]) {
8      #ifdef _OPENACC
9          // NVIDIA-GPU Nummer 1 auswaehlen
10         acc_set_device_num(0, acc_device_nvidia);
11
12         int a[10];
13         #pragma acc parallel loop copy(a[0:10])
14         for(int i=0; i<10; i++)
15             a[i] = i;
16
17         for(int i=0; i<10; i++)
18             printf("%d, ", a[i]);
19     #else
20         printf("OpenACC wird nicht unterstuetzt.\n");
21     #endif
22
23     printf("Fertig.\n");
24     return 0;
25 }
```

Code 2.1: „Hallo Welt!“-Beispiel in OpenACC.

ein Array a zum Beschleuniger. Anschließend wird a mit Hilfe des Beschleunigers initialisiert und am Ende auf dem Host-System ausgeben. Im Code sind alle Bestandteile von OpenACC enthalten: Compileranweisungen (`#pragma acc parallel for copy(a[0:10])`), Bibliotheksfunktionen (`acc_set_device_num()`) und Umgebungsvariablen (`_OPENACC`).

Der Code 2.1 kann mit dem gcc-Compiler (ab Version 7) wie folgt kompiliert werden:

```
gcc-7 <PROGRAMM-NAME>.c -fopenacc -foffload=nvptx-none -foffload="-O3" -O3
                                -o <PROGRAMM-NAME>
```

Die Ausgabe von Code 2.1 sieht wie folgt aus:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, Fertig.
```

2.2.3 OpenACC und OpenMP - ein Vergleich

In diesem Abschnitt wird zunächst OpenMP, dass einen ähnlichen Ansatz wie OpenACC verfolgt, kurz vorgestellt. Anschließend werden OpenACC und OpenMP verglichen und dabei die wichtigsten Gemeinsamkeiten und Unterschiede präsentiert.

2.2.3.1 OpenMP

Der Name OpenMP setzt sich aus zwei Teilen zusammen. Während das "Open" deutlich macht, dass es sich um einen offenen Standard handelt, steht das "MP" für *multi processing*. [HL08, Seite 2] Die offizielle Internetseite von OpenMP ist `http://www.openmp.org`. Von dort kann die komplette Spezifikation, das Syntax-Referenzhandbuch sowie ein Dokument mit ausführlich beschriebenen Beispielen im PDF-Format heruntergeladen werden. [Ope15e]

OpenMP ist eine offene Programmierschnittstelle, um in C, C++ und Fortran-Programmen Parallelität spezifizieren zu können. Dabei wählt OpenMP ein anderen Weg als viele konkurrierende Ansätze, da lediglich minimale Änderungen (in Form von Compileranweisungen, Bibliotheksfunktionen und Umgebungsvariablen) am ursprünglich sequentiellen Quellcode notwendig sind. Demzufolge können sequentielle C/C++ und Fortran-Programme deutlich schneller parallelisiert werden und die Lesbarkeit bleibt erhalten. [HL08, Seite 1]

Die Grundlage von OpenMP ist ein portables paralleles Programmiermodell für Shared-Memory-Architekturen, dass aufgrund seines offenen Standards verschiedenen Herstellern zur Verfügung steht. Dadurch wird OpenMP von zahlreichen Compilerhersteller unterstützt. Die entsprechenden Compiler verfügen oft über eine Kommandozeilenoption mit der die Interpretation von OpenMP-spezifischen Compileranweisungen an- und ausgeschaltet

werden kann. [HL08, Seite 1-2] Beim gcc-Compiler beispielsweise mit der Option `-fopenmp`.

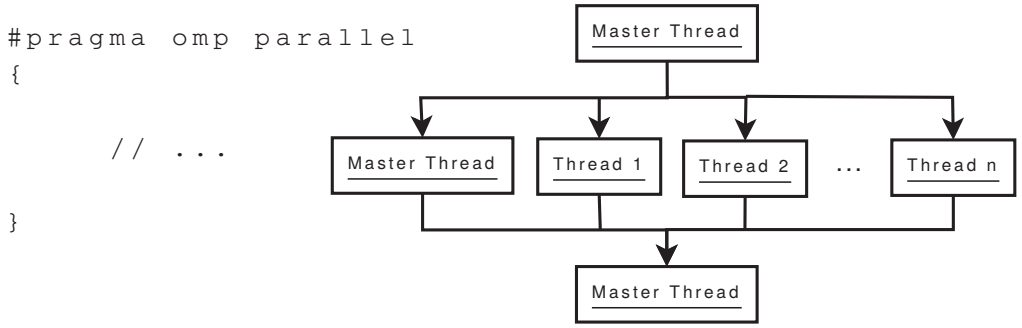

Abbildung 2.5: Verwendung des Fork/Join-Ausführungsmodell in OpenMP. [HL08, Seite 24]

Da OpenMP auf dem *Fork/Join*-Ausführungsmodell aufbaut (siehe Abbildung 2.5), ist zu Beginn eines jeden OpenMP-Programms nur ein sogenannter `Master-Thread` aktiv. Trifft dieser im weiteren Programmverlauf auf eine `#pragma omp parallel`-Anweisung, so wird ein paralleler Abschnitt betreten. Zu Beginn eines solchen Abschnitts werden weitere Threads erzeugt (engl. *fork*), die anschließend parallel Berechnungen durchführen. Am Ende des parallelen Abschnitts wird die Ausführung wieder zusammengeführt (engl. *join*). [HL08, Seite 23]

Um das ganze etwas anschaulicher zu machen folgt das „Hallo Welt!“-Beispiel für OpenMP (siehe Code 2.2). Es ist ein C-Programm, in dem zunächst mit OpenMP die Anzahl an Prozessoren im System bestimmt wird und anschließend jeder Thread seine Thread-Nummer ausgibt. Das Programm enthält alle Bestandteile von OpenMP: Compileranweisungen (`#pragma omp parallel`), Bibliotheksfunktionen (z.B. `omp_get_num_procs()`) und Umgebungsvariablen (`_OPENMP`). [HL08, Seite 25]

Der Code 2.2 kann mit dem gcc-Compiler wie folgt kompiliert werden:

```
gcc <PROGRAMM-NAME>.c -o <PROGRAMM-NAME> -fopenmp
```

Die Ausgabe von Code 2.2 könnte beispielsweise wie folgt aussehen:

```
Anzahl Prozessoren: 4
Thread 0 von 4 sagt "Hallo Welt!"
Thread 2 von 4 sagt "Hallo Welt!"
Thread 3 von 4 sagt "Hallo Welt!"
Thread 1 von 4 sagt "Hallo Welt!"
Fertig.
```

```
1   #include <stdio.h>
2
3   #ifdef _OPENMP
4       #include <omp.h>
5   #endif
6
7   int main(int argc, char* argv[]) {
8       #ifdef _OPENMP
9           printf("Anzahl Prozessoren: %d\n",
                omp_get_num_procs());
10
11          #pragma omp parallel
12          {
13              printf("Thread %d von %d sagt \"Hallo Welt!\"\n",
                    omp_get_thread_num(), omp_get_num_threads());
14          }
15      #else
16          printf("OpenMP wird nicht unterstuetzt.\n");
17      #endif
18
19      printf("Fertig.\n");
20
21      return 0;
22  }
```

Code 2.2: „Hallo Welt!“-Beispiel in OpenMP. [HL08, Seite 25]

2.2.3.2 OpenACC vs. OpenMP

Vorab sollte erwähnt werden, dass die erste OpenMP-Spezifikation (OpenMP 1.0) für C und C++ aus dem Jahre 1998 [Ope98] stammt, während die OpenACC-Spezifikation 1.0 erst im November 2011 [Ope11] präsentiert wurde. Demzufolge ist auch der Funktionsumfang von OpenMP etwas größer. Im Folgenden werden OpenACC 2.5 und OpenMP 4.1 verglichen. [Bey15, LJ15, Ope15b]

Der erste große Unterschied zwischen OpenACC und OpenMP ist die Art und Weise wie Direktive dem Compiler helfen oder Vorgaben an den Übersetzungsprozess machen. OpenACC-Direktive können sowohl implizit als auch explizit sein. Dabei geben diese dem Compiler beispielsweise einen Hinweis, dass es sich um eine parallele Schleife handelt und der Compiler Code generieren soll, dass diese so schnell wie möglich auf dem zuvor ausgewählten Beschleuniger ausgeführt werden kann. In der OpenACC Spezifikation 2.5 [Ope15d] wird dies in der Introduction (Abschnitt 1.0) wie folgt formuliert: “Das Pro-

grammiermodell [von OpenACC] erlaubt dem Programmierer, die den Compilern zur Verfügung stehenden Informationen zu erweitern, einschleißlich einer Anleitung zur Abbildung von Schleifen oder anderen leistungsbezogenen Daten auf einem Beschleuniger“. Weiterhin steht im Scope (Abschnitt 1.1) der OpenACC-Spezifikation 2.5: “Das Dokument zur OpenACC-API deckt nur die benutzergesteuerte Beschleunigerprogrammierung ab, wobei der Benutzer die Regionen eines Hostprogrammes angibt, die auf einen Beschleuniger ausgelagert werden sollen.“ Das Ganze kann wie folgt zusammengefasst werden. OpenACC-Direktive bilden die Anleitung zur Abbildung von Schleifen und der Programmierer gibt die Regionen eines Hostprogramms an, dass durch Offloading ausgelagert werden soll. Bei OpenMP hingegen sind Direktive direkte Befehle an den Compiler. Sie müssen also bei der späteren Kompilierung des Programmcodes unbedingt berücksichtigt werden. Im Comment Draft von OpenMP 4.1 [Ope15a] wird dies wie folgt beschrieben: “Der Programmierer spezifiziert ausdrücklich die zu ergreifenden Maßnahmen durch den Compiler und das Laufzeitsystem, um das Programm parallel auszuführen.“ Zusätzlich wird erwähnt, dass die OpenMP-API nicht die automatische Parallelisierung und Anweisungen an den Compiler abdeckt, um die Parallelisierung zu unterstützen. OpenMP ist demzufolge ausschließlich explizit. Der Programmierer gibt direkt die vom Compiler zu ergreifenden Aktionen an. Daher kann OpenMP auch als mechanischer Source-to-Source Präprozessor geschrieben werden, was sowohl eine Stärke als auch eine Schwäche sein kann. Insgesamt funktioniert OpenMP also wie folgt: es werden einige parallele Threads erstellt, auf die nun die anschließende Schleife verteilt werden soll. Dabei wird beispielsweise SIMD-Parallelität nur dann berücksichtigt, wenn der Programmierer dies explizit angibt (siehe Code 2.3).

```
1 // OpenACC
2 #pragma acc parallel loop
3 for-Schleife
4
5 // OpenMP
6 #pragma omp parallel for [simd]
7 for-Schleife
```

Code 2.3: Parallelisierung einer for-Schleife mit OpenACC und OpenMP. Bei der OpenMP-Variante wird die SIMD-Parallelität nur dann genutzt, wenn die Klausel `simd` angegeben wird.

Der zuvor angesprochene Unterschied spiegelt sich auch bei der Erstellung von Code für Host und Device/Beschleuniger wieder. Soll mit OpenMP 4.1 eine Schleife parallel auf dem Host ausgeführt werden, so genügt das folgende Statement:

```
#pragma omp parallel for/do
for-Schleife
```

Während bei der Variante für Beschleuniger viele verschiedene Möglichkeiten existieren, um beispielsweise die Art der Parallelität exakt festlegen zu können. Die folgenden Liste zeigt einige Beispiele, wobei nach jedem Statement die eigentliche parallele for-Schleife folgen würde:

- `#pragma omp target parallel do`
- `#pragma omp target teams distribute parallel for`
- `#pragma omp target teams distribute parallel for simd`
- `#pragma omp target teams distribute simd`
- `#pragma omp target parallel for simd`
- `#pragma omp target simd`

In OpenACC genügt lediglich eine Direktive, um eine parallele Schleife auf dem Host oder Beschleuniger auszuführen, da OpenACC-Direktiven nur einen Hinweis an den Compiler darstellen und keine direkten Vorgaben zur Parallelisierung machen:

```
#pragma acc parallel loop
for-Schleife
```

Weitere Details zur Parallelisierung von Schleifen in OpenMP und OpenACC können in [Bey15, ab Folie 19] nachgelesen werden. Zusätzlich werden in [LJ15, Folie 28 bis 37] einige Übersetzungen von OpenACC-Direktiven in OpenMP-Direktive vorgestellt.

Ein weiterer wichtiger Vorteil von OpenACC kann aus dem Parallelisierungsansatz von OpenACC abgeleitet werden. Da OpenACC keine direkten Vorgaben bei Parallelisierung an den Compiler macht, kann ein und der selbe Code auf unterschiedlicher Hardware (z.B. Multicore-x86, Multicore-ARM, GPUs, usw.) sehr gut parallel ausgeführt werden. Dies erfordert jedoch eine intelligente Codegenerierung durch den Compiler, da natürlich jede Hardware ihre Stärken und Schwächen besitzt. Daher kann der Compiler die Hinweise durch die OpenACC-Direktive nutzen, um je nach Zielhardware (z.B. gemeinsamer und geteilter Speicher) beispielsweise eine unterschiedliche Anzahl an Kerneln zu generieren. Bei OpenMP hingegen macht der Programmierer direkte Vorgaben an den Kompilierungsprozess, z.B. SIMD-Parallelität soll genutzt werden oder genau X Threads mit einem bestimmten Schedulingverfahren. Dadurch wird natürlich die Portabilität von OpenMP-Programmen (etwas) eingeschränkt.

Auch die Synchronisation zwischen einzelnen Threads spielt bei Parallelisierung von Programmen eine entscheidende Rolle. OpenMP bietet hier eine große Auswahl an unterschiedlichen Synchronisationsprimitiven an. Dazu gehören unter anderem die folgenden Klauseln:

`master`, `crititcal`, `barrier`, `taskwait`, `taskgroup`, `atomic`, `flush`, `ordered`, `depend`. Bei OpenACC gibt es aktuell nur `atomic` mit einigen Einschränkungen sowie `async` und `wait`. Der Grund ist klar. Wird eine Synchronisation bei einer skalierbaren Parallelität eingesetzt, wie sie bei OpenACC angestrebt wird, führt dies in den meisten Fällen zu einer Verlangsamung des parallelen Programms. Eine Synchronisation benötigt schließlich immer etwas Zeit.

Zusammenfassend lässt sich sagen, dass OpenMP als auch OpenACC umfangreiche Pragmas/Funktionen bereitstellen, um schnell und einfach parallele Programme zu erstellen. Dabei besitzen sie Stärken und auch Schwächen. Deshalb sollte die OpenACC- und die OpenMP-Gemeinschaft zusammenarbeiten und Ideen austauschen. Das dies bereits in der Vergangenheit funktioniert hat, zeigen einige Beispiele in [Bey15, Folie 13 bis 14]. Außerdem kann bereits heute OpenACC in OpenMP ausgeführt werden. Es können also beispielsweise mehrere OpenMP-Threads die gleichen oder unterschiedliche GPUs steuern. Eine Beispiel wird im Abschnitt 5.4 dieser Dokumentation vorgestellt. Wie Michael Wolfe, Technischer Leiter von OpenACC und Technologieführer beim Compilerhersteller The Portland Group, in [Ope15b] berichtet, war die ursprüngliche Absicht OpenMP und OpenACC irgendwann zu verbinden. Bis dahin muss jedoch das Problem zwischen den ausschließlich expliziten Direktiven in OpenMP und den impliziten und expliziten Beschreibungsmöglichkeiten von Parallelität in OpenACC gelöst werden. Bis jetzt kann trotzdem festgehalten werden, dass OpenACC das Beste ist, was OpenMP passieren konnte, da es die Innovation in der Parallelität antreibt, die natürlich am Ende auch OpenMP zugutekommen. Duncan Poole, Präsident von OpenACC und Leiter der Partner-Allianzen von NVIDIA, beschreibt es in [Ope15b] sehr treffend mit einer schönen Parallele: “Infiband war wahrscheinlich das Beste, was Ethernet passieren konnte und genauso ist es bei OpenACC und OpenMP“.

Kapitel 3

Zwei einfache OpenACC-Beispiele

In diesem Kapitel werden zwei Beispiele zur Nutzung von OpenACC in C/C++ unter Verwendung des `gcc7` -Compilers vorgestellt.

3.1 Approximation der Kreiszahl Pi

Die Kreiszahl Pi ($\pi = 3.14159\ldots$) gehört zu den berühmtesten Zahlen der Mathematik und beschreibt das Verhältnis von Umfang U zu Durchmesser d eines Kreises ($\pi = U/d$).

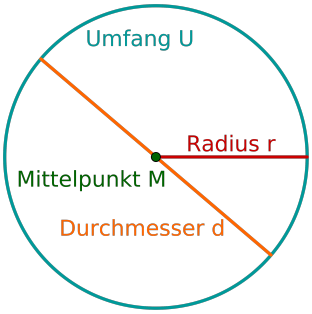

Abbildung 3.1: Ein Kreis mit Mittelpunkt M, Radius r, Durchmesser d und Umfang U.

Wir wollen im Folgenden eine Methode zur Approximation der irrationalen, transzendenten Kreiszahl Pi nutzen, um einen ersten Eindruck für die Verwendung der OpenACC-Bibliothek zu bekommen. Dabei wird zunächst ein sequentieller Algorithmus zur Approximation von Pi vorgestellt, der anschließend mit Hilfe von OpenACC parallelisiert und auf einer GPU zur Ausführung gebracht wird. Am Ende werden die Ausführungszeiten der beiden Implementierungen verglichen und ein kurzes Fazit gezogen.

3.1.1 Sequentielle Implementierung

Der Ausgangspunkt vieler paralleler Programme ist eine sequentielle Version, die anschließend parallelisiert wird. Deshalb soll zunächst ein sequentieller Algorithmus zur Appro-

ximation von Pi vorgestellt werden. Die Implementierung basiert auf der Leibnitz-Reihe und ist in Code 3.1 zu sehen. Sie besteht aus einer einfachen for-Schleife (siehe Codezeile 3), wobei die Anzahl der Schleifendurchläufe mit dem Parameter N festlegt wird und die Genauigkeit der Pi-Approximation bestimmt.

```
1 double calcPi(const long long N) {
2     double pi = 0.0f;
3     for (long long i=0; i<N; i++) {
4         double t = (double)( (i+0.5)/N );
5         pi += 4.0/( 1.0+t*t );
6     }
7     return pi/N;
8 }
```

Code 3.1: Sequentielle C-Implementierung zur Approximation von Pi. [Ope15c]

$$\begin{aligned}\pi &= 4 \arctan 1 \\ &= 4 \int_0^1 \frac{1}{1+x^2} dx \\ &\approx \frac{\sum_{i=0}^{N} \frac{4.0}{1.0 + (\frac{i+0.5}{N})^2}}{N}\end{aligned}$$

mathematischer Hintergrund ([Roy90] und [CS18, S. 590ff.])

3.1.2 Parallele Implementierung mit OpenACC

Nachdem im Abschnitt 3.1.1 eine sequentielle Variante zur Approximation von Pi vorgestellt wurde, soll diese nun mit Hilfe von OpenACC parallelisiert werden.

```
1  #include <openacc.h>      // OpenACC-Funktionen
2  #define NUM_GANGS 2048
3  #define NUM_WORKERS 32
4  #define VECTOR_LENGTH 32
5
6  double calcPi_openacc_gpu(const long long N) {
7    double pi = 0.0f;
8
9    // NVIDIA-GPU Nummer 1 auswaehlen
10   acc_set_device_num(0, acc_device_nvidia);
11
12   #pragma acc parallel num_gangs(NUM_GANGS)
        num_workers(NUM_WORKERS) vector_length(VECTOR_LENGTH)
13   #pragma acc loop gang worker vector  reduction(+:pi)
14   for (long long i=0; i<N; i++) {
15     double t = (double)( (i+0.5)/N );
16     pi += 4.0/( 1.0+t*t );
17   }
18
19   return pi/N;
20 }
```

Code 3.2: OpenACC-Implementierung zur Approximation von Pi.

