TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR TECHNISCHE INFORMATIK
PROFESSUR FÜR RECHNERARCHITEKTUR
PROF. DR. WOLFGANG E. NAGEL

Profilmodul Forschungsprojekt Grundlagen

Innovative Spracherweiterungen für Beschleunigerkarten am Beispiel von SYCL, HC, HIP und CUDA: Untersuchung zu Nutzbarkeit und Performance

Jan Stephan
(Mat.-Nr.: 3755136)

Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuer: Dr.-Ing. Bernd Trenkler
Matthias Werner, M.Sc.

Dresden, 21. Februar 2019

Inhaltsverzeichnis

Glossar

Device Ein von der CPU zu unterscheidendes Gerät für Berechnungen (Beschleuniger). Im Kontext dieser Arbeit stets eine GPU.

FLOP Fließkommaoperationen (engl. *floating point operations*).

Host Gerät, das einen Kernel auf dem Device ausführen lässt. Üblicherweise das Gerät, auf dem das Betriebssystem läuft, etwa ein PC oder ein Knoten auf einem Superrechner.

Kernel Programm, das auf einem Beschleuniger, wie etwa einer GPU, ausgeführt wird.

single-source compilation Die Quelltexte für Host und Device können sich in derselben Datei befinden und werden gleichzeitig vom Compiler verarbeitet.

split-source compilation Die Quelltexte für Host und Device werden getrennt verarbeitet. Der Host-Quelltext wird von einem normalen Compiler übersetzt, der Device-Quelltext von einem weiteren Compiler und unter Umständen erst zur Laufzeit des Programms.

Abkürzungsverzeichnis

API *application programming interface*

APU *accelerated processing unit*

CPU *central processing unit*

DSP *digital signal processor*

FLOPS *floating-point operations per second*

FMA *fused multiply-add*

FPGA *field programmable gate array*

GPGPU *general purpose computation on graphics processing unit*

GPU *graphics processing unit*

HC *Heterogeneous Compute API*

HCC *Heterogeneous Compute Compiler*

HIP *Heterogeneous-Computing Interface for Portability*

HPC *high-performance computing*

HSA *Heterogeneous System Architecture*

OpenCL *Open Compute Language*

OpenMP *Open Multi-Processing*

ROCm *Radeon Open Compute Platform*

SIMD *single-instruction, multiple data*

SIMT *single-instruction, multiple thread*

1 Einleitung

1.1 Motivation

Das Feld der *general purpose computation on graphics processing unit* (GPGPU) ist seit 2006 vor allem durch NVIDIAs CUDA-Plattform und GPUs geprägt und dominiert worden. Die Konkurrenz (vor allem der GPU-Hersteller AMD) konnte dem lange Zeit wenig entgegensetzen – heute (November 2018) sind NVIDIA-GPUs in fünf der zehn leistungsstärksten Supercomputer vorhanden, darunter die ersten beiden Plätze. Auf der Liste der 500 leistungsstärksten Supercomputer finden sich mit NVIDIA-GPUs bestückte Rechner insgesamt 126 Mal, GPUs des größten Konkurrenten AMD kein einziges Mal. Intels als GPU-Alternative vor- und mittlerweile zugunsten einer noch unbekannten zukünftigen Plattform (vgl. [Dam17]) eingestellten Xeon-Phi-Beschleuniger sind auf dieser Liste noch 30 Mal vertreten. (vgl. [TOP18])
Softwareseitig entstand ab 2008 mit der von Apple entwickelten und vom Khronos-Industriekonsortium standardisierten *Open Compute Language* (OpenCL) ein offener und hardware-übergreifender Konkurrent der CUDA-Plattform, der zunächst von einer Reihe wichtiger Hersteller (einschließlich NVIDIA) unterstützt wurde. Im Gegensatz zu CUDA wurde OpenCL von Anfang an auf verschiedene Arten von Beschleunigern ausgelegt, wie etwa Systeme mit einem *digital signal processor* (DSP) oder einem *field programmable gate array* (FPGA). Mit der Zeit zeigte sich jedoch, dass die Unterstützung durch die Hersteller mit jeder weiteren Revision des Standards abnahm; so ist die 2011 erschienene OpenCL-Version 1.2 bis heute die neueste Version, die von NVIDIA und AMD[1] vollständig unterstützt wird. Lediglich Intel bietet eine Implementierung eines modernen OpenCL-Standards an, beschränkt sich dabei aber auf seine CPUs und integrierten GPUs. Eine OpenCL-Implementierung für die neuesten Produktlinien der Xeon-Phi-Reihe, *Knights Landing* und *Knights Mill*, existiert nicht, während OpenCL 1.0 (2008) die einzige verfügbare Version für FPGAs darstellt.
Sowohl Khronos als auch AMD versuchen seit einigen Jahren, durch neue Technologien im Umfeld des *high-performance computing* (HPC) wieder Anschluss an NVIDIA zu finden. Im März 2014 stellte Khronos den SYCL-Standard vor, der auf OpenCLs Konzepten und Portabilität aufbaut, jedoch eine auf C++ basierende, deutlich modernere und angenehmere Programmierweise bieten soll. AMD folgte im April 2016 mit einer eigenen, quelloffenen Plattform namens *Radeon Open Compute Platform* (ROCm), die neben einer Portabilitätsschicht zu CUDA (*Heterogeneous-Computing Interface for Portability* (HIP)) einen auf AMD-GPUs spezialisierten C++-Dialekt (*Heterogeneous Compute API* (HC)) bietet.
Beide Ansätze befinden sich im Gegensatz zu CUDA noch am Anfang ihrer Entwicklung und erfahren laufend größere Änderungen sowie Feature-Updates. Eine vergleichende Untersuchung der bereits verfügbaren Fähigkeiten und Konzepte sowohl untereinander als auch im Vergleich zu CUDA ist jedoch möglich und in der wissenschaftlichen Literatur bisher nicht aufzufinden.