Direkt beim ersten Blick auf den Code 3.2 fällt auf, dass die Änderungen im Vergleich zur sequentiellen C-Implementierung (siehe Code 3.1) nur sehr gering ausfallen. Zunächst muss mit der OpenACC-Funktion `acc_set_device_num()` (siehe Codezeile 10 im Code 3.2) das Device ausgewählt werden, dass die parallelen Berechnungen durchführen soll. In diesem Fall entscheiden wir uns für die erste NVIDIA-GPU im Rechner. Um diese Funktion nutzen zu können, muss die OpenACC-Bibliothek mit eingebunden werden (siehe Codezeile 1). Anschließend folgen zwei `Pragmas` mit denen die parallele Berechnung beschrieben wird. Das ganze hat somit einen OpenMP-ähnlichen Stil. Das erste Pragma `#pragma acc parallel` (siehe Codezeile 12) startet eine parallele Compute-Region, wobei wir mit den anschließenden Klauseln folgende Dinge festlegen:

- **num_gangs**: Anzahl der Threadblöcke/Warps (CUDA *gridDim*)
- **num_workers**: Anzahl der Arbeiter pro Warp (CUDA *blockDim*)
- **vector_length**: Vektorgröße von jedem Arbeiter (SIMD-Ausführung)

An sich ist eine parallele Compute-Region nur von begrenztem Nutzen, aber wenn sie mit der Loop-Direktive kombiniert wird, erzeugt der Compiler eine parallele Version der Schleife für das zuvor gewählte Device. Auch in unserem Fall folgt mit dem Pragma `#pragma acc loop` (siehe Codezeile 13) eine solche Loop-Direktive, die dem Compiler dazu auffordert die anschließende for-Schleife (siehe Codezeile 14 bis 17) zu parallelisieren. Die zusätzlichen Klauseln spezifizieren die Anzahl der zu verwendenden Warps (`gang`), die Anzahl Arbeiter pro Warp (`worker`) und die Vektorgröße (`vector`). Abschließend folgt mit `reduction(+:pi)` der wichtige Hinweis, dass die berechneten Teilergebnisse der einzelnen parallelen Berechnungen in der Variable `pi` summiert werden sollen.

3.1.3 Kompilierung eines OpenACC-Programms

Ein mit OpenACC parallelisiertes C-Programm kann mit dem folgenden Kommando kompiliert werden:

```
gcc-7 <PROGRAMM-NAME>.c -fopenacc -foffload=nvptx-none -foffload="-O3" -O3
                                    -o <PROGRAMM-NAME>
```

Das Compiler-Flag `-fopenacc` aktiviert die OpenACC-Unterstützung. Die Kommandos `-foffload="-O3"` und `-O3` schalten alle Compiler-Optimierungen ein. Mit `-foffload=nvptx-none` wird das Offloading aktiviert, wobei die Option nvptx-none (nvptx = *Nvidia Parallel Thread Execution*) notwendig ist, da auf der GPU kein eigenes Betriebssystem läuft (`none`).

Die Anzahl an Warps (`num_gangs`), die Warp-Größe (`num_workers`) und die Vektorgröße (`vector_length`) können auch global per Compiler-Flag gesetzt werden:

```
-fopenacc-dim=[num_gangs]:[num_workers]:[vector_length]
```

Die Nutzung von NVIDIA-GPUs kann in einem zweiten Terminal überprüft werden. Dazu muss das folgende Kommando ausgeführt werden: `nvidia-smi -l 1`

3.1.4 Messungen

Im Abschnitt 3.1.2 haben wir gesehen, dass die Parallelisierung eines sequentiellen C-Programms mit OpenACC sehr einfach ist. Es reicht das Hinzufügen von wenigen Pragmas, um ein paralleles Programm für eine GPU zu erstellen.

Da neben der einfachen Umsetzung auch die konkreten Verbesserungen der Ausführungszeit eine wichtige Rolle spielen, wurden einige Messungen durchgeführt. Dabei rücken die im Code 3.2 vorgestellten Klauselparameter: die Anzahl an Warps (`gang`), Warp-Größe (`worker`) und die Vektorgröße (`vector`) in den Mittelpunkt, da sie die einzigen konfigurierbaren Parameter der gezeigten OpenACC-Implementierung sind. Die Ergebnisse der durchgeführten Zeitmessungen sind in der Tabelle 3.1 und 3.2 zu sehen.

N	num_gangs	num_workers	vector_length	Zeit in ms
10^3	32	8	32	0.45615
10^4	32	32	32	0.38580
10^5	16	16	32	0.37055
10^6	64	8	32	0.49300
10^7	256	32	32	1.31195
10^8	1024	32	32	8.69950
10^9	2048	16	32	81.89035

Tabelle 3.1: Die besten Ausführungszeiten und ihre Konfigurationen für die OpenACC-Implementierung der Pi-Approximation (siehe Code 3.2). Die Tabelle zeigt die Ergebnisse für N zwischen 10^3 und 10^9 in Millisekunden (ms). Durchgeführt wurden die Messungen mit der Grafikkarte NVIDIA® Quadro® P6000 des Rechners ls4gpu2.

Die Tabelle 3.1 macht deutlich, dass ein unterschiedlich großer Berechnungsaufwand (N) auch verschiedene Konfigurationen der Parameter `gang`, `worker` und `vector` erfordern. Während bei einem kleinen N wie beispielsweise $N = 10^4$, 32 Warps mit einer Größe von 8 und die Verwendung der Vektorgröße 32 zur geringsten Ausführungszeit führt, werden für $N = 10^9$ 2048 Warps mit einer Größe von 16 und Vektorgröße 32 die perfekte Wahl.

Das die Ausführungszeiten bei gleich großem Berechnungsaufwand (N) sehr stark schwanken können, zeigt die Tabelle 3.2. Werden die Klauselparameter passend gesetzt, so kann die Berechnung für $N = 10^9$ in einer Zeit von $81,89$ ms (siehe grüne Markierung in Tabelle 3.2) durchgeführt werden. Aber bei einer schlechten Wahl kann die Ausführungszeit bis zu 12x langsamer sein (z.B. $1017,37$ ms bei 8 Warps der Größe 2 und Vektorgröße 32; siehe rote Markierung in Tabelle 3.2). Das richtige Setzen der einzelnen Klauselparameter ist also die große Kunst bei der Programmierung mit OpenACC und sollte gut durchdacht sein.

num_gangs	num_workers	vector_length	Zeit in ms
8	2	32	1017,37465
8	4	32	509,40725
8	8	32	330,49580
8	16	32	301,90895
8	32	32	301,88095
16	2	32	512,80225
16	4	32	256,57105
16	8	32	165,80215
16	16	32	151,13820
16	32	32	151,14220
32	2	32	256,58310
32	4	32	164,74865
32	8	32	151,16990
32	16	32	151,18375
32	32	32	151,11695
...			
256	2	32	94,75030
256	4	32	85,32350
256	8	32	109,73615
256	16	32	85,27180
256	32	32	85,26960
512	2	32	85,49830
512	4	32	111,27395
512	8	32	85,40830
512	16	32	85,24640
512	32	32	85,32630
1024	2	32	89,44880
1024	4	32	88,55960
1024	8	32	87,21995
1024	16	32	85,53420
1024	32	32	83,13055
2048	2	32	84,15205
2048	4	32	83,53840
2048	8	32	82,86380
2048	16	32	81,89035
2048	32	32	82,28870

Tabelle 3.2: Die Ausführungszeiten der OpenACC-Implementierung der Pi-Approximation (siehe Code 3.2) für $N = 10^9$ mit unterschiedlichen Konfigurationen für num_gangs, num_workers und vector_length. Die Messwerte wurden mit der Grafikkarte NVIDIA® Quadro® P6000 des Rechners ls4gpu2 bestimmt. Mit rot ist die höchste und mit grün die geringste Ausführungszeit markiert. Sie unterscheiden sich um den Faktor 12.

3.1.5 Zusammenfassung

Um die Performance einer OpenACC-Implementierung besser einschätzen zu können, wurde neben der sequentiellen Variante auch eine OpenMP-Version der Approximation von Pi implementiert. Anschließend wurden alle Implementierungen mit einem N zwischen 10^3 und 10^{10} ausgeführt und die Ausführungszeiten bestimmt. Die Tabelle 3.3 fasst diese Ergebnisse zusammen.

Während die sequentielle Variante bei kleinem $N \leq 10^5$ überzeugt, erreicht bei größerem N (zwischen 10^5 und 10^6) die OpenMP-Implementierung die schnellste Ausführungszeit. Wird N noch weiter erhöht, so kann die komplette Power der GPU genutzt werden und somit ist bei $N \geq 10^8$ die OpenACC-Variante für die GPU am schnellsten. Erstaunlich ist besonders die langsame Ausführungszeit der OpenACC-Implementierung für die CPU. Der Grund ist die schlechte Unterstützung durch den gcc-Compiler.

N	**Sequentiell** CPU: Intel® Xeon® E5-2699 v4 Threads: 1 GCC-Compiler 7.1.0	**OpenMP** CPU: Intel® Xeon® E5-2699 v4 Threads: 20 GCC-Compiler 7.1.0	**OpenACC** CPU: Intel® Xeon® E5-2699 v4 ng=2048,nw=32,vl=32 GCC-Compiler 7.1.0	**OpenACC** GPU: NVIDIA® Quadro® P6000 ng=2048,nw=32,vl=32 GCC-Compiler 7.1.0
10^3	0,0077	0,3885	0,0056	2,5667
10^4	0,0750	0,8725	0,0418	2,5877
10^5	0,3332	0,4500	0,3279	2,5242
10^6	5,7357	1,8829	2,8381	2,6817
10^7	40,5872	3,5937	25,5672	2,5462
10^8	276,2175	27,0945	254,3576	9,8536
10^9	2.603,5820	174,4920	2.544,1908	82,1389
10^{10}	25.659,3237	1.704,1143	25.429,9191	807,2331

Tabelle 3.3: Die Ausführungszeiten von unterschiedlichen Implementierungen zur Approximation von Pi auf dem Rechner ls4gpu2. Die Tabelle zeigt jeweils den Mittelwert aus 20 Ausführungen für N zwischen 10^3 und 10^{10} in Millisekunden (ms). Die Abkürzungen ng, nw und vl stehen für die Anzahl an num_gangs (ng), num_workers (nw) und vector_length (vl). Die einzelnen Implementierungen befinden sich im Ordner: code/pi/

Anmerkung: Um die in Code 3.2 vorgestellte, parallele OpenACC-Implementierung für eine GPU auf der CPU des Host-Systems ausführen zu können, muss lediglich die Codezeile 10 wie folgt ersetzt werden: `acc_set_device_num(0, acc_device_host)`. Weitere Details zur Funktion acc_set_device_num() können im Abschnitt 4.2.1.4 nachgelesen werden.

3.2 Speichertransfer zwischen Host und Device

In diesem Abschnitt wird mit Hilfe von vier kurzen Beispielen ein Überblick zu den wichtigsten Methoden zum Speichertransfer zwischen Host und Device gegeben. In jedem Beispiel wird ein auf dem Host-System allokiertes und initialisiertes Array zum Device geschickt,

um dort jeden Array-Eintrag um eins zu erhöhen. Abschließend wird das Array zurück zum Host transferiert. Die Speichertransfers und Berechnungen werden je nach Beispiel mit synchronen und asynchronen Methoden realisert. Alle Beispiele sind in der Datei: `code/openacc_dataTransfer.c` zu finden.

3.2.1 Synchron

Unter einem **synchronen** Nachrichtenaustausch wird eine blockierende Kommunikation verstanden, d.h. der Sender wartet solange, bis eine Methode *send* (oder etwas Ähnliches) mit einem Ergebnis zurückkehrt. [MBW08, S. 308] Während der Wartezeit kann der Sender nichts anderes tun als auf eine Antwort zu warten. Im Folgenden wird diese Art der Kommunktion mit Hilfe von OpenACC-Pragmas und OpenACC-Funktionen gezeigt.

3.2.1.1 Mit OpenACC-Pragmas

Im Allgemeinen sind alle Pragma-Klauseln zum Speichertransfer zwischen Host-System und Device synchron umgesetzt. Dazu zählen u.a. folgende Klauseln: *copy*(...), *copyin*(...) und *copyout*(...). In einem ersten Beispiel konzentrieren wir uns auf die Verwendung der Klausel *copy*(...), die sowohl in einem Data-Pragma verwendet werden kann

```
#pragma acc data copy(a[0:N])
```

als auch direkt innerhalb eines Parallel-Loop-Pragmas

```
#pragma acc parallel loop copy(a[0:N])
```

In beiden Varianten wird zunächst ein auf dem Host-System befindliches Array a der Länge N auf das Device übertragen und anschließend zurückkopiert. Dabei werden beide Speichertransfers synchron ausgeführt. Der Host kann also erst die nächste Instruktion ausführen, wenn das Array a erfolgreich vom Device zum Host zurückkopiert wurde.

Im Code 3.3 wird zunächst ein Array a der Länge N auf dem Host-System allokiert und initialisiert (siehe Codezeilen 2 bis 4). Danach wird das Parallel-Loop-Pragma mit copy-Klausel genutzt (siehe Codezeilen 11 bis 13), um:

1. Das Array a vom Host-System zum Device zu kopieren. → copy(a[0:N])
2. Jeden Array-Eintrag auf dem Device um eins zu erhöhen. → parallel loop
3. Das Array a vom Device zum Host-System zurückzukopieren. → copy(a[0:N])

Bei dieser Variante verschmelzen also Speichertransfer und parallele Berechnung.

```
1     // Speicher auf dem Host-System allokieren und initialiseren
2     float *a = (float*) malloc(N* sizeof(float));
3     for(long long i=0; i<N; i++)
4         a[i] = i;
5
6     /*
7      * 1) Array a vom Host zum Device kopieren [copy(a[0:N])]
8      * 2) jeden Array-Eintrag auf dem Device um eins erhoehen
9      * 3) Array a vom Device zum Host kopieren [copy(a[0:N])]
10    */
11    #pragma acc parallel loop copy(a[0:N])
12    for(long long i=0; i<N; i++)
13        a[i] += 1;
14
15    // allokierten Speicher auf dem Host-System freigeben
16    free(a);
```

Code 3.3: Synchroner Speichertransfer mit OpenACC-Pragma. Im Code verschmelzen parallele Berechnung und Speichertransfer.

```
1     // Speicher auf dem Host-System allokieren und initialiseren
2     float *a = (float*) malloc(N* sizeof(float));
3     for(long long i=0; i<N; i++)
4         a[i] = i;
5
6     // 1) Array a vom Host zum Device kopieren
7     #pragma acc data copyin(a[0:N])
8
9     // 2) jeden Array-Eintrag auf dem Device um eins erhoehen
10    #pragma acc parallel loop
11    for(long long i=0; i<N; i++)
12        a[i] += 1;
13
14    // 3) Array a vom Device zum Host kopieren
15    #pragma acc data copyout(a[0:N])
16
17    // allokierten Speicher auf dem Host-System freigeben
18    free(a);
```

Code 3.4: Synchroner Speichertransfer mit OpenACC-Pragmas. Die parallele Berechung und der Speichertransfer wurden getrennt.

Das Speichertransfer und parallele Berechnung auch getrennt werden können, zeigt der Code 3.4. Auch diese Implementierung besteht, nach der Allokation und Initialisierung eines Array a auf dem Host-System (siehe Codezeilen 2 bis 4), aus drei Schritten:

1. Das Array a vom Host zum Device kopieren. Dazu wird ein Data-Pragma mit copyin-Klausel verwendet (siehe Codezeile 7).

2. Jeden Array-Eintrag auf dem Device um eins erhöhen (siehe Codezeilen 10 bis 12).

3. Das Array a vom Device zum Host-System zurückkopieren. Dazu wird ein Data-Pragma mit copyout-Klausel verwendet (siehe Codezeile 15).

Zeitmessung

Zur Bestimmung der einzelnen Speicherübertragungszeiten von Code 3.3 wurde das Kommandozeilentool *nvprof* [Har13] verwendet. Dabei wurde für jede Arraygröße N der Durchschnitt über 1000 Iterationen bestimmt. Die Resultate sind in Tabelle 3.4 zu sehen.

Die Grundlage für die einzelnen Speichertransfers bilden drei CUDA memcpy-Funktionen. Während der Ausführung von Code 3.3 wurden *Host to Host* (cuMemcpy) und *Device to Host* (cuMemcpyDtoH) jeweils nur einmal genutzt, während *Host to Device* (cuMemcpyHtoD) laut nvprof (Profilingtool, siehe Abschnitt 6.1) zweimal notwendig war, wobei der Eintrag in Tabelle 3.4 bereits die Summe beider Aufrufe darstellt.

N	**CUDA memcpy**			**Berechnungsanteil**
	Host to Host	Host to Device	Device to Host	
10^3	3,16 μs	1,74 μs	0,56 μs	92,56 %
10^4	3,11 μs	5,04 μs	3,27 μs	85,49 %
10^5	3,07 μs	35,92 μs	30,73 μs	49,74 %
10^6	2,73 μs	345,96 μs	429,05 μs	12,96 %
10^7	2,75 μs	3.470,00 μs	5.010,00 μs	6,44 %
10^8	3.22 μs	43.320,00 μs	32.340,00 μs	6,01 %

Tabelle 3.4: Speichertransferzeit von Code 3.3 für unterschiedlich große N aufgeschlüsselt in die drei CUDA memcpy-Varianten und unter Angabe der Berechnungsanteile. Verwendete Workstation: ls4gpu2 (Systemspezifikationen siehe Anhang A).

3.2.1.2 Mit OpenACC-Funktionen

Zur Realiserung eines synchronen Speicheraustausch zwischen Host und Device können neben OpenACC-Pragmas auch OpenACC-Funktionen verwendet werden. Ein Beispiel ist in Code 3.5 zu sehen.

Um OpenACC-Funktionen, zum Beispiel `acc_malloc()`, `acc_memcpy_to_device()`, `acc_memcpy_from_device()` und `acc_free()`, nutzen zu können, muss die Headerdatei `openacc.h` eingebunden werden.

```
1   #include <openacc.h> // fuer OpenACC-Funktionen
2
3   // Speicher auf dem Host-System allokieren und initialiseren
4   float *a_host = (float*) malloc(N * sizeof(float));
5   for(long long i=0; i<N; i++)
6       a_host[i] = i;
7
8   // Speicher auf dem Device allokieren
9   float *a_dev = acc_malloc(N * sizeof(float));
10
11  // Array-Eintraege vom Host zum Device transferieren
12  acc_memcpy_to_device(a_dev, a_host, N * sizeof(float));
13
14  // jeden Array-Eintrag um eins erhoehen
15  #pragma acc parallel loop
16  for(long long i=0; i<N; i++)
17      a_dev[i] += 1;
18
19  // Array-Eintraege vom Device zum Host transferieren
20  acc_memcpy_from_device(a_host, a_dev, N * sizeof(float));
21
22  // allokierten Speicher auf dem Device freigeben
23  acc_free(a_dev);
24
25  // allokierten Speicher auf dem Host-System freigeben
26  free(a_host);
```

Code 3.5: Synchroner Speichertransfer mit OpenACC-Funktionen.

Nach der Allokation und Initialisierung eines Array *a_host* auf dem Host-System (siehe Codezeilen 4 bis 6 im Code 3.5), wird ein Array *a_dev* mit gleicher Länge auf dem Device mit der OpenACC-Funktion `acc_malloc()` (weitere Details zur Funktion siehe Abschnitt 4.2.2.1) allokiert (siehe Codezeile 9). Anschließend werden die Einträge des *a_host*-Arrays mit der OpenACC-Funktion `acc_memcpy_to_device()` (weitere Details zur Funktion siehe Abschnitt 4.2.3.1) in das im Gerätespeicher befindliche Array *a_dev* kopiert (siehe Codezeile 12). Jetzt kann jeder Eintrag des *a_dev*-Arrays um eins erhöht werden (siehe Codezeile 15 bis 17). Dies geschieht genau wie zuvor im Code 3.3 parallel auf dem Device und mit Hilfe eines `#pragmas`. Danach werden die Einträge des *a_dev*-Arrays zurück auf das Host-System (genauer: in das Array *a_host*) kopiert. Dazu wird die Funktion `acc_memcpy_from_device()` (weitere Details zur Funktion siehe Abschnitt 4.2.3.2) verwendet (siehe Codezeile 20). Abschließend werden beide Arrays freigeben, das Array *a_dev* mit der OpenACC-Funktion `acc_free()` (weitere Details zur Funktion siehe Abschnitt 4.2.2.2)

in Codezeile 23 und das Array a_host mit der Funktion $free()$ in Codezeile 26.