1.2 Forschungsstand

Die wissenschaftliche Literatur betrachtet SYCL vornehmlich als Abstraktionsschicht über OpenCL. Aus diesem Grund sind Untersuchungen, die sich direkt auf SYCL fokussieren, relativ selten.

[1] AMD unterstützte zwischenzeitlich OpenCL 2.0, ist jedoch infolge des Umbaus des eigenen Treiber- und Compute-Ökosystems auf OpenCL 1.2 zurückgegangen.

Das im März 2014 veröffentlichte SYCL wurde erstmals im August des selben Jahres von Trigkas untersucht. Zu diesem Zweck wurde SYCL mit OpenCL und *Open Multi-Processing* (OpenMP) verglichen und auf Intels Xeon-Phi-Beschleunigern ausgeführt. (vgl. [Tri14])
Cardoso da Silva et al. führten im Oktober 2016 ebenfalls einen Vergleich zwischen SYCL, OpenCL und OpenMP durch, als Hardware kamen hier Intels Xeon-CPUs zum Einsatz. (vgl. [CPB16])
Jesenšek stellte 2017 eine SYCL-Implementierung für Intels Xeon-Phi-Prozessoren und das Betriebssystem Linux vor. (vgl. [Jes17])
Im Mai 2017 zeigten Goli et al. dass SYCL zur Beschleunigung des Deep-Learning-Frameworks *TensorFlow* eingesetzt werden kann. (vgl. [GIR17])
Zur selben Zeit verwendeten Copik und Kaiser SYCL als Backend für das HPC-Programmiermodell *HPX.Compute*. (vgl. [CK17])
Im selben Monat untersuchten Doumoulakis et al. die Interoperabilität zwischen SYCL und OpenCL auf GPUs und FPGAs. (vgl. [DKO17])
Ebenfalls im selben Monat implementierten Aliaga et al. BLAS-Algorithmen auf der Basis von Ausdrucksbäumen und SYCL. (vgl. [ARG17])
St Clere Smithe und Potter stellten im Mai 2018 ein energieeffizientes neuronales Netzwerk vor, das mittels SYCL implementiert wurde. (vgl. [SP18])
Gleichzeitig zeigte Fare einen ersten Ansatz, der das Profiling von in SYCL geschriebenen Anwendungen ermöglicht. (vgl. [Far18])
Zur selben Zeit präsentierten Keryell und Yu eine Untersuchung von auf SYCL aufbauenden Programmen, die auf FPGAs ausgeführt werden. (vgl. [KY18])
Afzal et al. lösten im Juli 2018 Maxwell-Gleichungen mit SYCL und verglichen die Performance mit einer vorherigen Implementierung in der Programmiersprache C. (vgl. [ASA+18])
In der wissenschaftlichen Literatur tauchen die Begriffe ROCm, HC und HIP erstmals 2016 auf. Während sich die Arbeiten der ersten Jahreshälfte vornehmlich mit der *Heterogeneous System Architecture* (HSA) befassen und ROCm lediglich als verwendeten Software-Unterbau erwähnen (vgl. etwa [Li16], [LT16]), unternahm Sun im Juli 2016 den vermutlich ersten Vergleich zwischen HIP und CUDA (vgl. [Sun16]. Es folgten viele verschiedene Leistungsanalysen auf der Basis von ROCm:
Im September 2016 veröffentlichten Sun et al. ihre Benchmark-Suite *Hetero-Mark*, die vornehmlich die Leistungsfähigkeit von *accelerated processing unit* (APU)s misst und dabei auf ROCm und HC aufsetzt (vgl. [SGZ+16]). Sun et al. nutzten *Hetero-Mark* sowie das Framework *DNNMark* im April 2018 erneut, um ROCm einer eingehenden Performance-Analyse zu unterziehen (vgl. [SMB+18]). In die gleiche Richtung geht die im April 2017 von Gómez-Luna et al. vorgestellte Benchmark-Suite *Chai* (vgl. [GLEC+17]).
Im Mai 2017 stellten Hou et al. Benchmarks vor, die auf der Ebene der Register, des L1-Caches und des *shared memory* arbeiteten und in HC und CUDA implementiert wurden. (vgl. [HWF17])
Im Juli 2017 portierte Konstantinidis seine Sammlung von Micro-Benchmarks, die die Leistungsfähigkeit von CUDA-*graphics processing unit* (GPU)s auf der Instruktionsebene prüfen, mit der Hilfe von HIP auf die ROCm-Plattform. (vgl. [Kon17])
Im Januar 2018 untersuchten Nobre et al. die Performanz und Genauigkeit von Fließkommazahlen mit halber Präzision auf AMD GPUs. (vgl. [NRB+18])