Zeitmessung

Zur Bestimmung der einzelnen Speicherübertragungszeiten von Code 3.5 wurde das Kommandozeilentool *nvprof* [Har13] verwendet. Dabei wurde für jede Arraygröße N der Durchschnitt über 1000 Iterationen bestimmt. Die Resultate sind in der Tabelle 3.5 zu sehen.

Die Grundlage für die einzelnen Speichertransfers bilden drei CUDA memcpy-Funktionen. Während der Ausführung von Code 3.5 wurden *Host to Host* (cuMemcpy) und *Device to Host* (cuMemcpyDtoH) jeweils nur einmal genutzt, während *Host to Device* (cuMemcpyHtoD) laut nvprof (Profilingtool, siehe Abschnitt 6.1) dreimal notwendig war, wobei der Eintrag in Tabelle 3.5 bereits die Summe aller drei Aufrufe darstellt.

N	**CUDA memcpy**			**Berechnungsanteil**
	Host to Host	Host to Device	Device to Host	
10^3	2,88 μs	2,56 μs	0,72 μs	91,55 %
10^4	3,17 μs	5,91 μs	3,42 μs	84,12 %
10^5	2,64 μs	36,59 μs	30,78 μs	49,65 %
10^6	2,81 μs	348,00 μs	421,14 μs	13,13 %
10^7	2,77 μs	3.416,00 μs	5.021,60 μs	6,49 %
10^8	3,01 μs	42.790,00 μs	32.081,00 μs	6,10 %

Tabelle 3.5: Speichertransferzeit von Code 3.5 für unterschiedlich große N aufgeschlüsselt in die drei CUDA memcpy-Varianten und unter Angabe der Berechnungsanteile. Verwendete Workstation: ls4gpu2 (Systemspezifikationen siehe Anhang A).

3.2.2 Asynchron

Ein **asynchroner** Nachrichtenaustausch bedeutet, dass der Sender nach dem Absenden der Nachricht mit Hilfe einer Methode *send* nicht blockiert und bis zum Eintreffen eines Ergebnisses andere Aufgaben erledigen kann. Die Antworten müssen also später irgendwie eingesammelt werden. [MBW08, S. 309]

3.2.2.1 Mit OpenACC-Pragmas

Der Code 3.6 zeigt, wie auf Basis von OpenACC-Pragmas eine asynchrone Kommunikation zwischen Host-System und Device realisiert werden kann. Zunächst wird ein Array der Länge N auf dem Host-System allokiert und initialisiert (siehe Codezeilen 2 bis 4 im Code 3.6). Anschließend kann, ähnlich wie im synchronen Fall (siehe Abschnitt 3.2.1.1), ein OpenACC-Pragma für den Datentransfer und die parallele Berechnung auf dem Device verwendet werden (siehe Codezeilen 11 bis 13). Der einzige Unterschied zum Code 3.3 ist

Verwendung von `async(0)` mit dem wir eine asynchrone Ausführung verlangen und diese mit der Nummer 0 versehen. Das bedeutet, dass die parallele Berechnung auf dem Device nicht blockierend ist und wir demzufolge auf die Fertigstellung warten müssen. Für das Warten kann das Pragma `#pragma acc wait(0)` genutzt werden (siehe Codezeile 19), wobei wir mit der Angabe der Zahl 0 den Bezug zur vorher vergebenen Nummer sehen. Das bedeutet insbesondere bei mehreren asynchronen Operationen, die miteinander synchronisiert werden müssen, ist auf eine korrekte Angabe der Nummer zu achten. Abschließend wird das Array auf dem Host-System freigeben (Codezeile 22).

```
1   // Speicher auf dem Host-System allokieren und initialiseren
2   float *a = (float*) malloc(N * sizeof(float));
3   for(long long i=0; i<N; i++)
4       a[i] = i;
5
6   /*
7    * 1) Array asynchron vom Host zum Device transferieren
       [copy(a[0:N])]
8    * 2) asynchron jeden Array-Eintrag um eins erhoehen
9    * 3) Array asynchron vom Device zum Host transferieren
       [copy(a[0:N])]
10  */
11  #pragma acc parallel loop async(0) copy(a[0:N])
12  for(long long i=0; i<N; i++)
13      a[i] += 1;
14
15  /*
16   * Warte auf die Fertigstellung der Transfers zwischen Host
17   * und Device, sowie die Device-Berechnungen
18  */
19  #pragma acc wait(0)
20
21  // allokierten Speicher auf dem Host-System freigeben
22  free(a);
```

Code 3.6: Asynchroner Speichertransfer mit OpenACC-Pragmas.

Zeitmessung

Zur Bestimmung der einzelnen Speicherübertragungszeiten von Code 3.6 wurde das Kommandozeilentool *nvprof* [Har13] verwendet. Dabei wurde für jede Arraygröße N der Durchschnitt über 1000 Iterationen bestimmt. Die Resultate sind in der Tabelle 3.6 zu sehen.

Die Grundlage für die einzelnen Speichertransfers bilden drei CUDA memcpy-Funktionen. Während der Ausführung von Code 3.6 wurden *Host to Host* (cuMemcpy) und *Device to*

Host (cuMemcpyDtoH) jeweils nur einmal genutzt, während *Host to Device* (cuMemcpyHtoD) laut nvprof (Profilingtool, siehe Abschnitt 6.1) zweimal notwendig war, wobei der Eintrag in Tabelle 3.6 bereits die Summe beider Aufrufe darstellt.

N	**CUDA memcpy**			**Berechnungsanteil**
	Host to Host	Host to Device	Device to Host	
10^3	3,31 μs	1,73 μs	0.98 μs	91,64 %
10^4	3,24 μs	5,04 μs	3,69 μs	84,73 %
10^5	17,70 μs	35,94 μs	31,12 μs	44,80 %
10^6	82,10 μs	342,00 μs	403,03 μs	12,12 %
10^7	69,83 μs	3.474,00 μs	4.902.10 μs	6,46 %
10^8	36.54 μs	42.810,00 μs	32.095,00 μs	6.32 %

Tabelle 3.6: Speichertransferzeit von Code 3.6 für unterschiedlich große N aufgeschlüsselt in die drei CUDA memcpy-Varianten und unter Angabe der Berechnungsanteile. Verwendete Workstation: ls4gpu2 (Systemspezifikationen siehe Anhang A).

3.2.2.2 Mit OpenACC-Funktionen

Ähnlich wie im synchronen Fall (siehe Abschnitt 3.2.1.2) kann auch ein asynchroner Datentransfer und die parallele Berechnung auf dem Device mit Hilfe von OpenACC-Funktionen realisiert werden (siehe Code 3.7). Um jedoch OpenACC-Funktionen wie `acc_malloc()`, `acc_memcpy_to_device_async()`, `acc_memcpy_from_device_async()`, `acc_wait()` und `acc_free()` nutzen zu können, muss die Headerdatei `openacc.h` eingebunden werden.

Nach der Allokation und Initialisierung eines Array *a_host* auf dem Host-System (siehe Codezeilen 4 bis 6 im Code 3.7), wird ein Array *a_dev* mit gleicher Länge auf dem Device mit der OpenACC-Funktion `acc_malloc()` (weitere Details zur Funktion siehe Abschnitt 4.2.2.1) allokiert (siehe Codezeile 9). Anschließend werden die Einträge des *a_host*-Arrays mit der asynchronen Funktion `acc_memcpy_to_device_async()` (weitere Details zur Funktion siehe Abschnitt 4.2.3.1) in das im Gerätespeicher befindliche Array *a_dev* kopiert und die Operation mit der Nummer 0 versehen (siehe Codezeile 12). Danach muss auf die Fertigstellung der asynchronen Operation 0 gewartet werden. Realisiert wird das ganze mit der Funktion `acc_wait(0)` (weitere Details zur Funktion siehe Abschnitt 4.2.4.1). Jetzt kann jeder Eintrag des *a_dev*-Arrays um eins erhöht werden (siehe Codezeile 21 bis 23). Dies geschieht genau wie im Code 3.6 parallel auf dem Device und mit Hilfe eines `#pragmas`, wobei wir auch hier auf das Ende der asynchronen Berechnungen warten müssen (siehe `acc_wait()` in Codezeile 26). Danach werden die Einträge des *a_dev*-Arrays zurück auf das Host-System (genauer: in das Array *a_host*) kopiert. Dazu wird die asynchrone Funktion `acc_memcpy_from_device_async()` (weitere Details zur Funktion siehe Abschnitt 4.2.3.2) verwendet (siehe Codezeile 29) und auch hier mit `acc_wait()` auf das

```
1  #include <openacc.h> // fuer OpenACC-Funktionen
2
3  // Speicher auf dem Host-System allokieren und initialiseren
4  float *a_host = (float*) malloc(N * sizeof(float));
5  for(long long i=0; i<N; i++)
6      a_host[i] = i;
7
8  // Speicher auf dem Device allokieren
9  float *a_dev = acc_malloc(N * sizeof(float));
10
11 // Array-Eintraege asynchron vom Host zum Device transferieren
12 acc_memcpy_to_device_async(a_dev, a_host, N*sizeof(float), 0);
13
14 /*
15  * auf die Fertigstellung des asynchronen Speichertransfers
16  * zwischen Host und Device warten
17 */
18 acc_wait(0);
19
20 // jeden Array-Eintrag um eins erhoehen
21 #pragma acc parallel loop async(1)
22 for(long long i=0; i<N; i++)
23     a_dev[i] += 1;
24
25 // auf die Fertigstellung der asynchronen Berechnung warten
26 acc_wait(1);
27
28 // Array-Eintraege asynchron vom Device zum Host transferieren
29 acc_memcpy_from_device_async(a_host, a_dev, N*sizeof(float),
       2);
30
31 /*
32  * auf die Fertigstellung des asynchronen Speichertransfers
33  * zwischen Device und Host warten
34 */
35 acc_wait(2);
36
37 // allokierten Speicher auf dem Device freigeben
38 acc_free(a_dev);
39
40 // allokierten Speicher auf dem Host-System freigeben
41 free(a_host);
```

Code 3.7: Asynchroner Speichertransfer mit OpenACC-Funktionen.

Ende des Kopiervorgangs gewartet (siehe Codezeile 35). Abschließend werden beide Arrays freigeben, das Array *a_dev* mit der OpenACC-Funktion `acc_free()` (weitere Details zur Funktion siehe Abschnitt 4.2.2.2) in Codezeile 38 und das Array *a_host* mit der Funktion *free*() in Codezeile 41.

Zeitmessung

Asynchrone Speichertransfers auf Basis von OpenACC-Funktionen, wie beispielsweise `acc_memcpy_to_device_async()` und `acc_memcpy_from_device_async()`, werden in der gcc-Version 8 und früher nicht unterstützt. Daher können an dieser Stelle keine Messwerte präsentiert werden.

3.2.3 Zusammenfasssung

Wie wir in den vergangenen Abschnitten gesehen haben, kann ein Speichertransfer zwischen Host-System und Device in OpenACC sowohl mit synchronen als auch mit asynchronen Methoden realisiert werden. Beide Varianten wurden mit OpenACC-Pragmas und -Funktionen umgesetzt.

Abschließend soll nun ein kurzer Blick auf die Ausführungszeiten der einzelnen Implementierungen geworfen werden. Dazu wurden alle Implementierungen:

Synchron:

- OpenACC-Pragmas (Code 3.3)
- OpenACC-Funktionen (Code 3.5)

Asynchron:

- OpenACC-Pragmas (Code 3.6)
- OpenACC-Funktionen (Code 3.7)

mit unterschiedlich großen Arrays (Arraylänge N) ausgeführt und der Durchschnitt über 1000 Iterationen bestimmt. Die Resultate sind in Tabelle 3.7 zu sehen.

N	**Synchron**		**Asynchron**	
	Pragmas (Code 3.3)	Funktionen (Code 3.5)	Pragmas (Code 3.6)	Funktionen (Code 3.7)
10^3	0,4170	0,6876	0,3775	–
10^4	0,4446	0,7588	0,4437	–
10^5	0,6489	0,8677	0,5892	–
10^6	2,0483	2,2268	2,0442	–
10^7	26,6609	26,7654	26,5486	–
10^8	270,3921	270,4488	270,3931	–

Tabelle 3.7: Ausführungszeiten (in ms) im Überblick. Verwendete Workstation: ls4gpu2 (Systemspezifikationen siehe Anhang A).

Bereits beim ersten Blick auf die Tabelle 3.7 fällt auf, dass es zwischen den einzelnen Implementierungen zeitlich gesehen nur kleine Unterschiede gibt. Im synchronen Fall ist die Pragma-Implementierung etwas schneller als die Umsetzung mit OpenACC-Funktionen, wobei der Unterschied bei großen N (z.B. $N = 10^8$) nur gering ist. Auch der Vergleich zwischen synchroner und asynchroner Pragma-Variante zeigt, dass die zusätzlich benötigten Synchronisationen im asynchronen Fall (realisiert mit `#pragma acc wait`) keine Performanceverluste zur Folge haben.

Kapitel 4

OpenACC im Detail

In diesem Kapitel werden die wichtigsten OpenACC-Umgebungsvariablen, -Funktionen und -Pragmas vorgestellt. Für detaillierte Informationen wird auf die offizielle OpenACC-Spezifikation verwiesen. Die nachfolgenden Erläuterungen beziehen sich auf die OpenACC-Spezifikation 2.6 vom November 2017. [Ope18a]

4.1 Wichtige Umgebungsvariablen und Makros

4.1.1 ACC_DEVICE_TYPE

Die Umgebungsvariable ACC_DEVICE_TYPE gibt den Gerätetyp an, mit dem die Verbindung hergestellt werden soll. In der gcc-Version 7.1.0 sind unter dem Typ acc_device_t folgenden Gerätetypen definiert:

- `acc_device_none`
- `acc_device_default`
- `acc_device_host`
- `acc_device_not_host`
- `acc_device_nvidia`

In den folgenden Abschnitten wird diese Umgebungsvariable mit $acc-device-type-var$ bezeichnet.

4.1.2 ACC_DEVICE_NUM

Die Umgebungsvariable ACC_DEVICE_NUM gibt die Gerätenummer an, mit der die Verbindung hergestellt werden soll. Es ist also ein Integer zwischen 0 bis (Anzahl Geräte vom Typ ACC_DEVICE_TYPE)-1.

In den folgenden Abschnitten wird diese Umgebungsvariable mit $acc-device-num-var$ bezeichnet.

4.1.3 _OPENACC

Das _OPENACC-Präprozessor-Makro ist so definiert, dass es den Wert `yyyymm` hat, wenn die OpenACC-Direktiven aktiviert sind. Die hier beschriebene Version hat den Wert 201711.

4.2 OpenACC-Funktionen

Um weitere OpenACC-spezifische Funktionen nutzen zu können, muss die Headerdatei `openacc.h` eingebunden werden.

```
#include <openacc.h>
```

4.2.1 Computedevice abfragen und auswählen

In diesem Abschnitt werden die wichtigsten Funktionen zur Abfrage und Auswahl eines Computedevice vorgestellt.

4.2.1.1 acc_get_num_devices

Die Routine *acc_get_num_devices* gibt die Anzahl der Geräte des angegebenen Typs zurück, die an den Host angeschlossen sind. Das Argument sagt aus, welche Art von Geräten gezählt werden soll.

Format:

```
int acc_get_num_devices( acc_device_t );
```

Beispiel: Die Anzahl an NVIDIA-GPUs in einem Computer bestimmen.

```
int num_nvidia_gpus = acc_get_num_devices( acc_device_nvidia );
```

4.2.1.2 acc_set_device_type

Die Routine *acc_set_device_type* teilt zur Laufzeit dem Programm mit, welchen Gerätetyp sie unter den verfügbaren Geräten verwenden soll und setzt den Wert von *acc-device-type-var* für den aktuellen Thread. Um ein spezielles Gerät vom Typ *acc_device_t* auszuwählen, kann die Routine *acc_set_device_num* (siehe Abschnitt 4.2.1.4) genutzt werden.

Format:

```
void acc_set_device_type( acc_device_t );
```

Beispiel: Den Host als Gerät auswählen.

```
acc_set_device_type( acc_device_host );
```

4.2.1.3 acc_get_device_type

Die Routine *acc_get_device_type* gibt den Wert von *acc-device-type-var* für den aktuellen Thread zurück, um dem Programm mitzuteilen, welcher Gerätetyp für die Ausführung der nächsten Compute-Region verwendet wird.

Format:

```
acc_device_t acc_get_device_type( void );
```

Beispiel:

```
acc_device_t device_type = acc_get_device_type( );
```

4.2.1.4 acc_set_device_num

Die Routine *acc_set_device_num* teilt zur Laufzeit dem Programm mit, welches Gerät unter den angeschlossenen Geräten des angegebenen Typs für Compute- oder Datenbereiche im aktuellen Thread verwendet werden soll und setzt den Wert von *acc-device-num-var*. Wenn der Wert von *devicenum* negativ ist, wird das Programm auf das Standardverhalten, das in der Implementierung definiert ist, zurückgesetzt. Wenn der Wert des zweiten Arguments 0 ist, bezieht sich die gewählte Gerätenummer auf alle angeschlossenen Gerätetypen.

Format:

```
acc_set_device_num( int, acc_device_t );
```

Beispiel: Wähle die erste NVIDIA-GPU.

```
acc_set_device_num( 0, acc_device_nvidia );
```

4.2.1.5 acc_get_device_num

Die Routine *acc_get_device_num* gibt den Wert von *acc-device-num-var* für den aktuellen Thread zurück.

Format:

```
int acc_get_device_num( acc_device_t );
```

Beispiel:

```
acc_get_device_num( acc_device_nvidia );
```

4.2.1.6 acc_get_property und acc_get_property_string

Beide Funktionen werden in der gcc-Version 8 und früher nicht unterstützt.

Die Routinen *acc_get_property* und *acc_get_property_string* geben den Wert der angegebenen Eigenschaft zurück. *devicenum* und *devicetype* spezifizieren das Gerät, das abgefragt wird. Wenn *devicetype* den Wert *acc_device_current* hat, dann wird *devicenum* ignoriert und der Wert der Eigenschaft für das aktuelle Gerät zurückgegeben. *property* ist eine Aufzählungskonstante, definiert in *openacc.h*. Ganzzahlige Eigenschaften werden von *acc_get_property* und String-Eigenschaften von *acc_get_property_string* zurückgegeben.

Die unterstützten *property*-Werte sind in der folgenden Tabelle angegeben.

property	Rückgabetyp	Rückgabewert
acc_property_memory	integer	Größe des Gerätespeichers in Bytes
acc_property_free_memory	integer	freier Gerätespeicher in Bytes
acc_property_name	string	Gerätename
acc_property_vendor	string	Gerätehersteller
acc_property_driver	string	Gerätetreiber-Version

Tabelle 4.1: Unterstützte *property*-Werte

Format:

```
size_t acc_get_property( int devicenum, acc_device_t devicetype,
acc_device_property_t property );

const char* acc_get_property_string( int devicenum, acc_device_t
devicetype, acc_device_property_t property );
```

Beispiel 1: Den freien Speicher auf der ersten NVIDIA-GPU erhalten.

```
size_t free_memory = acc_get_property( 0, acc_device_nvidia,
acc_property_free_memory );
```

Beispiel 2: Den Namen der ersten NVIDIA-GPU erhalten.

```
const char *gpu_name = acc_get_property( 0, acc_device_nvidia,
acc_property_name );
```

4.2.1.7 acc_init

Die Routine *acc_init* teilt zur Laufzeit dem Programm mit, dass der angegebene Gerätetyp initialisiert werden soll. Dies kann verwendet werden, um bei der Erfassung von Performance-Statistiken die Initialisierungskosten von den Berechnungskosten zu isolieren.

Format:

```
void acc_init( acc_device_t );
```

Beispiel: Eine NVIDIA-GPU initialisieren.

```
acc_init( acc_device_nvidia );
```

4.2.1.8 acc_shutdown

Die Routine *acc_shutdown* teilt zur Laufzeit dem Programm mit, dass es die Verbindung zum gegebenen Gerät trennen soll und alle Runtime-Ressourcen frei gibt.