1.3 Zielstellung

Das Ziel dieser Arbeit ist ein vergleichende Analyse der Programmiermodelle CUDA, SYCL und ROCm (bzw. HC und HIP) auf GPUs der Hersteller NVIDIA und AMD. Dabei sollen einerseits die den jeweiligen Modellen zugrunde liegenden Fähigkeiten und Konzepte verglichen, andererseits die konkret erreichbare Performanz anhand geeigneter Benchmarks ermittelt werden. Von den so gewonnenen Erkenntnissen werden Empfehlungen für den zukünftigen Einsatz dieser Modelle im HPC-Umfeld abgeleitet.

1.3.1 Anmerkung

Die vollständigen Quelltexte und Ergebnisse der Benchmarks sowie die LaTeX-Quelltexte dieser Arbeit sind öffentlich unter dem folgenden Link erreichbar: `https://github.com/j-stephan/fpg`

2 Funktionaler Vergleich der Spracherweiterungen

2.1 Anforderungen an Spracherweiterungen

Ein Vergleich der Fähigkeiten der einzelnen Spracherweiterungen erfordert einen Kriterienkatalog, anhand dessen die Bewertung erfolgen kann. Dabei sind sowohl die Sicht auf die Daten, also der Umgang der Spracherweiterungen mit den Daten, als auch die Sicht auf die Aufgaben, also die Transformation der Daten, zu berücksichtigen.

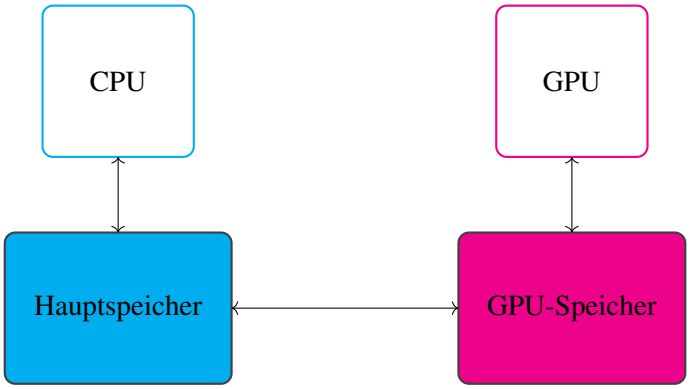
2.1.1 Datensicht

Der effiziente Umgang mit großen Datenmengen ist seit jeher eines der Kernprobleme des HPC im Allgemeinen und heterogener Systeme im Besonderen, d.h. unterschiedlicher Berechnungs-Hardware im selben Rechnersystem. Michael Wong[2] bezeichnete in einem Vortrag im November 2018 vor LLVM-Entwicklern die folgenden Punkte als essentielle Probleme der Programmierung heterogener Systeme (vgl. [Won18]):

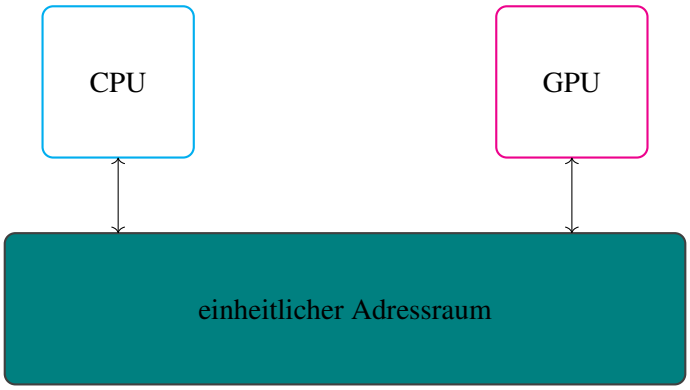
Datenbewegung Die Bewegung der Daten ist stets mit Kosten verbunden. Im Allgemeinen ist damit der Zeitbedarf gemeint, um Daten etwa vom Hauptspeicher des Systems in den Speicher der GPU zu kopieren. Im weiteren Sinne kann dieser Punkt aber auch andere Kriterien umfassen, wie z.B. den für den Kopiervorgang notwendigen Energiebedarf, der als Bestandteil des gesamten Energiebedarfs von HPC-Systemen in jüngerer Zeit ebenfalls Gegenstand der Forschung geworden ist.
Aus der Sicht des Programmierers ist außerdem die Frage relevant, ob die Datenbewegung *explizit* oder *implizit* erfolgt. Die *explizite* Datenbewegung hat den Vorteil, dass der Programmierer jederzeit die volle Kontrolle über die Verschiebung und damit auch über die Performanz hat, geht jedoch mit einer höheren Programmierkomplexität und damit Fehleranfälligkeit einher. Überdies wird die volle Kontrolle nicht notwendigerweise an jeder Stelle gebraucht und bläht damit den Quelltext unnötig auf. Die Adressräume bleiben bei der expliziten Datenbewegung getrennt (siehe Abbildung 2.1a)
Diesem Ansatz steht die *implizite* Datenbewegung gegenüber. Diese überlässt den Zeitpunkt und den konkreten Vorgang des Kopierens oder Verschiebens sowie den Umfang der bewegten Datenmenge der Laufzeitumgebung; aus Sicht des Programmierers existiert ein virtueller Adressraum, den sich CPU und GPU teilen (siehe Abbildung 2.1b).
Eine bewertende Gegenüberstellung beider Ansätze übersteigt den Rahmen dieser Arbeit (und ist bisher auch in der Literatur nicht zu finden). Stattdessen wird untersucht, in welcher Form beide Ansätze von den jeweiligen Spracherweiterungen unterstützt werden.

Datenanordnung Die Anordnung der Daten im Speicher kann aufgrund einer Reihe von Faktoren, wie etwa unterschiedlicher optimaler Zugriffsmuster von CPUs (*cachelines*) und GPUs (*strided access*), erheblich zur Performanz der Berechnung beitragen. Aus Sicht des Programmieres ist es daher wünschenswert, dass die Spracherweiterung eine abstrakte Sicht auf den von ihr verwendeten Speicher bietet, welche die optimalen Zugriffsmuster der verschiedenen Hardware kapselt.

[2] Mitglied des C++-Standard-Komitees, ehemaliger Projektleiter des XL-C++-Compilers der Firma IBM und derzeit Entwicklungsleiter des Unternehmens Codeplay, das unter anderem den SYCL-Standard implementiert



(a) Sicht auf den Adressraum bei expliziter Datenbewegung.



(b) Sicht auf den Adressraum bei impliziter Datenbewegung.

Abbildung 2.1: Verschiedene Adressraumsichten

Datenaffinität Die Datenaffinität definiert die Zuordnung und Nähe eines Speicherbereichs zu einer *bestimmten* Ausführungseinheit, die auf diesen Speicherbereich zugreifen muss. Im Hinblick auf GPUs meint dies die Zuordnung von im Hauptspeicher liegenden Daten auf eine oder mehrere GPUs.

Datenlokalität Die obigen Punkte sind alle eng mit dem Aspekt der Datenlokalität verknüpft. Im GPU-Kontext ist vornehmlich relevant, inwiefern sich die einzelnen Ebenen der Speicherhierarchie (globaler Speicher der GPU, lokaler Speicher und Cache der auf der GPU verbauten parallelen Prozessoren, Register der einzelnen Threads) durch die Spracherweiterungen ansteuern und nutzen lassen.

2.1.2 Aufgabensicht

Die im vorherigen Abschnitt genannten Punkte stellen die in dieser Arbeit verwendeten Vergleichskriterien dar, bedürfen jedoch noch einer Ergänzung:

Aufgabengraphen Die Abfolge und Abhängigkeiten einzelner *Kernel* lassen sich in Form eines Graphen darstellen (siehe Abbildung 2.2). Eine effiziente Ausnutzung der Parallelität voneinander unabhängiger Aufgaben setzt voraus, dass Kernel *asynchron* – sowohl im Hinblick auf die CPU als auch untereinander – und *parallel* auf der gleichen GPU ausgeführt werden können. Im Rahmen dieser Arbeit wird deshalb ebenfalls untersucht, welche Mittel die einzelnen Spracherweiterungen zur Verfügung stellen, um Aufgabengraphen zu implementieren.

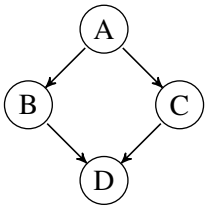


Abbildung 2.2: Beispielhafter Aufgabengraph. Der Kernel D hängt von den Kerneln B und C ab, die voneinander unabhängig sind, jedoch beide vom Kernel A abhängen.