Format:

```
void acc_shutdown( acc_device_t );
```

Beispiel: Die Verbindung zur NVIDIA-GPU trennen.

```
acc_shutdown( acc_device_nvidia );
```

4.2.1.9 acc_on_device

Die Routine *acc_on_device* kann verwendet werden, um verschiedene Pfade auszuführen, je nachdem, ob der Code auf dem Host oder auf einem Gerät ausgeführt wird. Wenn die Routine *acc_on_device* ein Argument bekommt, dass zur Übersetzungszeit bekannt ist, wertet sie zur Compile-Zeit eine Konstante aus. Das Argument muss einem der definierten Gerätetypen entsprechen. Wenn das Argument *acc_device_host* ist, gibt diese Routine ein Wert ungleich Null zurück.

Format:

```
int acc_on_device( acc_device_t );
```

Beispiel:

```
int device_or_host = acc_on_device ( acc_device_nvidia );
```

4.2.2 Speicher allokieren und freigeben

In diesem Abschnitt werden Funktionen zur Allokationen und Deallokation von Speicher auf einem Gerät präsentiert.

Im Folgenden ist h_void* ein void-Zeiger auf den Hostspeicher und d_void* ein void-Zeiger auf den Gerätespeicher. Formell bedeutet das:

```
#define h_void void     und     #define d_void void
```

4.2.2.1 acc_malloc

Die Routine *acc_malloc* allokiert Speicher auf dem Gerät.

Format:

```
d_void* acc_malloc( size_t );
```

Beispiel: Ein double-Array der Größe 10 auf dem vorher gewählten Gerät (siehe Abschnitt 4.2.1) allokieren (*acc_malloc*) und danach wieder freigeben (*acc_free*).

```
double* array = (double*) acc_malloc( 10 * sizeof(double) );
acc_free( array );
```

4.2.2.2 acc_free

Die Routine *acc_free* gibt Speicher auf dem Gerät frei.

Format:

```
void acc_free( d_void* );
```

Beispiel: Ein double-Array der Größe 10 auf dem vorher gewählten Gerät (siehe Abschnitt 4.2.1) allokieren (*acc_malloc*) und danach wieder freigeben (*acc_free*).

```
double* array = (double*) acc_malloc( 10 * sizeof(double) );
acc_free( array );
```

4.2.2.3 acc_create

Die *acc_create*-Routinen testen, ob die Daten bereits auf dem Gerät vorhanden sind; wenn nicht, allokieren sie so viel Speicher auf dem Gerät wie für den angegebenen Hostspeicher notwendig ist. Die _async-Version dieser Funktion kann die Datenzuweisung asynchron ausführen. Das async-Argument entspricht der asynchronen Warteschlangennummer in die die Operation eingefügt wird. Die synchrone Version kehrt erst nach der Zuweisung der Daten zurück.

Format:

```
d_void* acc_create( h_void*, size_t );
void acc_create_async( h_void*, size_t, int async );
```

Beispiel: Speicher auf dem Gerät allokieren (*acc_create*). Anschließend Speicher wieder freigeben (*acc_delete*).

```
double h_array[10];
double* d_array = acc_create( h_array, 10 * sizeof(double) );
acc_delete( h_array, 10 * sizeof(double) );
```

4.2.2.4 acc_delete

Die *acc_delete*-Routinen deallokieren den Speicher des Geräts, der zu dem angegebenen lokalen Speicher gehört.

Format:

```
void acc_delete( h_void*, size_t );
void acc_delete_async( h_void*, size_t, int async );
```

Beispiel: Speicher auf dem Gerät allokieren (*acc_create*). Anschließend Speicher wieder freigeben (*acc_delete*).

```
double h_array[10];
double* d_array = acc_create( h_array, 10 * sizeof(double) );
acc_delete( h_array, 10 * sizeof(double) );
```

4.2.2.5 acc_copyin

Die *acc_copyin*-Routinen testen, ob die Daten bereits auf dem Gerät vorhanden sind; wenn nicht, allokieren sie so viel Speicher auf dem Gerät wie für den angegebenen Hostspeicher notwendig ist. Anschließend kopieren sie die Daten in zuvor allokierten Gerätespeicher. Die _async-Version dieser Funktion führt alle Datenübertragungen asynchron. Das heißt, die Funktion kann zurückkehren, bevor die Daten übertragen wurden. Das async-Argument entspricht der asynchronen Warteschlangennummer in die die Operation eingefügt wird. Die synchrone Version kehrt erst nach vollständiger Übertragung der Daten zurück.

Format:

```
d_void* acc_copyin( h_void*, size_t );
void acc_copyin_async( h_void*, size_t, int );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Danach Daten zurück auf das Hostsystem kopieren und den auf dem Gerät allokierten Speicher freigeben (zurückkopieren und freigeben: *acc_copyout*).

```
double h_array[10];
double* d_array = acc_copyin( h_array, 10 * sizeof(double) );
acc_copyout( h_array, 10 * sizeof(double) );
```

4.2.2.6 acc_copyout

Die *acc_copyout*-Routinen kopieren Daten vom Gerätespeicher in Speicher des Hostsystems und geben anschließend den Speicher auf dem Gerät frei.

Format:

```
void acc_copyout( h_void*, size_t );
void acc_copyout_async( h_void*, size_t, int async );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Danach Daten zurück auf das Hostsystem kopieren und den auf dem Gerät allokierten Speicher freigeben (zurückkopieren und freigeben: *acc_copyout*).

```
double h_array[10];
double* d_array = acc_copyin( h_array, 10 * sizeof(double) );
acc_copyout( h_array, 10 * sizeof(double) );
```

4.2.2.7 acc_update_device

Die *acc_update_device*-Routinen aktualisieren die Daten der Gerätekopie mit den Daten im Hostspeicher.

Format:

```
void acc_update_device( h_void*, size_t );
void acc_update_device_async( h_void*, size_t, int async );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Anschließend die Daten im Hostsystem ändern/updaten und die Änderungen auf das Array im Gerät weitergeben (*acc_update*).

```
double h_array[10];
double* d_array = acc_copyin( h_array, 10 * sizeof(double) );

/* h_array-Einträge ändern/updaten */

// nur die ersten 5 Einträge aktualisieren:
acc_update_device( h_array, 5 * sizeof(double) );
```

4.2.2.8 acc_update_self

Die *acc_update_self*-Routinen aktualisieren die Daten im Hostsystem mit den Daten im Gerätespeicher.

Format:

```
void acc_update_self( h_void*, size_t );
void acc_update_self_async( h_void*, size_t, int async );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Anschließend die Daten im Gerätespeicher ändern/updaten und die Änderungen an das Array im Hostsystem weitergeben (*acc_update_self*).

```
double h_array[10];
double* d_array = acc_copyin( h_array, 10 * sizeof(double) );
/* d_array-Einträge ändern/updaten */
acc_update_self( h_array, 10 * sizeof(double));
```

4.2.2.9 acc_deviceptr

Die Routine *acc_deviceptr* gibt den Gerätezeiger zurück, der zu der angegeben Hostadresse gehört.

Format:

```
d_void* acc_deviceptr( h_void* );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Dabei vergessen den zurückgegebenen *d_void*-Zeiger der Routine *acc_copyin* zu speichern. Deshalb mit *acc_deviceptr* den Zeiger auf den Geräte-Speicher erneut abfragen.

```
double h_array[10];
acc_copyin( h_array, 10 * sizeof(double) );
double* d_array = acc_deviceptr( h_array );
```

4.2.2.10 acc_hostptr

Die Routine *acc_hostptr* gibt den Hostzeiger zurück, der zu der angegeben Geräteadresse gehört.

Format:

```
h_void* acc_hostptr( d_void* );
```

Beispiel: Array vom Hostsystem auf das Gerät kopieren (*acc_copyin*). Danach mit *acc_hostptr* den Hostzeiger von der Geräteadresse *d_array* erhalten.

```
double h_array[10];
double* d_array = acc_copyin( h_array, 10 * sizeof(double) );
double* h_array2 = acc_hostptr( d_array );
```

4.2.3 Speichertransfer

In diesem Abschnitt werden Funktionen zum Datentransfer zwischen Host und Device und umgekehrt vorgestellt.

4.2.3.1 acc_memcpy_to_device

Die *acc_memcpy_to_device*-Routinen kopieren Daten aus dem lokalen Speicher in den Gerätespeicher.

Format:

```
void acc_memcpy_to_device( d_void* dest, h_void* src, size_t
bytes );
void acc_memcpy_to_device_async( d_void* dest, h_void* src,
size_t bytes, int async );
```

Beispiel: Array vom Host- in den Gerätespeicher kopieren.

```
double h_array[10];
double* d_array = (double*) acc_malloc( 10 * sizeof(double) );
/* h_array-Einträge ändern/updaten */
acc_memcpy_to_device( d_array, h_array, 10 * sizeof(double) );
acc_free( d_array );
```

4.2.3.2 acc_memcpy_from_device

Die *acc_memcpy_from_device*-Routinen kopieren Daten aus dem Gerätespeicher in den lokalen Speicher.

Format:

```
void acc_memcpy_from_device( h_void* dest, d_void* src, size_t
bytes );
void acc_memcpy_from_device_async( h_void* dest, d_void* src,
size_t bytes, int async );
```

Beispiel: Array vom Geräte- in den Hostspeicher kopieren.

```
double h_array[10];
double* d_array = (double*) acc_malloc( 10 * sizeof(double) );
/* d_array-Einträge ändern/updaten */
acc_memcpy_from_device( h_array, d_array, 10 * sizeof(double) );
acc_free( d_array );
```

4.2.3.3 acc_memcpy_device

Die *acc_memcpy_device*-Routinen kopieren Daten von einem Speicherplatz zu einem anderen Speicherplatz auf dem aktuellen Gerät.

Format:

```
void acc_memcpy_device( d_void* dest, d_void* src, size_t bytes
);
void acc_memcpy_device_async( d_void* dest, d_void* src, size_t
bytes, int async );
```

Beispiel: Eine zweite Kopie von einem Array im Gerätespeicher anlegen.

```
double* d_src = (double*) acc_malloc( 10 * sizeof(double) );
double* d_dest = (double*) acc_malloc( 10 * sizeof(double) );
acc_memcpy_device( d_dest, d_src, 10 * sizeof(double) );
acc_free( d_src ); acc_free( d_dest );
```

4.2.4 Synchronisation

In diesem Abschnitt werden einige Funktionen zur Synchronisation von Operationen vorgestellt. Dabei werden zwei Arten von Routinen unterschieden: das eigentliche Warten oder lediglich ein Test, ob eine oder mehrere Operationen bereits beendet sind.

4.2.4.1 acc_wait

Die Routine *acc_wait* wartet auf den Abschluss aller zugehörigen asynchronen Operationen auf dem aktuellen Gerät.

Format:

```
void acc_wait( int );
```

Beispiel: Auf die Fertigstellung der asynchronen Operation(en) mit der Nummer 1 warten.

```
acc_wait( 1 );
```

4.2.4.2 acc_wait_all

Die Routine *acc_wait_all* wartet auf den Abschluss aller asynchronen Operationen.

Format:

```
void acc_wait_all( );
```

4.2.4.3 acc_async_test

Die Routine *acc_async_test* testet, ob alle zugehörigen asynchronen Operationen auf dem aktuellen Gerät beendet sind. Sind einige asynchrone Operationen noch nicht abgeschlossen sind, gibt die *acc_async_test*-Routine eine Null zurück.

Format:

```
int acc_async_test( int );
```

Beispiel: Prüfen, ob die asynchrone(n) Operation(en) mit der Nummer 1 bereits beendet sind.

```
if(acc_async_test( 1 )) != 0) {...}
```

4.2.4.4 acc_async_test_all

Die Routine *acc_async_test_all* testet, ob alle asynchronen Operationen auf dem aktuellen Gerät beendet sind. Sind einige asynchrone Operationen noch nicht abgeschlossen sind, gibt die *acc_async_test_all*-Routine eine Null zurück.

Format:

```
int acc_async_test_all( );
```

Beispiel: Prüfen, ob alle asynchronen Operationen bereits beendet sind.

```
if(acc_async_test_all( )) != 0) {...}
```

4.3 OpenACC-Pragmas

Ein OpenACC-Pragma ist eine Anweisung, die einen nachfolgenden Codeblock oder eine nachfolgende Codezeile betrifft. In C und C++ werden sie mit dem `#pragma`-Mechanismus beschrieben. Jedes OpenACC-Pragma startet mit `#pragma acc`. Die komplette Syntax sieht somit wie folgt aus:

```
#pragma acc Direktive [Klausel [[,] Klausel] ...]
{Codezeile/-block}
```

Im Folgenden werden die einzelnen OpenACC-Direktiven (u.a. `parallel`, `kernels`, `serial`) und -Klauseln kurz vorgestellt. Die Ausführungen beziehen sich auf die OpenACC-Spezifikation 2.6 vom November 2017. [Ope18a]

4.3.1 Direktive

In diesem Abschnitt werden die wichtigsten OpenACC-Direktive kurz vorgestellt. Dazu gehören: `parallel`, `kernels`, `serial`, `loop` und `data`.

4.3.1.1 Parallel

Mit Hilfe der Direktive `parallel` wird eine parallele Ausführung auf dem aktuellen Beschleuniger (z.B. GPU oder Multicore-CPU) gestartet. In C und C++ sieht die Syntax wie folgt aus:

```
#pragma acc parallel [Klausel [[,] Klausel] ...]
{Codezeile/-block}
```

Trifft das Programm auf ein OpenACC-Pragma mit der Direktive `parallel` so werden mehrere Gruppen (*gangs*) von Arbeiten (*workers*) erzeugt, die die anschließende Codezeile oder den anschließenden Codeblock parallel auf dem Beschleuniger ausführen. Dabei ist die Anzahl der Gruppen, die Anzahl der Arbeiter pro Gruppe und die Vektorlänge (*vector length*) pro Arbeiter für den kompletten parallelen Bereich konstant und kann vorher definiert werden (siehe Abschnitt 4.3.3.1). Ist die `async`-Klausel (siehe Abschnitt 4.3.3.1) nicht vorhanden, gibt es eine implizierte Barriere am Ende der parallelen Region und die Ausführung des lokalen Threads wird erst dann fortgesetzt wenn alle Gruppen von Arbeitern das Ende der parallelen Region erreicht haben.

Die Direktive `parallel` kann mit weiteren Klauseln kombiniert werden. Dazu gehören:

- **Ausführungsklauseln**: async, wait, num_gangs, num_workers, vector_length, device_type, if, reduction
- **Datenklauseln**: copy, copyin, copyout, create, no_create, present, deviceptr, attach, private, firstprivate

Einige Ausführungsklauseln werden im Abschnitt 4.3.3.1 kurz vorstellt. Die Datenklauseln im Abschnitt 4.3.3.2. Für weitere Details wird auf die OpenACC-Spezifikation [Ope18a] verwiesen.

4.3.1.2 Kernels

Diese Direktive definiert eine Programmregion, die in eine Sequenz von Kerneln kompiliert und anschließend auf dem aktuellen Beschleuniger ausgeführt wird. Die Syntax in C und C++ sieht wie folgt aus:

```
#pragma acc kernels [Klausel [[,] Klausel] ...]
{Codezeile/-block}
```

Entdeckt der Compiler im Code eine `kernels`-Region so teilt er den Code innerhalb der Region in eine Sequenz von Kerneln für den Beschleuniger. Normalerweise wird jede geschachtelte Schleife ein einzelner Kernel. Erreicht das Programm während der Ausführung eine `kernels`-Region, so werden die vom Compiler erzeugten Kernel auf dem Beschleuniger ausgeführt. Die Anzahl der Gruppen (*gangs*), die Anzahl der Arbeiter pro Gruppe (*workers*) und die Vektorlänge (*vector length*) pro Arbeiter kann bei jedem Kernel unterschiedlich sein. Ist die `async`-Klausel (siehe Abschnitt 4.3.3.1) nicht vorhanden, gibt es eine implizierte Barriere am Ende der parallelen Region und die Ausführung des lokalen Threads wird erst dann fortgesetzt wenn alle Gruppen von Arbeitern das Ende der parallelen Region erreicht haben.

Die Direktive `kernels` kann mit weiteren Klauseln kombiniert werden. Dazu gehören:

- **Ausführungsklauseln**: async, wait, num_gangs, num_workers, vector_length, device_type, if; reduction ist hier nicht möglich!!
- **Datenklauseln**: copy, copyin, copyout, create, no_create, present, deviceptr, attach

Einige Ausführungsklauseln werden im Abschnitt 4.3.3.1 kurz vorstellt. Die Datenklauseln im Abschnitt 4.3.3.2. Für weitere Details wird auf die OpenACC-Spezifikation [Ope18a] verwiesen.

4.3.1.3 Serial

Durch die `serial`-Direktive wird auf eine/n Codeblock/-zeile hingewiesen, die auf dem Beschleuniger seriell ausgeführt werden soll. Die Syntax in C und C++ sieht wie folgt aus:

```
#pragma acc serial [Klausel [[,] Klausel] ...]
{Codezeile/-block}
```

Erreicht ein Programm eine `serial`-Region, so wird auf dem Beschleuniger eine Gruppe mit einem Arbeiter und einer Vektorlänge von eins erzeugt und der Code innerhalb der Region sequentiell ausgeführt. Somit ist eine `serial`-Region nichts anderes als eine `parallel`-Direktive mit den Klauseln `num_gangs(1) num_workers(1) vector_length(1)`. Ist die `async`-Klausel (siehe Abschnitt 4.3.3.1) nicht vorhanden, gibt es eine implizierte Barriere am Ende der parallelen Region und die Ausführung des lokalen Threads wird erst dann fortgesetzt wenn alle Gruppen von Arbeitern das Ende der parallelen Region erreicht haben.

Die Direktive `serial` kann mit weiteren Klauseln kombiniert werden. Dazu gehören:

- **Ausführungsklauseln**: async, wait, device_type, if, reduction, private, firstprivate

- **Datenklauseln**: copy, copyin, copyout, create, no_create, present, deviceptr, attach

Einige Ausführungsklauseln werden im Abschnitt 4.3.3.1 kurz vorstellt. Die Datenklauseln im Abschnitt 4.3.3.2. Für weitere Details wird auf die OpenACC-Spezifikation [Ope18a] verwiesen.

4.3.1.4 Loop

Eine `loop`-Direktive betrifft eine direkt folgende Schleife bzw. geschachtelte Schleifen. Sie beschreibt, welche Art von Parallelität verwendet werden soll, um die Schleife auszuführen. Außerdem können bei Bedarf private Variablen und Arrays deklariert sowie eine Reduktionsoperation spezifiziert werden. Das Ganze kann in C und C++ wie folgt verwendet werden:

```
#pragma acc loop [Klausel [[,] Klausel] ...]
for-Schleife
```

Die Direktive `loop` kann mit weiteren Klauseln kombiniert werden. Dazu gehören:

- **Ausführungsklauseln**: seq, reduction, private, device_type
- **Schleifenklauseln**: collapse, auto, independent, tile, gang, worker, vector

Einige Ausführungsklauseln werden im Abschnitt 4.3.3.1 kurz vorstellt. Die Schleifenklauseln im Abschnitt 4.3.3.3. Für weitere Details wird auf die OpenACC-Spezifikation [Ope18a] verwiesen.

4.3.1.5 Data

Mit Hilfe der `data`-Direktive können Variablen, Arrays und Subarrays im Beschleunigerspeicher allokiert werden, die für die Dauer der Region zur Verfügung stehen. Dazu werden Daten zu Beginn der Region aus dem Hostspeicher in den Beschleunigerspeicher kopiert und am Ende zurück. Die Syntax in C und C++ ist wie folgt:

```
#pragma acc data [Klausel [[,] Klausel] ...]
{Codezeile/-block}
```

Die Direktive `data` kann mit weiteren Klauseln kombiniert werden. Dazu gehören:

- **Ausführungsklauseln**: if
- **Datenklauseln**: copy, copyin, copyout, create, no_create, present, deviceptr, attach

Einige Ausführungsklauseln werden im Abschnitt 4.3.3.1 kurz vorstellt. Die Datenklauseln im Abschnitt 4.3.3.2. Für weitere Details wird auf die OpenACC-Spezifikation [Ope18a] verwiesen.

4.3.2 Direktive kombinieren

Einige der in Abschnnitt 4.3.1 vorgestellt OpenACC-Direktive können auch kombiniert werden. Diese werden im Folgenden kurz vorgestellt.

4.3.2.1 Parallel Loop

Die Kombination der `parallel`- und der `loop`-Direktive ist eine Abkürzung für eine loop-Direktive innerhalb einer `parallel`-Region. Das ganze sieht in C und C++ wie folgt aus:

```
#pragma acc parallel loop [Klausel [[,] Klausel] ...]
for-Schleife
```

Alle Klauseln, die in einer `parallel`- oder einer `loop`-Direktive erlaubt sind, können auch hier verwendet werden. Die `private`- und die `reduction`-Klausel, die in beiden Direktiven verwendet werden kann, besitzt hier die selbe Semantik wie bei einer einfachen `loop`-Direktive.

4.3.2.2 Kernels Loop

Die Kombination der `kernels`- und der `loop`-Direktive ist eine Abkürzung für eine loop-Direktive innerhalb einer `kernels`-Region. Es folgt die Verwendung in C und C++:

```
#pragma acc kernels loop [Klausel [[,] Klausel] ...]
for-Schleife
```

Alle Klauseln, die in einer `kernels`- oder einer `loop`-Direktive erlaubt sind, können auch hier verwendet werden.

4.3.2.3 Serial Loop

Die Kombination der `serial`- und der `loop`-Direktive ist eine Abkürzung für eine loop-Direktive innerhalb einer `parallel`-Region. In C und C++ wird das Ganze wie folgt verwendet:

```
#pragma acc serial loop [Klausel [[,] Klausel] ...]
for-Schleife
```

Alle Klauseln, die in einer `serial`- oder einer `loop`-Direktive erlaubt sind, können auch hier verwendet werden. Die `private`-Klausel, die in beiden Direktiven verwendet werden kann, besitzt ausschließlich die Semantik wie bei einer einfachen `loop`-Direktive.

4.3.3 Klauseln

In diesem Abschnitt werden einige Details zu den zuvor in Abschnitt 4.3.1 erwähnten Klauseln vorgestellt.

4.3.3.1 Ausführungsklauseln

num_gangs

Die `num_gangs`-Klausel ist in der `parallel` und `kernels`-Direktive erlaubt. Der angegeben Integer bestimmt wie viele Gruppen von Arbeitern (Threadblöcke/Warps, in CUDA: gridDim) verwendet werden sollen. Wird die Anzahl an Thread-Blöcken nicht spezifiziert, so wird ein Defaultwert verwendet.