2.2 CUDA

2.2.1 Einführung

Da der CUDA-Compiler `nvcc` ein C++-Compiler ist, lässt sich CUDA neben C++ auch in klassischen C-Programmen nutzen, sofern die Inkompatibilitäten zwischen C und C++ beachtet werden. Zu diesem Zweck ist das *application programming interface* (API) auf C-Konventionen beschränkt und unterstützt C++-Idiome nur sehr begrenzt. Die CUDA-Kernel selbst werden in einem C++-Dialekt geschrieben. (vgl. [cud18a], Abschnitt F)

CUDA-Kernel werden gemeinsam mit dem sie aufrufenden Quelltext kompiliert; CUDA folgt damit dem Modell der *single-source compilation*[3]. `nvcc` extrahiert dazu die Kernel und Device-Funktionen aus dem Quelltext und reicht den verbleibenden C- oder C++-Quelltext an den auf dem System befindlichen entsprechenden Compiler weiter. Der Device-Quelltext wird in einem weiteren Schritt vom tatsächlichen Device-Compiler `cicc` in die Maschinensprache der Ziel-GPU übersetzt.

Der Quelltext 2.1 zeigt einen Beispiel-Kernel in CUDA. Das Schlüsselwort `__global__` markiert einen Kernel, der vom Compiler extrahiert und für das Device übersetzt wird. Mit dem Schlüsselwort `__device__` werden Funktionen markiert, die innerhalb eines Kernels aufgerufen werden können, selbst jedoch keine eigenständigen Kernel darstellen. Die letzte Zeile zeigt den Aufruf des Kernels aus dem Host-Code heraus. Die hier sichtbaren Konzepte der Blocks und Threads werden in den nächsten Abschnitten ausführlich erklärt.

```
__device__ int foo() { /* ... */ }
__global__ void vec_add(const float* a, const float* b, float* c,
                        std::size_t dim)
{
    auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < dim) return;

    auto k = foo();
    c[i] = a[i] + b[i] + k;
}

vec_add<<<block_num, thread_num>>>(dev_a, dev_b, dev_c, dim);
```

Quelltext 2.1: Beispielkernel in CUDA

[3] CUDAs frühester Konkurrent OpenCL kompiliert die Device-Kernel dagegen erst zur Laufzeit des Programms, um zwischen verschiedenen Architekturen portabel zu bleiben (*split-source compilation*.)

2.2.2 Ausführungsmodell

Ein *thread* ist die kleinste selbstständige Einheit, die eine Aufgabe ausführen kann und direkt auf einen Hardware-Thread eines Multiprozessors abbildbar. *Threads* sind auf der Hardware-Ebene in *warps* – bestehend aus 32 *threads* – zusammengefasst, die synchron[4] auch unterhalb dieser Grenze dieselben Instruktionen ausführen. *Warps* entsprechen damit Registern der CPU, die das Modell *single-instruction, multiple data* (SIMD) unterstützen, sind im Vergleich zu diesen allerdings flexibler programmierbar. Dieses Hardware-Modell wird von NVIDIA auch als *single-instruction, multiple thread* (SIMT) bezeichnet. Die nächsthöhere Ebene bilden *blocks*, die aus bis zu 1.024 *threads* in drei Dimensionen bestehen können (das Produkt der *thread*-Zahl in x-, y- und z-Richtung darf 1.024 also nicht übersteigen). Während die Anzahl der *threads* in diesem Rahmen frei wählbar ist, bedingt die feste *warp*-Größe, dass eine *block*-Größe ein Vielfaches von 32 sein sollte. Andernfalls könnte der letzte *warp* jedes *blocks* nicht vollständig genutzt werden, was zu geringerer Parallelität auf *thread*-Ebene führen würde. Ein *block* wird genau einem Multiprozessor zugeordnet und von diesem ausgeführt. Sind mehr *blocks* als Multiprozessoren vorhanden, werden *blocks*, die z.B. durch Speicherzugriffe blockiert sind, vom Scheduler in einen wartenden Zustand versetzt und durch wartende, aber bereite *blocks* ersetzt. Dieser Kontextwechsel geschieht auf einer GPU bei einer geringen Anzahl von *blocks* pro Multiprozessor deutlich schneller als auf einer CPU und ist mit nur wenigen Zyklen verbunden, da die verwendeten Register und Caches nicht geleert und im Speicher hinterlegt werden müssen. Eine zu hohe Zahl von *blocks* pro Multiprozessor kann im Umkehrschluss einen gegenteiligen Effekt hervorrufen, durch den der Kontextwechsel nicht mehr derart performant ist.
Alle *threads* eines *blocks* können untereinander über den Speicher des Multiprozessors kommunizieren (*shared memory*). Barrieren ermöglichen die Synchronisation zwischen *warps*, während *threads* innerhalb eines *warps* durch spezielle Intrinsiken kommunizieren können.
Die Gesamtheit der *blocks* bildet das *grid*, das ebenfalls bis zu drei Dimensionen umfassen kann. Eine Synchronisierung zwischen *blocks* ist mit CUDA selbst nicht möglich, allerdings ermöglicht das im Zusammenhang mit CUDA 9 eingeführte (vgl. [cud18c], S. 3) API *cooperative groups* eine *grid*-weite Barriere. (vgl. [cud18a], Abschnitt 2.2)

2.2.3 Datensicht

Datenbewegung CUDA bietet mehrere Möglichkeiten der Datenbewegung an. Die *explizite* Datenbewegung wird seit der ersten CUDA-Version unterstützt und erfordert vom Programmierer die Allokation von Host- und Device-Speicher, das Auslösen des Kopiervorgangs der Daten vom Host auf das Device (und zurück) und das Freigeben des Speichers, wenn er nicht mehr benötigt wird (siehe Quelltext 2.2).

[4] Die neuen Architekturen seit Volta können mit weniger als 32 *threads* besser umgehen als die Vorgängerarchitekturen. Möglich wird dies durch ein Verfahren namens *independent thread scheduling*, das diese Hardware-Limitierung teilweise aufhebt.

```
auto a_h = new float[num_elems];
auto b_h = new float[num_elems];

/* a_h initialisieren */

auto a_d = static_cast<float*>(nullptr);
auto b_d = static_cast<float*>(nullptr);
cudaMalloc(&a_d, num_elems * sizeof(float));
cudaMalloc(&b_d, num_elems * sizeof(float));

cudaMemcpy(a_d, a_h, num_elems * sizeof(float),
           cudaMemcpyHostToDevice);

kernel<<<...>>>(a_d, b_d, num_elems);

cudaMemcpy(b_h, b_d, num_elems * sizeof(float),
           cudaMemcpyDeviceToHost);

cudaFree(b_d);
cudaFree(a_d);

delete[] b_h;
delete[] a_h;
```

Quelltext 2.2: Explizite Datenbewegung mit CUDA

Seit dem im Jahr 2011 erschienenen CUDA 4 unterstützt CUDA einen einheitlichen virtuellen Adressraum für Host und Device (vgl. [cud11], S. 4). Dieser ermöglichte noch keine implizite Datenbewegung, ersparte dem Programmierer aber die Angabe der Kopierrichtung beim Aufruf von `cudaMemcpy`. Der letzte Parameter des Befehls konnte seit diesem Zeitpunkt durch `cudaMemcpyDefault` ersetzt werden.

Die *implizite* Datenbewegung wird seit dem im Jahr 2014 erschienenen CUDA 6 unterstützt (vgl. [cud14], S. 3). Der Befehl `cudaMallocManaged` allokiert einen Speicherbereich, der sowohl vom Host als auch vom Device angesteuert werden kann. Die CUDA-Laufzeitumgebung sorgt dann im Hintergrund für das Kopieren der notwendigen Speicherbereiche. Es obliegt jedoch dem Programmierer, die Synchronisierung zwischen Host und Device auszuführen. Das in Quelltext 2.2 aufgeführte Beispiel wird dadurch zu der in Quelltext 2.3 gezeigten Vereinfachung.

```
auto a = static_cast<float*>(nullptr);
auto b = static_cast<float*>(nullptr);

cudaMallocManaged(&a, num_elems * sizeof(float));
cudaMallocManaged(&b, num_elems * sizeof(float));

kernel<<<...>>>(a, b, num_elems);

cudaDeviceSynchronize();

/* b ab hier auf dem Host nutzbar */

cudaFree(b);
cudaFree(a);
```

Quelltext 2.3: Implizite Datenbewegung ab CUDA 6

Im Jahr 2016 führte NVIDIA die Pascal-Architektur und CUDA 8 ein. Mit dieser neuen Architektur wurde die implizite Datenbewegung weiter vereinfacht, da es nun möglich war, gänzlich auf CUDA zur Speicherallokation zu verzichten: der Aufruf von **new** oder `malloc` genügt (vgl. [Har16]). Dadurch wurde auch die Verwendung von C++-Containern im Zusammenhang mit CUDA einfacher, deren Speicherbereich man nun direkt an CUDA-Kernel übergeben konnte (siehe Quelltext 2.4).

```
auto a = std::vector<float>{};
auto b = std::vector<float>{};

a.resize(num_elems);
b.resize(num_elems);

kernel<<<...>>>(a.data(), b.data(), num_elems);

cudaDeviceSynchronize();

/* b ab hier auf dem Host nutzbar */
```

Quelltext 2.4: Implizite Datenbewegung ab CUDA 8 und Pascal

Datenanordnung Die Anordnung der Daten im globalen Speicher der GPU entspricht bei der Allokation mit `cudaMalloc` und der Kopie mit `cudaMemcpy` der Anordnung der Daten im Host-Speicher. Es ist Aufgabe des Programmierers, die Effizienz der Anordnung und des Zugriffsverhaltens sicherzustellen (vgl. [cud18a], Abschnitt 5.3.2, Überschrift „Global Memory“).