```
#pragma acc [parallel/kernels] num_gangs(Integer)
```

Größere Integer führen im Allgemeinen zu einer geringeren Ausführungszeit. Die Klausel hat die höchste Auswirkung auf die Ausführungszeit der Datenverarbeitung.

num_workers

Die `num_workers`-Klausel ist in der `parallel` und `kernels`-Direktive erlaubt. Der angegeben Integer bestimmt wie viele Arbeiter pro Gruppe (Threads pro Threadblock, in CUDA: blockDim) verwendet werden sollen. Wird die Anzahl an Arbeiter pro Threadblöcken nicht spezifiziert, so wird ein Defaultwert verwendet.

```
#pragma acc [parallel/kernels] num_worker(Integer)
```

Bei der Kompilierung mit dem gcc 7.1.0 und früher sind Werte zwischen 2 und 32 möglich, wobei ein größerer Wert im Normalfall zu einer geringeren Ausführungszeit führt.

vector_length

Die `vector_length`-Klausel ist in der `parallel` und `kernels`-Direktive erlaubt. Der angegeben Integer bestimmt die Vektorlänge (Prozessor), die für Vektor- oder SIMD-Operationen verwendet werden sollen. Die Vektorlänge wird sowohl bei Schleifen, die mit der `vector_length clause` versehen wurden verwendet, als auch bei automatischen Vektorisierungen durch den Compiler. Wird die Anzahl an Vektorlänge nicht spezifiziert, so wird ein Defaultwert verwendet.

```
#pragma acc [parallel/kernels] vector_length(Integer)
```

Bei der Kompilierung mit dem gcc 7.1.0 und früher ist die Vektorlänge immer genau 32. Der Wert kann also nicht vom Programmierer angepasst werden.

reduction

Die `reduction`-Klausel ist in der `parallel`, `serial` und `loop`-Direktive erlaubt. Sie spezifiziert einen Reduktionsoperation sowie einen oder mehrere skalare Variablen. Für jede

parallele Gruppe (gang) wird von jeder Variable eine private Kopie erzeugt und für den angegeben Reduktionsoperator initialisiert. Am Ende der Region werden die einzelnen Werte unter Verwendung des Reduktionsoperators kombiniert und das Ergebnis in der originalen Variable gespeichert. Das Resultat ist erst am Ende der Region verfügbar.

```
#pragma acc [parallel/serials/loop] reduction(Operator : List)
```

Die Tabelle 4.2 zeigt alle gültigen Reduktionsoperatoren und die dazugehörigen Initialisierungswerte.

Operation	**Initialisierungswert**
+	0
*	1
max	kleinster Wert
min	größter Wert
&	~0
|	0
%	0
&&	1
||	0

Tabelle 4.2: Reduktionsoperationen und ihre Initialisierungswerte.

Die unterstützten Datentypen sind:

- in C: char, int, float, double, _Complex
- in C++: char, wchar_t, int, float, double

async

Durch die Verwendung der `async`-Klausel wird die nachfolgende Codezeile oder der nachfolgende Codeblock auf der angegeben asynchronen Queue (Expression) ausgeführt. Ohne Argument wird das Ganze auf der standardmäßigen asynchronen Queue ausgeführt.

```
#pragma acc [parallel/kernels/serial] async([Expression])
```

wait

Die Compute-Region wartet so lange bis alle Ausführungen der angegeben asynchronen Queue(s) komplett ausgeführt wurden. Ohne Argument wird auf die Fertigstellung aller asynchronen Queues gewartet.

```
#pragma acc [parallel/kernels/serial] wait([Expression-List])
```

if

Die `if`-Klausel ist optional. Gibt der Programmierer keine `if`-Klausel an, dann generiert der Compiler Code für den aktuellen Beschleuniger.

```
#pragma acc [parallel/kernels/serial] if(condition)
```

Wird eine `if`-Klausel angegeben, so erstellt der Compiler zwei Kopien der Direktivenausführung, eine Kopie für die Ausführung auf dem Beschleuniger und eine für die Ausführung durch einen lokalen Thread. Wenn die if-Bedingung in C und C++ zum Ergebnis 0 führt, so wird die Direktivenausführung durch den lokalen Thread ausgeführt, andernfalls (ungleich 0) durch den Beschleuniger.

```
#pragma acc data if(condition)
```

Wird die `if`-Klausel in der `data`-Direktive verwendet und die Bedingung ist 0, dann wird kein Speicher auf dem Beschleuniger allokiert und auch keine Daten vom Hostsystem in den Beschleunigerspeicher kopiert.

private

Die `private`-Klausel ist in der `parallel`, `serial` und `loop`-Direktive erlaubt. Sie kann verwendet werden, um für jede Gruppe von Threads (gangs) eine private Kopie anzulegen.

```
#pragma acc [parallel/serial/loop] private(list)
```

firstprivate

Die `firstprivate`-Klausel ist in der `parallel` und `serial`-Direktive erlaubt. Sie kann verwendet werden, um für jede Gruppe von Threads (gangs) eine private Kopie anzulegen und diese mit dem Wert der Variablen zu initialisieren der die (parallele) Region aufruft.

```
#pragma acc [parallel/serial] firstprivate(list)
```

4.3.3.2 Datenklauseln

copy

Die `copy`-Klausel ist in der `parallel`, `kernels`, `serial` und `data`-Direktive erlaubt. In C und C++ gilt die folgende Syntax:

```
#pragma acc [parallel/kernels/serial/data] copy(list)
```

Beim Betreten der Region: Sind die Daten in `list` bereits auf dem Beschleuniger verfügbar, so wird der Referenzcounter um eins erhöht und diese Kopie verwendet. Andernfalls wird Speicher auf dem Beschleuniger allokiert, der Wert vom Hostsystem zum Beschleuniger kopiert und der Referenzcounter auf eins gesetzt.

Beim Verlassen der Region: Der Referenzcounter wird um eins reduziert. Ist der Referenzcounter 0, dann werden die Daten vom Beschleunigerspeicher in den Speicher des Hostsystems kopiert und der belegte Speicher auf dem Beschleuniger freigegeben.

copyin

Die `copyin`-Klausel ist in der `parallel`, `kernels`, `serial` und `data`-Direktive erlaubt. In C und C++ gilt die folgende Syntax:

```
#pragma acc [parallel/kernels/serial/data] copyin(list)
```

Beim Betreten der Region: Sind die Daten in `list` bereits auf dem Beschleuniger verfügbar, so wird der Referenzcounter um eins erhöht und diese Kopie verwendet. Andernfalls wird Speicher auf dem Beschleuniger allokiert, der Wert vom Hostsystem zum Beschleuniger kopiert und der Referenzcounter auf eins gesetzt.

Beim Verlassen der Region: Der Referenzcounter wird um eins reduziert. Ist der Referenzcounter 0, so wird der belegte Speicher auf dem Beschleuniger freigeben.

copyout

Die `copyout`-Klausel ist in der `parallel`, `kernels`, `serial` und `data`-Direktive erlaubt. In C und C++ gilt die folgende Syntax:

```
#pragma acc [parallel/kernels/serial/data] copyout(list)
```

Beim Betreten der Region: Sind die Daten in `list` bereits auf dem Beschleuniger verfügbar, so wird der Referenzcounter um eins erhöht und diese Kopie verwendet. Andernfalls wird Speicher auf dem Beschleuniger allokiert und der Referenzcounter auf eins gesetzt.

Beim Verlassen der Region: Der Referenzcounter wird um eins reduziert. Ist der Referenzcounter 0, dann werden die Daten vom Beschleunigerspeicher in den Speicher des Hostsystems kopiert und der belegte Speicher auf dem Beschleuniger freigeben.

create

Die `create`-Klausel ist in der `parallel`, `kernels`, `serial` und `data`-Direktive erlaubt. In C und C++ gilt die folgende Syntax:

```
#pragma acc [parallel/kernels/serial/data] create(list)
```

Beim Betreten der Region: Sind die Daten in `list` bereits auf dem Beschleuniger verfügbar, so wird der Referenzcounter um eins erhöht und diese Kopie verwendet. Andernfalls wird Speicher auf dem Beschleuniger allokiert und der Referenzcounter auf eins gesetzt.

Beim Verlassen der Region: Der Referenzcounter wird um eins reduziert. Ist der Referenzcounter 0, so wird der belegte Speicher auf dem Beschleuniger freigegeben.

4.3.3.3 Schleifenklauseln

gang

Bei einer `loop`-Direktive innerhalb einer `parallel`-Direktive: Teilt die Iterationen der Schleife oder Schleifen auf die zur Verfügung stehenden Threadblöcke (*gangs*) in der parallelen Region auf. Die Anzahl an Threadblöcken kann individuell eingestellt werden (siehe `num_gangs` in Abschnitt 4.3.3.1). Verwendung:

```
#pragma acc parallel
{
     #pragma acc loop gang
     for-Schleife
}
```

Bei einer `loop`-Direktive innerhalb einer `kernels`-Direktive: Führt die Iterationen der Schleife oder Schleifen parallel mit maximal `num_gangs` Threadblöcken aus.

```
#pragma acc kernels
{
     #pragma acc loop gang[(num_gangs)]
     for-Schleife
}
```

Die `gang`-Klausel kann auch mit der `worker`- und `vector`-Klausel kombiniert werden.

worker

Bei einer `loop`-Direktive innerhalb einer `parallel`-Direktive: Teilt die Iterationen der Schleife oder Schleifen auf die zur Verfügung stehenden Arbeit (*worker*) pro Threadblock (*gangs*) in der parallelen Region auf. Die Anzahl an Arbeiter pro Threadblock kann individuell eingestellt werden (siehe `num_workers` in Abschnitt 4.3.3.1). Verwendung:

```
#pragma acc parallel
{
     #pragma acc loop worker
     for-Schleife
}
```

Bei einer `loop`-Direktive innerhalb einer `kernels`-Direktive: Führt die Iterationen der Schleife oder Schleifen parallel mit maximal `num_works` Arbeitern pro Threadblock aus. Verwendung:

```
#pragma acc kernels
{
     #pragma acc loop worker[(num_workers)]
     for-Schleife
}
```

Die `worker`-Klausel kann auch mit der `gang`- und `vector`-Klausel kombiniert werden.

vector

Bei einer `loop`-Direktive innerhalb einer `parallel`-Direktive: Berechnet die Iteration der Schleife oder Schleifen im SIMD- oder Vektormodus. Die Vektorlänge kann individuell eingestellt werden (siehe `vector_length` in Abschnitt 4.3.3.1). Verwendung:

```
#pragma acc parallel
{
     #pragma acc loop vector
     for-Schleife
}
```

Bei einer `loop`-Direktive innerhalb einer `kernels`-Direktive: Berechnet die Iterationen der Schleife oder Schleifen im SIMD- oder Vektormodus mit einer maximalen Vektorlänge von `vector_length`. Verwendung:

```
#pragma acc kernels
{
     #pragma acc loop vector[(vector_length)]
     for-Schleife
}
```

Die `vector`-Klausel kann auch mit der `gang`- und `worker`-Klausel kombiniert werden.

collapse

Wendet die zugehörige Direktive auf die folgenden `n` eng verschachtelten Schleifen an. Das folgende Beispiel zeigt die Verwendung:

```
#pragma acc parallel for collapse(n)
for-Schleife
{
     for-Schleife
     {
          ...
     }
}
```

seq

Berechnet die Schleife oder Schleifen sequentiell. Beispiel:

```
#pragma acc parallel for seq
for-Schleife
```

auto

Gibt dem Compiler die Anweisung, die Schleife oder Schleifen zu analysieren, um festzustellen, ob sie sicher parallel ausgeführt werden kann/können, und wenn ja eine `gang`-, `worker`- und `vector`-Parallelität anzuwenden. Beispiel:

```
#pragma acc parallel for auto
for-Schleife
```

independent

Gibt an, dass die Schleifeniterationen datenunabhängig sind und parallel ausgeführt werden können. Dabei werden Erkenntnisse aus übergeordneten Compiler-Abhängigkeitsanalysen überschrieben. Beispiel:

```
#pragma acc parallel for independet
for-Schleife
```

Kapitel 5

Parallele Nutzung von mehreren GPUs

In diesem Kapitel werden zwei Ansätze zur parallelen Nutzung von mehreren GPUs in einem Computer unter Verwendung von OpenACC präsentiert. Abschnitt 5.1 zeigt zunächst das allgemeine Vorgehen der beiden Varianten. Anschließend werden einige praktische Implementierungen gezeigt, die auf dem Algorithmus zur Approximation von Pi (siehe Abschnitt 3.1) aufbauen. Den Anfang macht eine OpenACC-Implementierung (Abschnitt 5.2) auf Basis einer asynchronen GPU-Nutzung. Außerdem werden pthreads und OpenACC (Abschnitt 5.3) sowie OpenMP und OpenACC (Abschnitt 5.4) kombiniert, um zwei Beispiele zu zeigen, bei denen mit Hilfe von CPU-Threads die vorhandenen GPUs im Computer parallel genutzt werden. Abschließend wird die Ausführungszeit der gezeigten Implementierungen verglichen und ein kurzes Fazit gezogen (Abschnitt 5.5).

5.1 Allgemeines

Enthält ein Computer mehrere GPUs, so können diese auch parallel genutzt werden. Im Folgenden werden zwei unterschiedliche Varianten vorgestellt.

Um die Nutzung der vorhandenen NVIDIA-GPUs zu überprüfen, kann in einem zweiten Terminal das folgende Kommando ausgeführt werden:

```
nvidia-smi -l 1
```

5.1.1 Variante 1

Die erste Variante basiert auf der asynchronen Verwendung einer GPU mit Hilfe von OpenACC. Wie wir bereits in Abschnitt 4.3.3.1 gesehen haben, kann in einem OpenACC-Pragma die Ausführungsklausel `async([Expression])` verwendet werden, um einen darauf

folgenden Codeblock asynchron auf einer GPU auszuführen. Natürlich muss später mit Hilfe der `wait`-Klausel auf die Fertigstellung gewartet werden, wenn das Ergebnis für weitere Berechnungen benötigt wird. Genau dieser Mechanismus wird im Folgenden verwendet, um alle GPUs nacheinander mit Arbeit zu versorgen und anschließend auf das Ende aller Berechnungen zu warten. Der Code 5.1 zeigt den Pseudocode dieser Variante.

```
1    num_gpus = GPU-Anzahl im Computer
2
3    for i=0 to num_gpus
4        gpu i auswaehlen
5        asynchrone Berechnung auf gpu i starten, async()
6
7    for i=0 to num_gpus
8        gpu i auswaehlen
9        auf das Ende der Berechnungen auf gpu i warten, wait()
```

Code 5.1: Pseudocode der Variante 1 zur parallelen Nutzung mehrerer GPUs in einem Computer.

Ist der Berechnungsaufwand auf den GPUs lang genug, so arbeiten alle vorhanden GPUs für einen gewissen Zeitraum parallel. Außerdem kann nach dem Start der GPU-Berechnungen die CPU des Hostsystems für andere Berechnungen genutzt werden, bevor auf das Ende der GPU-Berechnungen gewartet wird.

Eine OpenACC-Implementierung dieser Variante für den Algorithmus zur Approximation der Kreiszahl Pi (sequentielle Implementierung siehe Abschnitt 3.1.1) wird im Abschnitt 5.2 detailliert vorgestellt.

5.1.2 Variante 2

Die zweite Variante nutzt die Generierung von CPU-Threads, um die einzelnen GPUs im Computer parallel mit Aufgaben zu versorgen. Die allgemeine Herangehensweise besteht somit aus zwei Schritten:

1. Erzeugung von Threads $<=$ GPU-Anzahl

   $\rightarrow$ zum Beispiel mit der phtreads- oder der OpenMP-Bibliothek

2. Jeder Thread i versorgt eine GPU i mit einer Aufgabe (mit $i <=$ GPU-Anzahl)

Der große Nachteil dieser Variante ist, dass neben OpenACC eine Bibliothek zur Erzeugung von CPU-Threads benötigt wird.

Eine praktische Implementierung dieses Ansatzes wird mit der Kombination von pthreads und OpenACC (Abschnitt 5.2) sowie von OpenMP und OpenACC (Abschnitt 5.3) gezeigt. Beide Beispiele implementieren den Algorithmus zur Approximation der Kreiszahl Pi (sequentielle Implementierung siehe Abschnitt 3.1.1).

5.2 OpenACC

In diesem Abschnitt wird die Variante 1 (siehe Abschnitt 5.1.1) genutzt, um den Algorithmus zur Approximation der Kreiszahl Pi (sequentielle Implementierung siehe Abschnitt 3.1.1) auf die vorhandenen GPUs in einem Computer aufzuteilen und so die Ausführungszeit für große Ns zu senken.

Es folgt zunächst der C-Code (Code 5.2), der mit einigen Kommentaren versehen wurde. Anschließend wird auf die wichtigsten Codezeilen kurz eingegangen.