Einen Sonderfall stellen zweidimensionale Arrays dar. Die Adresse A eines Felds eines 2D-Arrays mit der Startadresse S und der Breite B durch einen Thread mit den Koordinaten (x, y) berechnet sich in vielen Anwendungsfällen wie folgt:

$$A = S + y * B + x$$

Ein effizienter Zugriff dieser Form erfordert sowohl eine Block-Breite als auch eine Array-Breite, die ein mathematisches Vielfaches der Warpbreite sind. Mit `cudaMallocPitch` und `cudaMemcpy2D` kann man letzterer Anforderung Rechnung tragen. Eine Erweiterung dieses Prinzips auf dreidimensionale Arrays, die man sich auch als Array von 2D-Arrays vorstellen kann, ist mit `cudaMalloc3D` und `cudaMemcpy3D` möglich. (vgl. [cud18a], Abschnitt 5.3.2, Überschrift „Two-Dimensional Arrays“)
Sofern häufig auf benachbarte Spalten und Zeilen eines Arrays zugegriffen wird, lässt sich dies effizienter auslesen, wenn die eigentlich für Grafikoperationen – die häufig benachbarte Pixel manipulieren – vorgesehenen Textur-Caches der GPU genutzt werden. Eine für diese Caches optimale Speicheranordnung lässt sich mit `cudaMallocArray` bewerkstelligen. Ein auf diese Weise angelegtes `cudaArray` ist kein Zeiger, sondern eine für den Programmierer opake Datenstruktur, auf die nur über spezielle Texturbefehle zugegriffen werden kann. (vgl. [cud18a], Abschnitt 5.3.2, Überschrift „Texture and Surface Memory“)

Datenaffinität Die explizite Datenbewegung ordnet die kopierten Daten automatisch der *aktiven* GPU zu. Der CUDA-Kontext kennt generell nur ein aktives Device; will man in einem Multi-GPU-System auf eine andere GPU wechseln, geschieht dies durch den manuellen Wechsel mit dem Befehl `cudaSetDevice` (siehe Quelltext 2.5).
Dieses Verfahren gilt zum Teil auch für die implizite Datenbewegung. Speicher, der mit dem Befehl `cudaMallocManaged` reserviert wurde, wird bei allen NVIDIA-GPU-Architekturen bis einschließlich *Maxwell* nur zwischen dem Host und dem zum Zeitpunkt der Reservierung aktiven Device ausgetauscht. Ab der *Kepler*-Architektur wird der Speicher dagegen auf allen GPUs sichtbar und nach Bedarf auf die GPU kopiert, welche die Daten benötigt. (vgl. [cud18a], Abschnitt K.1.5)

```
auto count = 0;
cudaGetDeviceCount(&count); // Zahl der GPUs abfragen

cudaSetDevice(0); // Device 0 aktiv
cudaMemcpy(..., cudaMemcpyHostToDevice); // kopieren auf Device 0

cudaSetDevice(1); // Device 1 aktiv
cudaMemcpy(..., cudaMemcpyHostToDevice); // kopieren auf Device 1
```

Quelltext 2.5: Setzen der aktiven GPU mit CUDA

Datenlokalität CUDA kennt sieben Ebenen innerhalb der GPU-Speicherhierarchie (vgl. [cud18a], Abschnitt 5.3.2).
Global memory ist der Hauptspeicher der GPU und für alle Multiprozessoren sichtbar. Die Zugriffe sind jedoch zeitaufwendig.
Zugriffe auf den *global memory* erfolgen über L1- (pro Multiprozessor) und L2-Caches (alle Multiprozessoren). Die Caches lassen sich vom Programmierer nicht direkt ansteuern.

Constant memory liegt im *global memory*, wird aber auf eine spezielle Weise in den Cache geladen. Er ermöglicht deutlich schnellere Speicherzugriffe, wenn alle *threads* eines *warps* gleichzeitig auf dasselbe Datum zugreifen.
Operationen, die die Textureinheiten der GPU verwenden, können über die Textur-Caches der GPU lesend auf den *global memory* zugreifen. Diese Caches sind für häufige zweidimensionale Speicherzugriffe auf benachbarte Zeilen und Spalten optimiert. Sie sind vom Programmierer mittels der Intrinsik `__ldg` direkt ansprechbar oder können implizit durch die in CUDA enthaltenen Texturbefehle genutzt werden.
Shared memory ist ein spezieller L1-Cache, der direkt vom Programmierer lesbar und beschreibbar ist. Der *shared memory* ist in seiner Kapazität sehr eingeschränkt (er umfasst nur einige KiB), ermöglicht dem Programmierer aber sehr schnelle Zugriffe sowie die Kommunikation zwischen *threads* eines *blocks*.
Die Register des Multiprozessors werden *threads* nach Bedarf zugeordnet und sind vom Programmierer nicht direkt ansprechbar.
Local memory ist *thread*-lokaler *global memory*. In diesem werden lokale Variablen, die nicht mehr in die Register passen (weil deren Anzahl erschöpft ist), oder Arrays unbekannter Länge automatisch untergebracht. Da der *local memory* versteckt im *global memory* liegt, ist seine Nutzung aufgrund der langen Zugriffszeiten üblicherweise unerwünscht.