```
1  #include <openacc.h>  // fuer OpenACC-Funktionen
2
3  #define N 1000000000000
4
5  #define ng 2048 // num_gangs
6  #define nw 32   // num_workers
7  #define vl 32   // vector_length
8
9  /* Berechnung von PI mit OpenACC-Unterstuetzung */
10 double asyncUseAllGPUs() {
11   double pi_tmp = 0.0f;
12
13   int num_gpus = acc_get_num_devices(acc_device_nvidia);
14   for(int gpu=0; gpu<num_gpus; gpu++) {
15     acc_set_device_num(gpu, acc_device_nvidia);
16
17     #pragma acc enter data copyin(pi_tmp)
18     #pragma acc parallel num_gangs(ng) num_workers(nw)
          vector_length(vl) async(gpu)
19     #pragma acc loop gang worker vector reduction(+:pi_tmp)
20     for (long long i=(long long)gpu; i<N; i+=num_gpus) {
21       double t = (double)((i+0.5)/N);
22       pi_tmp += 4.0/(1.0+t*t);
23     }
24   }
25
26   /*
27    * 1) auf die Fertigstellung der Teilberechnungen warten
```

```
28     * 2) Teilergebnisse summieren
29    */
30    double pi = 0.0f;
31    for(int gpu=0; gpu<num_gpus; gpu++) {
32      acc_set_device_num(gpu, acc_device_nvidia);
33
34      #pragma acc wait
35      #pragma acc exit data copyout(pi_tmp)
36      pi += pi_tmp;
37    }
38
39    return pi/N;
40  }
41
42  int main(void) {
43    double pi = asyncUseAllGPUs();
44    return 0;
45  }
```

Code 5.2: Pi-Approximation mit OpenACC auf Basis von asynchronen Pragmas.

Der Code 5.2 besitzt eine starke Ähnlichkeit mit dem Pseudocode der Variante 1 (siehe Code 5.1). Zunächst wird mit der OpenACC-Funktion `acc_get_num_devices(acc_device_nvidia)` (Codezeile: 13) die Anzahl an NVIDIA-GPUs im System bestimmt.

Danach startet die Versorgung der einzelnen GPUs des Computers mit einer Berechnung (Codezeile: 14 bis 24), genau wie beim Pseudocode der Variante 1 (siehe erste for-Schleife in Code 5.1). Der erste Schritt ist die Auswahl der NVIDIA-GPU *gpu* mit der OpenACC-Funktion `acc_set_device_num(gpu, acc_device_nvidia)` (Codezeile: 15). Anschließend berechnet jede GPU nur einen bestimmten Teil von Pi, da die entsprechende for-Schleife (Codezeile: 20 bis 23) bei der jeweiligen GPU-Nummer (`gpu`) startet und Schritte der Größe `num_gpus` macht. Die OpenACC-Pragmas vor der Schleife haben die folgende Bedeutung:

- `#pragma acc enter data copyin(pi_tmp)`

  → Die Variable *pi_tmp* vom Hostsystem zur GPU *gpu* kopieren.

- `#pragma acc parallel num_gangs(ng) num_worker(nw) vector_length(vl) async(gpu)`

  → Eine parallele-Region starten und dabei die Threadblock-Größe mit *ng*, die Threadanzahl pro Threadblock mit *nw* und die Vektorenlänge pro Thread mit *vl* initialisieren. Außerdem wird mit *async*(*gpu*) eine asynchrone Ausführung gefordert.

- `#pragma acc loop gang vector worker reduction(+:pi_tmp)`

  → Die darauf folgende for-Schleife wird unter Verwendung der zuvor spezifizierten Threadblock-Größe (*gang*), Threadanzahl pro Threadblock (*worker*) und Vektorlänge pro Thread (*vector*) auf der GPU ausgeführt. Zusätzlich wird eine Reduktion mit der Operation + durchgeführt, wobei das finale Ergebnis der Reduktion in der Variable *pi_tmp* gespeichert wird.

Der letzte Schritt der Implementierung ist das Warten auf das Ende der Berechnungen von jeder GPU. Dies geschieht in der Codezeile 30 bis 37 der OpenACC-Implementierung (Code 5.2). Zunächst wird erneut die NVIDIA-GPU *gpu* ausgewählt (Codezeile: 32). Danach muss nur noch auf die Fertigstellung aller Berechnungen von GPU *gpu* gewartet und das Teilergebnis zurück in den Speicher des Hostsystems kopiert werden. Dies geschieht mit zwei OpenACC-Pragmas (Codezeile: 34 bis 35):

- `#pragma acc wait`

  → Auf die Fertigstellung aller asynchronen Berechnungen auf der zuvor ausgewählten GPU *gpu* warten.

- `#pragma acc exit data copyout(pi_tmp)`

  → Das Ergebnis der GPU-Berechnung (*pi_tmp*) zurück in den Speicher des Hostsystems kopieren.

Am Ende jeder Schleifeiteration wird `pi_tmp` (ein Teilergebnis) zu `pi` (dem späteren Endergebnis) addiert (Codezeile: 36).

5.3 Pthreads mit OpenACC

In diesem Abschnitt wird die parallele Nutzung von mehreren GPUs in einem Computer mit Hilfe der Approximation von Pi gezeigt. Dazu wird die Pthread- und OpenACC-Bibliothek genutzt. Das ganze basiert somit auf der Variante 2 (siehe Abschnitt 5.1.2).

Zunächst folgt der C-Code mit einigen Kommentaren, wobei danach auf die wichtigsten Codezeilen eingegangen wird.

```
1 #include <stdlib.h>   // zur Speicherallokation
2 #include <pthread.h>  // zur CPU-Thread Erstellung
3 #include <openacc.h>  // fuer OpenACC-Funktionen
4
5 #define N 1000000000000
6
```

```
7  #define ng 2048 // num_gangs
8  #define nw 32   // num_workers
9  #define vl 32   // vector_length
10
11 /* Pthread-Daten */
12 typedef struct Data {
13     int gpu_id;
14     int num_gpus;
15 } data_t;
16
17 /* Berechnung von PI mit OpenACC-Unterstuetzung */
18 double calcPI(const int gpu_id, const int num_gpus) {
19   double pi = 0.0f;
20
21   #pragma acc parallel num_gangs(ng) num_workers(nw)
       vector_length(vl)
22   #pragma acc loop gang vector worker  reduction(+:pi)
23   for (long long i=gpu_id; i<N; i+=num_gpus) {
24     double t = (double)((i+0.5)/N);
25     pi += 4.0/(1.0+t*t);
26   }
27
28   return pi;
29 }
30
31 /* Pthread-Startfunktion */
32 void *calculatePI(void *data) {
33   // Pthread-Daten auspacken
34   data_t *my_data = (data_t*) data;
35   int gpu_id = (*my_data).gpu_id;     // GPU_id
36   int num_gpus = (*my_data).num_gpus; // Anzahl an vorhanden GPUs
37   free(data);
38
39   // waehle GPU
40   acc_set_device_num(gpu_id, acc_device_nvidia);
41
42   // berechne Teilergebnis von PI
43   double* sub_pi = (double*) malloc(sizeof(double));
44   *sub_pi = calcPI(gpu_id, num_gpus);
45
46   return sub_pi;
47 }
48
49 /* Pthread-Generierung */
```

```
50 double pthreadUseAllGPUs(void) {
51   // Anzahl an NVIDIA-GPUS
52   int num_gpus = acc_get_num_devices(acc_device_nvidia);
53
54   // CPU-Threads erstellen = Anzahl an NVIDIA-GPUs
55   pthread_t* threads = (pthread_t*) malloc(num_gpus *
       sizeof(pthread_t));
56   int rc;  // return code
57
58   data_t* data;
59   for(int g=0; g<num_gpus; g++) {
60     // setze gpu_id und Anzahl an GPUs
61     data = malloc(sizeof(data_t));
62     (*data).gpu_id = g;
63     (*data).num_gpus = num_gpus;
64
65     // erstellen Pthreads mit den notwendigen Daten
66     rc = pthread_create(&threads[g], NULL, calculatePI, (void*)
         data);
67     if (rc) {
68       fprintf(stderr, "Error creating thread. Return code from
           pthread_create() is %d\n", rc);
69       exit(-1);
70     }
71   }
72
73   double pi = 0.0f;
74   double* tmp;
75   // warten bis alle Pthreads fertig sind, summiere Teilergebnisse
76   for(int g=0; g<num_gpus; g++) {
77     rc = pthread_join(threads[g], (void*) &tmp);
78     if(rc) {
79       fprintf(stderr, "Error joining thread. Return code from
           pthread_join() is %d\n", rc);
80       exit(-1);
81     }
82
83     pi += *tmp;
84     free(tmp);
85   }
86
87   free(threads);
88   return pi/N;
89 }
```

```
90
91 int main(void) {
92   double pi = pthreadUseAllGPUs();
93   return 0;
94 }
```

Code 5.3: Pi-Approximation mit phreads und OpenACC.

Die Pthread-Implementierung (Code 5.3) besteht aus drei Funktionen, die im Folgenden kurz vorgestellt werden.

Die Funktion `double pthreadUseAllGPUs(...)` (Codezeile: 49 bis 89) ist das Herz der Implementierung, denn hier werden die einzelnen pthreads generiert. Grundlage für die Erstellung ist die Anzahl an NVIDIA-GPUs im System, die mit der OpenACC-Funktion `acc_get_num_devices(acc_device_nvidia)` (Codezeile: 52) abgefragt werden können. Danach werden die einzelnen pthreads erzeugt (Codezeile: 54 bis 71), wobei jeder von ihnen die Startfunktion `void *calculatePI(...)` aufruft und die Parameter

- `gpu_id`: GPU, die dieser phtread mit Arbeit versorgen soll
- `num_gpus`: Anzahl der GPUs im System

in Form des structs `data_t` erhält (vgl. `pthread_create(...)` in Codezeile: 66). Abschließend wird auf die Fertigstellung der einzelnen Berechnungen gewartet (vgl. `pthread_join(...)` in Codezeile: 77) und die Teilergebnisse summiert (Codezeile: 83).

Nach der Erzeugung eines pthreads wird die Startfunktion `void *calculatePI(...)` ausgeführt (Codezeile: 31 bis 47). Zunächst werden die übergebenen Daten (`gpu_id` und `num_gpus`) ausgepackt (Codezeile: 33 bis 37). Danach wählt jeder pthread mit der OpenACC-Funktion `acc_set_device_num(gpu_id, acc_device_nvidia)` die ihm zugeteilte NVIDIA-GPU aus (Codezeile: 40) und startet die Berechnung von `sub_pi`, also einem Teil von Pi. Für die Berechnung wird die Funktion `double calcPI(...)` verwendet.

Die Funktion `double calcPI(...)` (Codezeile: 17 bis 29) ist bereits aus dem Kapitel 3 bekannt und wird zur Approximation von Pi auf einer GPU verwendet. Dazu wird auf die OpenACC-Bibliothek zurückgegriffen. Die einzige Besonderheit dieser Funktion ist die for-Schleife (Codezeile: 23), die mit der übergebenen `gpu_id` startet und deren Schritte eine Größe von `num_gpus` besitzen. Demzufolge bearbeitet jede verwendete GPU nur einen Teil der Approximation von Pi.

Zur Kompilierung des C-Programms in Code 5.3 (`openacc_pthreads.c`) kann das folgende Kommando verwendet werden:

```
gcc-7 openacc_pthreads.c -fopenacc -foffload=nvptx-none -foffload="-O3" -O3
                         -o openacc_pthreads -lpthread
```

Die Compiler-Flags `-fopenacc` und `-lpthread` aktivieren die OpenACC- und pthread-Unterstützung. Zusätzlich werden die Optionen `-foffload="-O3"` und `-O3` verwendet, um alle Compiler-Optimierungen einzuschalten. Mit `-foffload=nvptx-none` wird das Offloading aktiviert, wobei die Option nvptx-none (nvptx = *Nvidia Parallel Thread Execution*) notwendig ist, da auf der GPU kein eigenes Betriebssystem läuft.

5.4 OpenMP mit OpenACC

In diesem Abschnitt wird die parallele Nutzung von mehreren GPUs in einem Computer mit Hilfe der Approximation von Pi gezeigt. Dazu wird die OpenMP- und OpenACC-Bibliothek genutzt. Das ganze basiert somit auf der Variante 2 (siehe Abschnitt 5.1.2).

Die OpenMP-Implementierung (Code 5.4) ist eine doppelte Reduktion, wobei jeder OpenMP-Thread (um genau zu sein: die von ihm verwendete GPU) nur einen Teil der Pi-Approximation übernimmt. Die Implementierung besteht aus zwei Funktionen, die im Folgenden kurz vorgestellt werden.

Die Funktion `double ompUseAllGPUs(...)` (Codezeile: 26 bis 39) ist das Herz dieser Implementierung. Zunächst wird die Anzahl der vorhandenden NVIDIA-GPUs mit der OpenACC-Funktion `acc_get_num_devices(acc_device_nvidia)` abgefragt (Codezeile: 29). Anschließend wird eine parallele Reduktion mit Hilfe von OpenMP durchgeführt (Codezeile: 31 bis 36), wobei die Anzahl der OpenMP-Threads gleich der Anzahl an NVIDIA-GPUs entspricht (vgl. `num_threads(num_gpus)` in Codezeile: 33). Die Reduktion besteht aus der Teilberechnung von Pi, jeweils mit der Funktion `calcPI(...)`, und einer anschließenden Summierung der Teilergebnisse (Codezeile: 35).

Die Funktion `double calcPI(...)` (Codezeile: 9 bis 24) ist bereits aus dem Kapitel 3 bekannt und wird zur Approximation von Pi auf einer GPU verwendet. Dazu wird auf die OpenACC-Bibliothek zurückgegriffen, um eine der vorhandenden GPUs auszuwählen (Verwendung der OpenACC-Funktion `acc_set_device_num(gpu_id, acc_device_nvidia)`) und dort eine parallele Reduktion durchzuführen. Die einzige Besonderheit dieser Funktion ist die for-Schleife (Codezeile: 18), die mit der übergebenen `gpu_id` startet und die Schritte eine Größe von `num_gpus` besitzen. Demzufolge bearbeitet jede verwendete GPU nur einen Teil der Approximation von Pi.

Zur Kompilierung des C-Programms in Code 5.4 (`openacc_omp.c`) kann das folgende Kommando verwendet werden:

```
gcc-7 openacc_omp.c -fopenacc -foffload=nvptx-none -foffload="-O3" -O3 -o
                                    openacc_omp
```

Die Compiler-Flags unterscheiden sich nicht von einer normalen OpenACC-Kompilierung, da OpenMP direkt in OpenACC enthalten ist. Somit muss auch nicht zusätzlich die Headerdatei `omp.h` mit in den C-Code eingebunden werden.

```
1  #include <openacc.h>  // fuer OpenACC-Funktionen
2
3  #define N 1000000000000
4
5  #define ng 2048 // num_gangs
6  #define nw 32   // num_workers
7  #define vl 32   // vector_length
8
9  /* Berechnung von PI mit OpenACC-Unterstuetzung */
10 double calcPI(const int gpu_id, const int num_gpus) {
11   // waehle GPU
12   acc_set_device_num(gpu_id, acc_device_nvidia);
13
14   // berechne PI mit paralleler Reduktion (mit OpenACC)
15   double pi = 0.0f;
16   #pragma acc parallel num_gangs(ng) num_workers(nw)
       vector_length(vl)
17   #pragma acc loop gang vector worker  reduction(+:pi)
18   for (long long i=gpu_id; i<N; i+=num_gpus) {
19     double t = (double)((i+0.5)/N);
20     pi += 4.0/(1.0+t*t);
21   }
22
23   return pi;
24 }
25
26 /* Thread-Generierung mit OpenMP */
27 double ompUseAllGPUs(void) {
28   // Anzahl an NVIDIA-GPUs bestimmen
29   int num_gpus = acc_get_num_devices(acc_device_nvidia);
30
31   // parallele Reduktion mit Hilfe von OpenMP
32   double pi = 0.0f;
33   #pragma omp parallel for reduction(+:pi) num_threads(num_gpus)
34   for(int gpu_id=0; gpu_id<num_gpus; gpu_id++) {
35     pi += calcPI(gpu_id, num_gpus);
36   }
37
38   return pi/N;
39 }
```

```
40
41 int main(void) {
42   double pi = ompUseAllGPUs();
43   return 0;
44 }
```

Code 5.4: Pi-Approximation mit OpenMP und OpenACC.

5.5 Zusammenfassung

Im ersten Abschnitt dieses Kapitels (Abschnitt 5.1) wurden zwei Varianten zur parallelen Nutzung von mehreren GPUs innerhalb eines Computer vorgestellt. Anschließend wurde mit einer OpenACC-Implementierung (Abschnitt 5.2) ein Beispiel für die erste Variante präsentiert und mit der Kombination von pthreads und OpenACC (Abschnitt 5.3) sowie OpenMP und OpenACC (Abschnitt 5.4) zwei Beispiele, die auf der zweiten Variante aufbauen, vorgestellt. Abschließend wurden alle Implementierungen mit dem gcc 7.1.0 auf der Workstation ls4gpu2 (weitere Details zur Workstation siehe Anhang A) kompiliert und einige Messungen durchgeführt. Die Tabelle 5.1 fasst diese Ergebnisse zusammen.

N	**OpenACC** GPU: 1x NVIDIA® Quadro® P6000	**asynchron mit OpenACC** GPUs: 2x NVIDIA® Quadro® P6000	**pthreads mit OpenACC** GPUs: 2x NVIDIA® Quadro® P6000	**OpenMP mit OpenACC** GPUs: 2x NVIDIA® Quadro® P6000
10^3	2,56	4,60	4,70	7,63
10^4	2,58	4,18	5,11	7,66
10^5	2,52	4,41	4,97	7,85
10^6	2,68	4,12	4,46	7,85
10^7	2,54	4,20	4,87	7,86
10^8	9,85	6,04	6,81	11,42
10^9	82,13	43,00	43,23	84,43
10^{10}	807,23	409,14	412,01	812,00
10^{11}	–	4.256,21	4.441,33	8.260,88
10^{12}	–	44.691,87	44.690,94	81.789,01

Tabelle 5.1: Die Ausführungszeiten für die Approximation von Pi auf der Workstation ls4gpu2 für N zwischen 10^3 und 10^{12} in Millisekunden (ms). Zur Kompilierung der C-Implementierungen wurde der gcc 7.1.0 verwendet. Die Anzahl an *num_gangs* wurde auf 2048 gesetzt. *num_workers* und *vector_length* wurden mit 32 initialisiert. Alle Implementierungen inkl. Makefile und ein Skript zur Messung der einzelnen Ausführungszeiten befinden sich im Ordner: code/useAllGPUs/measurements/

Die Tabelle 5.1 macht deutlich, dass bei kleinen $N \leq 10^7$ der Berechnungsaufwand für zwei und sogar eine GPU zu gering ist. Demzufolge sind die Ausführungszeiten von jeder Implementierung in etwa konstant und werden von der Initialisierungszeit der GPU geprägt. Am schnellsten ist in diesem Bereich (N zwischen 10^3 und 10^7) die OpenACC-Implementierung für eine GPU. Ab $N \geq 10^8$ ist der Berechnungsaufwand so groß, dass sowohl die Ausführungszeiten bei der Nutzung von einer GPU als auch bei der parallelen Nutzung von zwei GPUs ansteigt. Die schnellsten Implementierungen sind ab jetzt die OpenACC-Implementierung für zwei GPUs unter Verwendung der asynchronen GPU-Steuerung und die Kombination aus pthreads und OpenACC.

Auf den ersten Blick überraschend ist die langsame Ausführungszeit der Kombination von OpenMP und OpenACC. Dies liegt am gcc 7.1.0 der OpenMP und OpenACC nicht gleichzeitig unterstützt. Daher werden die beiden GPUs nicht parallel aufgerufen und die Berechnung gestartet, sondern sequentiell. Der Ergebnis ist in etwa die gleiche Zeit wie bei der OpenACC-Implementierung für eine GPU, wobei bei kleinem $N \leq 10^7$ u.a. die zusätzlich Initialisierungszeit der zweiten GPU für die mehr als doppelte lange Ausführungszeit verantwortlich ist.

Kapitel 6

Profiling Tools

In diesem Kapitel werden mit nvprof und dem NVIDIA® Visual Profiler zwei Profiling Tools für OpenACC-Programme vorgestellt. Beide Tools sind Bestandteil des CUDA Toolkits von NVIDIA®. [Cud]

6.1 Kommandozeilentool nvprof

6.1.1 Allgemeines

Seit CUDA 5 enthält das CUDA-Toolkit [Cud] das leistungsfähige Tool *nvprof*. Bei nvprof handelt es sich um einen Kommandozeilen-Profiler, der für Linux, Windows und OS X zur Verfügung steht. [Har13] Mit Hilfe von nvprof können Daten gesammelt und auf der Kommandozeile angezeigt werden. Zu den Daten zählen CUDA-bezogene Aktivitäten auf CPU und GPU (Kernelausführungen, Speichertransfers und Aufrufe der CUDA-API) sowie Metriken und Ereignisse von CUDA-Kerneln. [NVI18b, Kapitel 3] Da auch mit OpenACC-Direktiven parallelisierter C, C++ und Fortran-Programme mit Hilfe eines OpenACC-Compilers die CUDA-Funktionalität nutzen, können auch OpenACC-Programme mit nvprof untersucht werden.

Um eine Anwendung von der Kommandozeile aus zu untersuchen kann nvprof wie folgt verwendet werden:

```
nvprof [Optionen] [Anwendung] [Anwendungsargumente]
```

Hinweis: Standardmäßig wird die Ausgabe von nvprof auf `stderr` umgeleitet. Um die Ausgabe in eine Datei umzuleiten kann die Option `--log-file <Name der Logdatei>` verwendet werden.

6.1.2 Profiling Modes

Die vier wichtigsten Modi, um Anwendungen mit nvprof zu untersuchen sind der *Summary Mode*, der *GPU-Trace and API-Trace Mode*, der *Event/Metric Summary Mode* und der *Event/Metric Trace Mode*. [NVI18b, Abschnitt 3.2]

Alle weiteren Optionen sind unter [NVI18b, Abschnitt 3.1] oder der Hilfeseite von nvprof zu finden:

```
nvprof --help
```

6.1.2.1 Summary Mode

Der *Summary Mode* ist der standardmäßige Modus von nvprof. In diesem Modus gibt nvprof eine einzelne Ergebniszeile für jede Kernelfunktion und jeden Typ von CUDA-Speichertransfers aus, der von der Anwendung ausgeführt wird. Zusätzlich wird für jeden Kernel die Gesamtzeit, der Durchschnitt sowie die minimale und maximale Zeit aller Kernel- und Speichertransfer-Aufrufe angezeigt. Die Ausführungszeit spiegelt dabei jeweils die benötigte Zeit auf dem verwendeten Gerät wieder. Um den *Summary Mode* von nvprof zu verwenden, kann das folgende Kommando verwendet werden:

```
nvprof <Anwendung> <Anwendungsargumente>
```

Das Ergebnis für den Code 3.2 befindet sich in der Datei:

```
code/profiling/results/summary_mode.log
```

6.1.2.2 GPU-Trace and API-Trace Mode

Neben dem Summary Mode kann das Verhalten einer Anwendung mit dem *GPU-Trace and API-Trace Mode* untersucht werden. Beide Modi können einzeln oder zusammen aktiviert werden. Der GPU-Trace Mode zeigt in chronologischer Reihenfolge alle Aktivitäten auf der GPU zum Beispiel Kernelausführungen und Speichertransfers an. Für jeden Kernel oder jeden Speichertransfer werden detaillierte Informationen wie Kernelparameter, Shared-Memory-Nutzung und Speichertransfer-Durchsatz gemessen und angezeigt. Der API-Trace Mode zeigt auf einer Zeitskala alle CUDA- und Treiber-API Aufrufe des Hosts in chronologischer Reihenfolge an. Die folgenden Kommandos können zur Aktivierung des GPU-Trace Mode und des API-Trace Mode verwendet werden:

```
nvprof --print-gpu-trace <Anwendung> <Anwendungsargumente>
nvprof --print-api-trace <Anwendung> <Anwendungsargumente>
```

Das Ergebnis für den Code 3.2 befindet sich in den beiden folgenden Dateien:

```
code/profiling/results/print-gpu-trace_mode.log
code/profiling/results/print-api-trace_mode.log
```

6.1.2.3 Event/Metric Summary Mode

Ein weiterer Modus von nvprof zur Profilierung einer Anwendung ist der *Event/Metric Summary Mode*. Dieser Modus kann genutzt werden, um Ereignisse und Metriken auf einer NVIDIA-GPU anzuzeigen. Eine Liste aller verfügbaren Ereignisse und Metriken kann mit den folgenden Kommandos angezeigt werden.

```
Liste aller verfügbaren Ereignisse:
nvprof --query-events
Liste aller verfügbaren Metriken:
nvprof --query-metrics
```

Das Profiling Tool nvprof kann mehrere Ereignisse/Metriken gleichzeitig erfassen. Hier ist ein Beispiel:

```
nvprof --events warps_launched,local_load --metrics ipc <Anwendung>
<Anwendungsargumente>
```

Das Ergebnis für den Code 3.2 unter Verwendung des Beispielkommandos befindet sich in der Datei:

```
code/profiling/results/eventMetric-summary_mode.log
```

6.1.2.4 Event/Metric Trace Mode

Im *Event/Metric-Trace Mode* werden Event- und Metrikwerte für jede Kernel-Ausführung angezeigt. Standardmäßig werden Ereignis- und Metrikwerte für alle Einheiten der GPU aggregiert. Zum Beispiel werden multiprozessorspezifische Ereignisse über alle Multiprozessoren auf der GPU aggregiert. Wenn `--aggregate-mode off` angegeben wird, werden die Werte jeder Einheit angezeigt.

Im folgenden Beispiel wird beispielsweise der Ereigniswert für jeden Multiprozessor der GPU angezeigt:

```
nvprof --aggregate-mode off --events local_load --print-gpu-trace
<Anwendung> <Anwendungsargumente>
```

Das Ergebnis für den Code 3.2 unter Verwendung des Beispielkommandos befindet sich in der Datei:

```
code/profiling/results/eventMetric-trace_mode.log
```

6.2 NVIDIA® Visual Profiler

Der NVIDIA® Visual Profiler (siehe Abbildung 6.1) ist ein 2008 eingeführtes, plattformübergreifendes Profiling Tool, dass zum CUDA-Toolkit gehört und zur Optimierung von CUDA- und OpenACC-Anwendungen verwendet werden kann. Es unterstützt alle seit 2006 unter Linux, Mac OS X und Windows ausgelieferten CUDA-fähigen NVIDIA-Grafikprozessoren. [nvi18a]

Die folgenden Abschnitte geben einen Einblick in die Verwendung des Visual Profilers von NVIDIA.

6.2.1 Eine Anwendung zum Profiling vorbereiten

Der Visual Profiler von NVIDIA erfordert keine Anwendungsänderungen. Trotzdem kann durch einfache Änderungen und Ergänzungen die Benutzerfreundlichkeit und die Effektivität am Anwendungscode erheblich gesteigert werden. Einige Hinweise sind unter [NVI18b, Kapitel 1] zu finden.

6.2.2 Eine Session erstellen

Der erste Schritt beim Einsatz des Visual Profilers besteht in der Erstellung einer Profiling-Session. Eine Session enthält die mit Ihrer Anwendung verbundenen Einstellungen, Daten und Ergebnisse.

6.2.2.1 Eine neue ausführbare Session erstellen

Eine neue, ausführbare Session kann über den Eintrag `Profile An Application` der Begrüßungsseite oder über den Eintrag `New Session` im Menüpunkt `File` erstellt werden. Die Erstellung wird vom Programm geführt. Dabei muss das zu untersuchende Programm sowie die Profiling-Optionen spezifiziert werden.

Nach der Erstellung wird das angegebene Programm mehrfach vom Visual Profiler ausgeführt und alle für die Analyse notwendigen Metriken und Ereignisse gesammelt.

6.2.2.2 Import einer Session

Um eine Session zu importieren, muss die Option `Import...` im Menüpunkt `File` gewählt werden. Anschließend führt ein Dialog durch den Importvorgang.

Die notwendigen Metriken und Ereignisse müssen zuvor mit dem Kommandozeilentool nvprof gesammelt werden. Eine Anleitung befindet sich im Abschnitt 6.3.

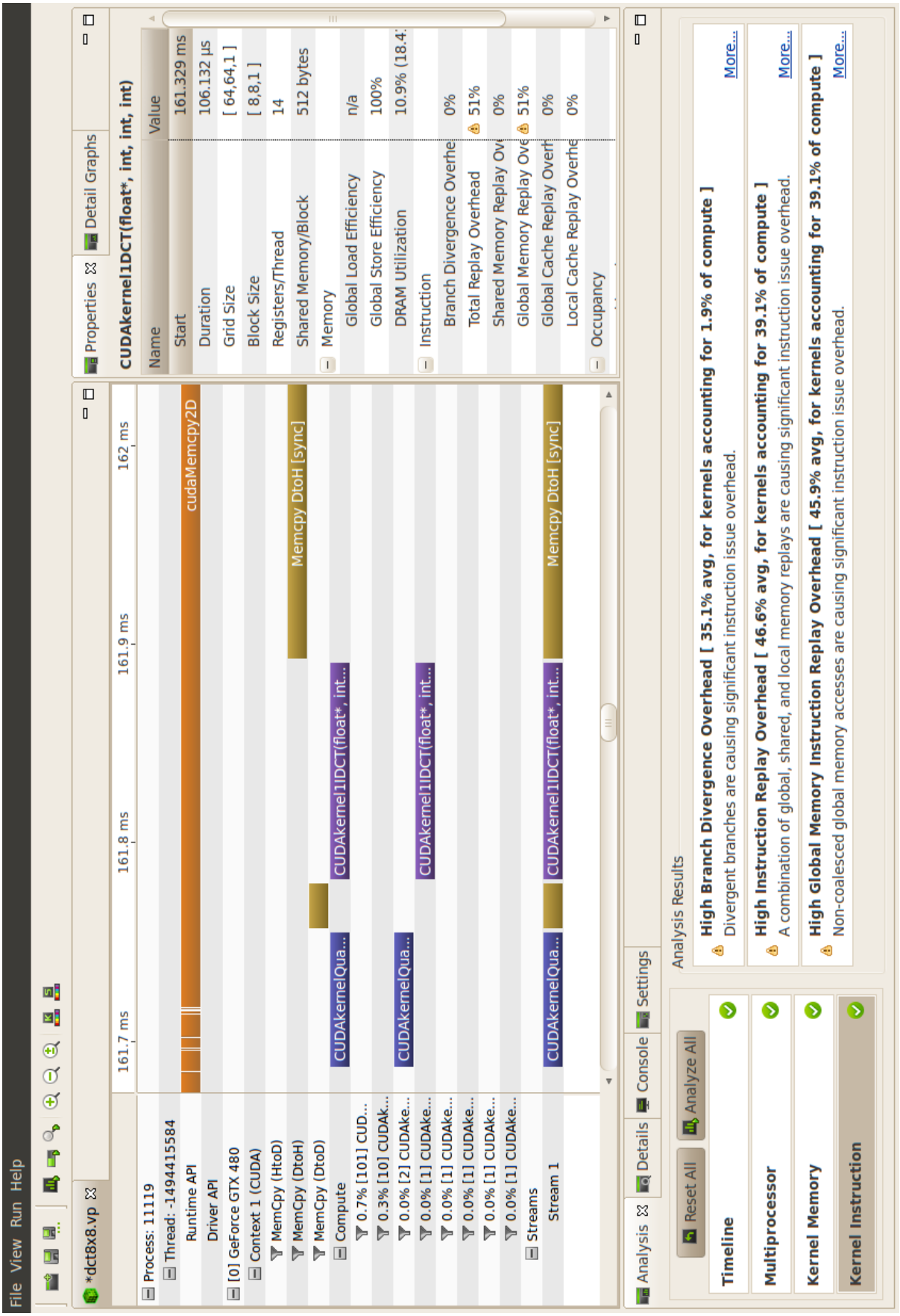

Abbildung 6.1: NVIDIA® Visual Profiler [nvi18a]

6.2.3 Anwendungsanalyse

Wurde bei der Erstellung der Session die Option `Run guided analysis` gewählt, so führt der Visual Profiler sofort die Anwendung aus, um alle notwendigen Metriken und Ereignisse zu bestimmen, die für die erste Stufe der geführten Analyse notwendig sind. Anschließend kann das Analysesystem verwendet werden, um Gründe für das leistungsbegrenzte Verhalten der Anwendung zu erhalten. Ein Beispiel ist in Abbildung 6.2 zu sehen.

6.2.4 Die Timeline erkunden

Zusätzlich zum geführten Analysesystem wird in der Timeline des Visual Profilers die CPU- und GPU-Aktivität angezeigt, die während der Anwendungsausführung aufgetreten ist (vgl. Abbildung 6.3). Am oberen Rand der Timeline befindet sich ein horizontales Lineal, dass die verstrichene Zeit seit dem Start des Profilings angibt. Auf der linken Seite ist ein vertikales Lineal, dass beschreibt was in jeder horizontalen Zeile der Timeline dargestellt ist.

Weitere Informationen zum Aufbau und der Steuerung der Timeline können unter [NVI18b, Abschnitt 2.4.1] nachgelesen werden.

6.2.5 Weitere Details

Zusätzlich zu den Ergebnissen in der Analyseansicht können auch die im Rahmen der Analyse gesammelten spezifischen Metriken und Ereigniswerte betrachtet werden. Diese sind unter dem Eintrag `GPU-Details` zu finden. Dabei können spezifische Metriken und Ereigniswerte gesammelt werden, die das Verhalten der Kernel in der Anwendung aufzeigen (vgl. Abbildung 6.4).

Weitere Informationen zur `GPU-Details`-Ansicht können unter [NVI18b, Abschnitt 2.4.4] nachgelesen werden.

6.3 Remote Profiling

Remote Profiling kann genutzt werden, um Daten auf einem entfernten System, auf das nur per Terminal zugegriffen werden kann, zu sammeln. Die gesammelten Daten können anschließend auf dem Host-System angezeigt und analysiert werden. Im Folgenden werden zwei Varianten kurz vorgestellt. [NVI18b, Kapitel 4]

6.3.1 Mit nvprof

Das Kommandozeilentool nvprof kann zunächst manuell auf dem Remote System ausgeführt werden, um alle notwendigen Daten zu bestimmen. Anschließend können die Daten

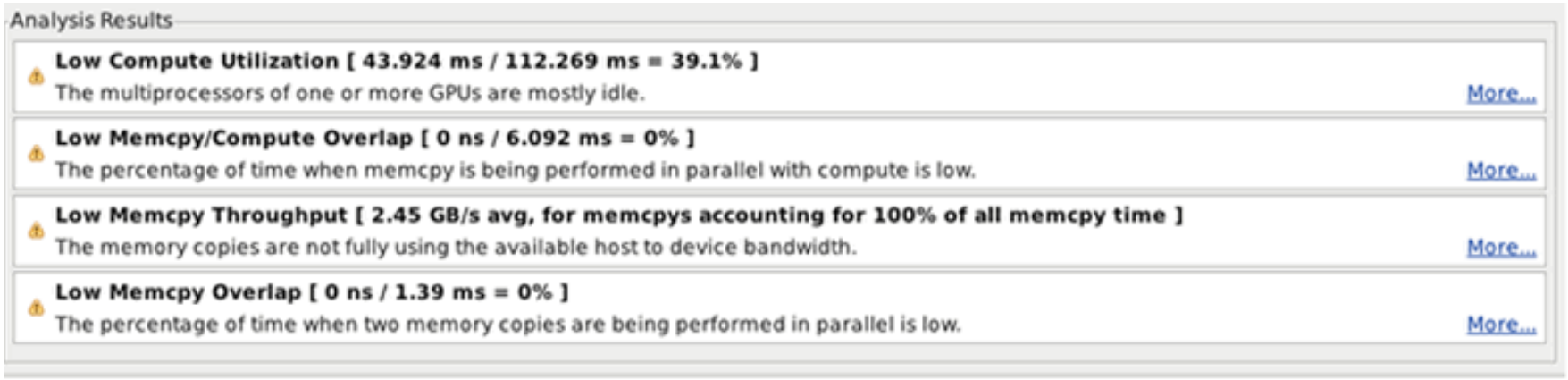

Abbildung 6.2: NVIDIA® Visual Profiler: Das Analysesystem listet einige Gründe für leistungsbegrenzte Verhalten der Anwendung auf.

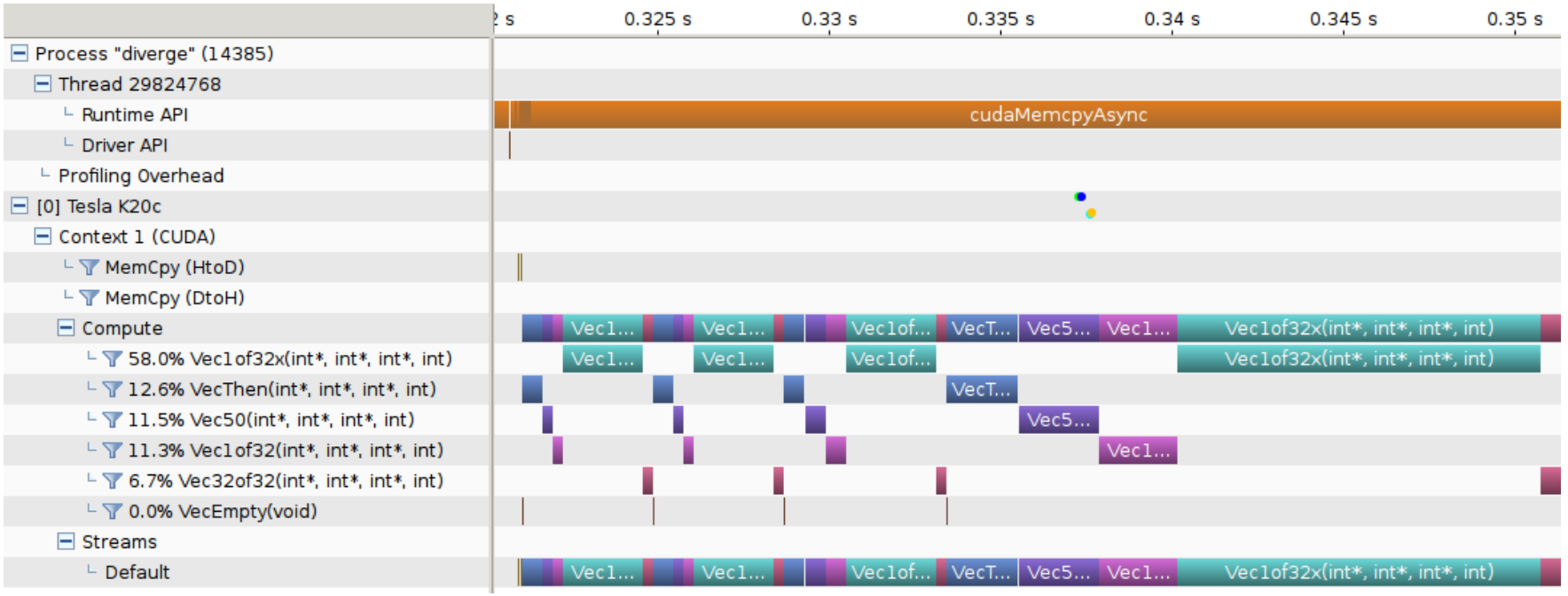

Abbildung 6.3: NVIDIA® Visual Profiler: Timeline [NVI18b]

Analysis | Details | Console | Settings

Name	Start Time	Duration	Grid Size	Block Size	Regs	Static SMem	Dynamic SMem	Size	Throughput
__pgi_uacc_cuda_fill_32_gpu	246.812 ms	5.088 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
Memcpy HtoD [async]	273.629 ms	325.572 µs	n/a	n/a	n/a	n/a	n/a	4 MB	12.286 GB/s
__pgi_uacc_cuda_fill_32_gpu	274.005 ms	4.864 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
Memcpy HtoD [async]	274.492 ms	324.292 µs	n/a	n/a	n/a	n/a	n/a	4 MB	12.335 GB/s
__pgi_uacc_cuda_fill_32_gpu	274.824 ms	4.768 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
MatrixMult_26_gpu	274.853 ms	10.52 ms	[63,16,1]	[16,32,1]	32	0	0	n/a	n/a
Memcpy DtoH [async]	285.43 ms	305.924 µs	n/a	n/a	n/a	n/a	n/a	4 MB	13.075 GB/s
__pgi_uacc_cuda_fill_32_gpu	288.084 ms	4.64 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
Memcpy HtoD [async]	288.477 ms	324.772 µs	n/a	n/a	n/a	n/a	n/a	4 MB	12.316 GB/s
__pgi_uacc_cuda_fill_32_gpu	288.812 ms	4.672 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
Memcpy HtoD [async]	288.895 ms	324.1 µs	n/a	n/a	n/a	n/a	n/a	4 MB	12.342 GB/s
__pgi_uacc_cuda_fill_32_gpu	289.231 ms	4.64 µs	[8,1,1]	[128,1,1]	24	0	0	n/a	n/a
MatrixMult_26_gpu	289.246 ms	10.563 ms	[63,16,1]	[16,32,1]	32	0	0	n/a	n/a

Abbildung 6.4: NVIDIA® Visual Profiler: GPU-Details

in den NVIDIA® Visual Profiler importiert und analysiert werden. Beide Schritte werden als Nächstes kurz beschrieben.

6.3.1.1 Daten sammeln

Es gibt drei typische Anwendungsfälle, die im Folgenden kurz vorgestellt werden.

Timeline

Der erste Anwendungsfall ist das Bestimmen der Timeline einer Anwendung. Dazu wird auf dem entfernten System das folgende Kommando ausgeführt:

```
nvprof -export-profile timeline.prof <Anwendung> <Anwendungsargumente>
```

Die von nvprof gesammelten Daten werden in der Datei `timeline.prof` gespeichert. Danach sollte die Datei zurück auf das Host-System kopiert werden, um sie in den NVIDIA® Visual Profiler zu importieren. Die Importierung ist im Abschnitt 6.3.1.2 beschrieben.

Das Ergebnis für den Code 3.2 befindet sich in der Datei:

```
code/profiling/remote/timeline.prof
```

Metriken und Ereignisse

Der zweite Anwendungsfall ist das Sammeln von Metriken und Ereignissen von allen Kerneln einer Anwendung. Um die Daten so exakt wie möglich zu bestimmen, werden alle Kernelausführungen auf der GPU serialisiert. Anschließend werden die gesammelten Daten in einer Timeline zusammengeführt und es entsteht ein genaues Bild des Anwendungsverhaltens. Die Metriken und Ereignisse können nach dem Flag `--metrics` für Metriken und `--events` für Ereignisse spezifiziert werden. Daher können bei einem Aufruf von nvprof beliebig viele Metriken und Ereignisse gesammelt werden. Das folgende Kommando nutzt das `--metrics`-Flag, um zwei Metriken zu bestimmen:

```
nvprof -metrics achieved_occupancy,executed_ipc -o metrics.prof
<Anwendung> <Anwendungsargumente>
```

Die von nvprof gesammelten Daten werden in der Datei `metrics.prof` gespeichert. Danach sollte die Datei zurück auf das Host-System kopiert werden, um sie in den NVIDIA® Visual Profiler zu importieren. Die Importierung ist im Abschnitt 6.3.1.2 beschrieben.

Das Ergebnis für den Code 3.2 befindet sich in der Datei:

```
code/profiling/remote/metrics.prof
```

Analyse eines einzelnen Kernels

Die letzte Variante kann dazu genutzt werden, um Metriken eines einzelnen Kernels zu bestimmen. Dabei ist besonders wichtig, dass die Option `--kernels` genutzt und vor der Option `--analysis-metrics` notiert wird, da nur so die Metriken für den oder die angegebenen Kernel gesammelt werden. Das folgende Kommando zeigt den Aufbau des Kommandos:

```
nvprof -kernels <Kernel> -analysis-metrics -o analysis.prof
<Anwendung> <Anwendungsargumente>
```

Die von nvprof gesammelten Daten werden in der Datei `analysis.prof` gespeichert. Danach sollte die Datei zurück auf das Host-System kopiert werden, um sie in den NVIDIA® Visual Profiler zu importieren. Der Import ist im Abschnitt 6.2.2.2 beschrieben.

Das Ergebnis für den Code 3.2 befindet sich in der Datei:

```
code/profiling/remote/analysis.prof
```

6.3.1.2 Daten ansehen und analysieren

Um die mit nvprof bestimmten Daten in dem NVIDIA® Visual Profiler anzuzeigen und zu analysieren, müssen sie importiert werden. Weitere Details zum Import werden im Abschnitt 6.2.2.2 beschrieben.

Timeline, Metriken und Ereignisse

Die Daten der Timeline (z.B. die `timeline.prof`-Datei von Abschnitt 6.3.1.1) können über die im NVIDIA® Visual Profiler integrierte Importfunktion importiert werden. Zusätzlich können die gesammelten Metriken und Ereignisdaten (z.B. die `metrics.prof`-Datei von Abschnitt 6.3.1.1) genutzt werden, um die Kernel der Timeline mit zusätzlichen Informationen anzureichern.

Geführte Analyse für einzelnen Kernel

Um die gesammelten Analysedaten eines einzelnen Kernels (z.B. die `analysis.prof`-Datei von Abschnitt 6.3.1.1) im NVIDIA® Visual Profiler anzuzeigen, muss die entsprechenden Datei importiert werden. Nach dem Import kann das vom NVIDIA® Visual Profiler geführte Analysesystem verwendet werden, um den Kernel weiter zu optimieren.

6.3.2 Mit NVIDIA® Visual Profiler

Auch der NVIDIA® Visual Profiler kann zum Remote Profiling genutzt werden. Eine Anleitung befindet sich unter: [NVI18b, Abschnitt 4.1].

Anhang A

ls4gpu1 und ls4gpu2 am Lehrstuhl 4

Der Lehrstuhl 4 der Fakultät für Informatik TU Dortmund verfügt über Computer, mit rechenstarken Grafikkarten.

Die Workstation `ls4gpu1` ist mit dem Befehl `ssh ls4gpu1.cs.tu-dortmund.de` erreichbar. Sie enthält eine Intel® Xeon® E5-2690 v4 CPU und eine NVIDIA® Quadro® K6000 Grafikkarte. Der Arbeitsspeicher (RAM) beträgt 32 GB.

Die Workstation `ls4gpu2` ist mit einer Intel® Xeon® E5-2699 v4 CPU und zwei NVIDIA® Quadro® P6000 Grafikkarten ausgestattet. Außerdem sind 64GB RAM enthalten. Erreichbar ist der Server mit dem Befehl `ssh ls4gpu2.cs.tu-dortmund.de`.

Auf beiden Workstations ist der `gcc7` installiert, der mit Befehl `gcc-7` genutzt werden kann.

A.1 NVIDIA® Quadro® K6000 und P6000

In der Tabelle A.1 sind die wichtigsten Informationen zu den Grafikkarten NVIDIA® Quadro® K6000 und P6000 zu finden.

	Quadro® K6000	**Quadro® P6000**
NVIDIA CUDA Cores	2880	3840
Compute Capability	3.5	6.1
Maximums		
Threads per Block	1024	1024
Threads per Multiprocessor	2048	2048
Shared Memory per Block	48 KiB	48 KiB
Shared Memory per Multiprocessor	–	96 KiB
Registers per Block	65536	65536
Registers per Multiprocessor	–	65536

Grid Dimensions	[2147483647, 65535, 65535]	[2147483647, 65536, 65536]
Block Dimensions	[1024, 1024, 64]	[1024, 1024, 64]
Warps per Multiprocessor	64	64
Blocks per Multiprocessor	32	32
Half Precision FLOP/s	–	98,7 GigaFLOP/s
Single Precision FLOP/s	–	12,634 TeraFLOP/s
Double Precision FLOP/s	–	394,8 GigaFLOP/s
Multiprocessor		
Multiprocessor	15	30
Clock Rate	902 MHz	1.645 MHz
Concurrent Kernel	true	true
Max IPC	7	6
Threads per Warp	32	32
Memory		
Global Memory Bandwidth	288,384 GB/s	433,248 GB/s
Global Memory Size	11,992 GiB	23,871 GiB
Constant Memory Size	64 KiB	64 KiB
L2 Cache Size	1,5 MiB	3 MiB
Memcpy Engines	2	2
PCIe		
Generation	3	3
Link Rate	8 Gbit/s	8 Gbit/s
Link Width	16	16

Tabelle A.1: NVIDIA® Quadro® K6000 und P6000 Spezifikationen. [nvi]

A.2 Intel® Xeon® E5-2690 v4 und E5-2699 v4

In der Tabelle A.2 sind die wichtigsten Informationen zum Prozessor Intel® Xeon® E5-2690 v4 und E5-2699 v4 zu finden.

	Intel® Xeon® E5-2690 v4	**Intel® Xeon® E5-2699 v4**
Number of Cores	14	22
Number of Threads	28	44
Processor Base Frequency	2,60 GHz	2,20 GHz
Max Turbo Frequency	3,50 GHz	3,60 Ghz
Cache	35 MB SmartCache	55 MB SmartCache
Bus Speed	9.6 GT/s QPI	9,6 GT/s QPI
Number of QPI Links	2	2
TDP	135 W	145 W
Memory Specifications		
Max Memory Size	1,54 TB	1,54 TB
Memory Types	DDR4 1600/1866/2133/2400	DDR4 1600/1866/2133/2400

Max Number of Memory Channels	4	4
Max Memory Bandwidth	76,8 GB/s	76,8 GB/s
Physical Address Extensions	46-bit	46-bit
ECC Memory Supported	Yes	Yes
Expansion Options		
Scalability	2S	2S
PCI Express Revision	3.0	3.0
PCI Express Configurations	x4, x8, x16	x4, x8, x16
Max Number of PCI Express Lanes	40	40

Tabelle A.2: Intel® Xeon® E5-2690 v4 und E5-2699 v4 Spezifikation. [int16a, int16b]

Anhang B

PGI-Compiler

B.1 Allgemeines

PGI (ehemals The Portland Group, Inc.) war ein Unternehmen, das eine Reihe von kommerziell erhältlichen Fortran-, C- und C++-Compilern und Tools für Linux-, Mac- und Windows-Workstations, -Server und -Cluster erstellte. Diese Compiler und Tools umfassen die PGF77-, PGFORTRAN-, PGC++- und PGCC-Compiler, den PGI-Profiler und den PGI-Debugger. Die Compiler können insbesondere für die Erstellung von parallelen multicore Anwendungen mit OpenMP, MPI, CUDA oder OpenACC für Intel und AMD CPUs sowie AMD und NVIDIA GPUs genutzt werden. Seit dem 29. Juli 2013 gehört das Unternehmen zur NVIDIA Corporation. Alle Tools werden seitdem unter der Leitung von NVIDIA weiterentwickelt und sind weiterhin kommerziell erhältlich. Eine Ausnahme bildet die PGI Community Edition, die unter bestimmten Voraussetzungen kostenlos genutzt werden kann. [PGIb, PGIc]

Hinweis: Die 3. Generation des Linux Cluster Dortmund (LiDO3), dass vom IT & Medien Centrum (ITMC) der Technischen Universität Dortmund betrieben wird, bietet den Nutzerinnen und Nutzer die Möglichkeit PGI-Compiler und -Tools zu nutzen. [LiD, Folie 30]

OpenACC	PGI-Version	Bemerkung
1.0	12.6	Ausnahme: deviceptr() in Ftn ab Version 14.1
2.0	15.1	–
2.5	17.1	–
2.6	18.3	–

Tabelle B.1: OpenACC-Unterstützung in PGI. [PGIa]

Ein Vorteil des PGI-Compilers gegenüber dem gcc-Compiler ist die umfangreiche Unterstützung von OpenACC. Die Tabelle B.1 gibt einen Überblick über die Compilerversion und die unterstützte OpenACC-Spezifikation. Die neuste Version 18.7 ist am 16.8.2018 erschienen. Die jeweils aktuelle Version kann auf der folgenden Webseite [PGId] nachgelesen werden.

B.2 OpenACC-Kompilierung mit PGCC und PGC++

Die beiden folgenden Befehle zeigen die Kompilierung von C- bzw. C++-Programmen mit dem PGI-Compiler PGCC und PGC++.

```
pgcc <Programmname>.c -o <Programmname>
pgc++ <Programmname>.c -o <Programmname>
```

Wichtige Optionen:

- -acc → Aktiviert die OpenACC-Unterstützung.
- -ta=nvidia → Code für eine NVIDIA-GPU generieren.
- -ta=host → Code für eine Host-CPU generieren.
- -ta=nvidia,host → Code für Host-CPU und NVIDIA-GPU generieren.
- -Minfo=all → Zeige alle Kompilierungsinformationen.
- -fastsse → Das optimale Flag für Geräte, die SSE/SSE2-fähig sind.
- -OX → Die Compiler-Optimierung X ($1 \leq X \leq 4$) einstellen.
- -mp → Aktiviert die OpenMP-Unterstützung.

Beispiele:

Zur Kompilierung des OpenACC-Programms in Code 2.1 kann der folgende Befehl genutzt werden:

```
pgcc <Programmname>.c -acc -ta=nvidia -o <Programmname> -Minfo=all
```

Das OpenMP-Programm in Code 2.2 kann mit diesem Befehl kompiliert werden:

```
pgcc <Programmname>.c -mp -fastsse -o <Programmname> -Minfo=all
```

B.3 Profiling

Auch das Profiling von OpenACC-Programmen profitiert von der Kompilierung mit PGI-Compilern. Dadurch sind im Kommandozeilentool nvprof (siehe Abschnitt 6.1) weitere und bessere Optionen möglich, die direkt die OpenACC-Pragmas und Funktionen auflisten und mit Ausführungszeiten versehen.

Die drei zusätzlichen Optionen:

- `--print-openacc-summary`

→ Eine Zusammenfassung des OpenACC-Profilingergebnis ausgeben.

- `--print-openacc-trace`

→ Das OpenACC-Profilingergebnis in Ablaufreihenfolge anzeigen.

- `--print-openacc-constructs`

→ Zeigt die übergeordneten Routinennamen (z.B. CUDA-Funktionen) im OpenACC-Profilingergebnis an. Außerdem wird die Rechen- und Übertragungszeit ausgeben.

Die vorgestellten Optionen können wie folgt verwendet werden:

```
nvprof --print-openacc-summary <Programmname>
nvprof --print-openacc-trace <Programmname>
nvprof --print-openacc-constructs <Programmname>
```

Hinweis: Bei Programmen, die mit dem gcc kompiliert wurden, führen alle drei Optionen zu leeren Ausgaben!

Literaturverzeichnis

[Bey15] J.C. Beyer. Comparing OpenACC 2.5 and OpenMP 4.1. https://openmpcon.org/wp-content/uploads/openmpcon2015-james-beyer-comparison.pdf, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[Bor17] E. Born. PCIe 4.0 specification finally out with 16 GT/s on tap. https://techreport.com/news/32064/pcie-4-0-specification-finally-out-with-16-gt-s-on-tap, 2017. [Online; zuletzt abgerufen: 03. Juli 2018].

[Bro18] BrookGPU Homepage. http://graphics.stanford.edu/projects/brookgpu/, 2018. [Online; zuletzt abgerufen: 5. Juli 2018].

[CS18] I.D. Chivers and J. Sleightholme. *Introduction to Programming with Fortran*. Springer International Publishing, 2018.

[Cud] CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit. [Online; zuletzt abgerufen: 26. Juni 2018].

[CUD18a] CUDA Toolkit Documentation. https://docs.nvidia.com/cuda, 2018. [Online; zuletzt abgerufen: 22. Juni 2018].

[CUD18b] CUDA Toolkit Documentation - CUDA C Kernels. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#kernels, 2018. [Online; zuletzt abgerufen: 22. Juni 2018].

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[Hag16] A. Hagemeier. Einführung in die Parallelprogrammierung. http://www.fz-juelich.de/ias/jsc/EN/AboutUs/Staff/Hagemeier_A/docs-parallel-programming/Parallel-Programming-Basics.pdf?__blob=publicationFile, 2016. [Online; zuletzt abgerufen: 25. Juni 2018].

[Har13] M. Harris. nvprof is Your Handy Universal GPU Profiler. https://devblogs.nvidia.com/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler, 2013. [Online; zuletzt abgerufen: 12. Juni 2018].

[HL08] S. Hoffmann and R. Lienhart. *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*. Informatik im Fokus. Springer Berlin Heidelberg, 2008.

[int16a] Intel® Xeon® Processor E5-2690 v4 Specification. https://ark.intel.com/products/91770/Intel-Xeon-Processor-E5-2690-v4-35M-Cache-2_60-GHz, 2016. [Online; zuletzt abgerufen: 12. Juni 2018].

[int16b] Intel® Xeon® Processor E5-2690 v4 Specification. https://ark.intel.com/en/products/91317/Intel-Xeon-Processor-E5-2699-v4-55M-Cache-2_20-GHz, 2016. [Online; zuletzt abgerufen: 12. Juni 2018].

[KH16] D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 3 edition, 2016.

[Kro18] Krohnos Group. https://www.khronos.org/about/, 2018. [Online; zuletzt abgerufen: 5. Juli 2018].

[KVBC12] H. Kim, R. Vuduc, S. Baghsorkhi, and J. Choi. *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.

[LiD] Einführung in LiDO3. https://www.lido.tu-dortmund.de/cms/de/LiDO3/lido3kurz.pdf. [zuletzt abgerufen: 23. August 2018].

[LJ15] J. Larkin and G. Juckeland. Comparing OpenACC and OpenMP Performance and Programmability. http://on-demand.gputechconf.com/gtc/2015/presentation/S5196-Jeff-Larkin.pdf, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[MBW08] P. Mandl, A. Bakomenko, and J. Weiß. *Grundkurs Datenkommunikation*. Technische und Ingenieurinformatik. Vieweg + Teubner, 2008.

[NFH07] A. Nischwitz, M. Fischer, and P. Haberäcker. *Computergrafik und Bildverarbeitung: Alles für Studium und Praxis - Bildverarbeitungswerkzeuge, Beispiel-Software und interaktive Vorlesungen online verfügbar*. Vieweg+Teubner Verlag, 2007.

[nvi] NVIDIA® Visual Profiler (Version: 8.0). Link zum Tool: https://developer.nvidia.com/nvidia-visual-profiler. [zuletzt abgerufen: 12. Juni 2018].

[nvi18a] NVIDIA Visual Profiler - Homepage. https://developer.nvidia.com/nvidia-visual-profiler, 2018. [Online; zuletzt abgerufen: 9. Juli 2018].

[NVI18b] Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2018. [Online; zuletzt abgerufen: 12. Juni 2018].

[Ope98] OpenMP Specification 1.0 for C and C++. https://www.openmp.org/wp-content/uploads/cspec10.pdf, 1998. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope11] OpenACC Specification 1.0. https://www.openacc.org/sites/default/files/inline-files/OpenACC_1_0_specification.pdf, 2011. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope15a] Final Comment Draft for the OpenMP 4.1 Specification. https://www.openmp.org/wp-content/uploads/cspec10.pdf, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope15b] Is OpenACC The Best Thing To Happen To OpenMP? https://www.nextplatform.com/2015/11/30/is-openacc-the-best-thing-to-happen-to-openmp, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope15c] OpenACC support in GCC. http://scelementary.com/2015/04/25/openacc-in-gcc.html, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope15d] OpenMP Specification 2.5. https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf, 2015. [Online; zuletzt abgerufen: 26. Juli 2018].

[Ope15e] OpenMP Specifications. https://www.openmp.org/specifications, 2015. [Online; zuletzt abgerufen: 21. Juni 2018].

[Ope18a] OpenACC Specification. https://www.openacc.org/specification, 2018. [Online; zuletzt abgerufen: 14. Juni 2018].

[Ope18b] OpenCL Specifications. https://www.khronos.org/registry/OpenCL/specs/, 2018. [Online; zuletzt abgerufen: 5. Juli 2018].

[PGIa] PGI Accelerator Compilers with OpenACC Directives. https://www.pgroup.com/resources/accel.htm#spec. [zuletzt abgerufen: 23. August 2018].

[PGIb] PGI Community Edition. https://www.pgroup.com/products/community.htm. [zuletzt abgerufen: 23. August 2018].

[PGIc] PGI Product Feature Comparison. https://www.pgroup.com/products/feature-compare.htm. [zuletzt abgerufen: 23. August 2018].

[PGId] PGI Release Archive. https://www.pgroup.com/support/release_archive.php. [zuletzt abgerufen: 23. August 2018].

[Roy90] R. Roy. The Discovery of the Series Formula for π by Leibniz, Gregory and Nilakantha. *Mathematics Magazine*, 63(5):291–306, 1990.

[RR12] T. Rauber and G. Rünger. *Parallele Programmierung*. eXamen.press. Springer Berlin Heidelberg, 2012.