2.2.4 Aufgabensicht

Aufgabengraphen Aufgabengraphen lassen sich in CUDA auf zwei Arten implementieren. Der herkömmliche Weg ist die Verwendung von *streams*. Diese können als Warteschlangen für Aufgaben (Speicherbewegungen, Kernel-Ausführungen) gesehen werden und sind untereinander und in Bezug auf den Host asynchron. Das heißt, dass verschiedene Kernel in getrennten *streams* parallel auf der GPU ausgeführt werden können (entsprechende Hardware-Ressourcen vorausgesetzt). Innerhalb eines *streams* sind alle Aufgaben immer serialisiert, zwei Kernel im selben *stream* werden also immer nacheinander ausgeführt.
Die Kommunikation bzw. Synchronisation zwischen *streams* erfolgt über *events*. *Events* werden z.B. durch das Beenden eines Kernels ausgelöst. Ist ein Kernel eines anderen *streams* von diesem abhängig, lässt sich durch das Warten auf dieses *event* der *stream* blockieren, bis die Ausführung fortgesetzt werden kann.
Der Quelltext 2.6 zeigt die Implementierung des Beispiel-Graphen in Abbildung 2.2 mit *streams*.

```
A<<<..., stream1>>>();
// eventA wird ausgelöst, sobald A fertig ist
cudaEventRecord(eventA, stream1);

B<<<..., stream1>>>(); // wartet automatisch auf A

// stream2 wartet auf eventA
cudaStreamWaitEvent(stream2, eventA);
// C kann jetzt ausgeführt werden
C<<<..., stream2>>>();
// eventC wird ausgelöst, sobald C fertig ist
cudaEventRecord(eventC, stream2);

// stream1 wartet auf eventC
cudaStreamEvent(stream1, eventC);
D<<<..., stream1>>>(); // wartet automatisch auf B
```

Quelltext 2.6: Aufgabengraph mit *streams*

Der *stream*-Ansatz hat den Nachteil, dass bei iterativen Verfahren, die das mehrfache Ausführen des Graphen erfordern, durch das ständige Starten der Kernel und das manuelle Blockieren der *streams* viel Overhead erzeugt wird. Die mit dem 2018 veröffentlichten CUDA 10 eingeführten *CUDA Graphs* (vgl. [cud18b], S. 4) sollen dieses Problem lösen (vgl. [Ram18] für eine Einführung).
Aufgabengraphen lassen sich mit diesem API direkt implementieren. Quelltext 2.7 zeigt die Abbildung 2.2 entsprechende Implementierung.

```
cudaGraphCreate(&graph);

// Abhängigkeiten definieren
cudaGraphAddNode(graph, A, {}, ...);
cudaGraphAddNode(graph, B, {A}, ...);
cudaGraphAddNode(graph, C, {A}, ...);
cudaGraphAddNode(graph, D, {B, C}, ...);

// Graph-Instanz erzeugen
cudaGraphInstantiate(&instance, graph);

// Graph-Instanz auf stream ausführen
cudaGraphLaunch(instance, stream);
```

Quelltext 2.7: Aufgabengraph mit *CUDA Graphs*

2.3 HIP

2.3.1 Einführung

AMDs HIP-API ist als portable Alternative zu CUDA gedacht und de facto ein Klon des CUDA-API, bei dem fast jeder CUDA-Befehl `cudaCmd` seine genaue Entsprechung im HIP-Befehl `hipCmd` fin-

det. Tatsächlich werden HIP-Quelltexte, die für eine NVIDIA-GPU kompiliert werden sollen, wieder in CUDA-Quelltexte umgewandelt, da die HIP-Befehle die äquivalenten CUDA-Befehle ohne Umwege aufrufen.
Die in den vorherigen Abschnitten für CUDA dargestellten Prinzipien gelten daher weitestgehend auch für HIP; vorhandene Unterschiede werden in den folgenden Abschnitten aufgeführt.
HIP für AMD-GPUs ist eine dünne Schicht über AMDs eigentlichem GPGPU-API, HC, das in Abschnitt 2.4 vorgestellt wird.
Der Quelltext 2.8 zeigt einen in HIP geschriebenen Beispielkernel und den zugehörigen Aufruf innerhalb des Host-Codes. Die Block- und Threadkonzepte entsprechen den in CUDA vorhandenen.

```
__device__ int foo() { /* ... */ }

__global__ void vec_add(const float* a, const float* b, float* c,
                        std::size_t dim)
{
    auto i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if(i < dim) return;

    auto k = foo();
    c[i] = a[i] + b[i] + k;
}

hipLaunchKernelGGL(vec_add, block_num, thread_num, 0, 0,
                   dev_a, dev_b, dev_c, dim);
```

Quelltext 2.8: Beispielkernel in HIP

2.3.2 Datensicht

Datenbewegung Die implizite Datenbewegung, die mit `cudaMallocManaged` möglich ist, hat gegenwärtig kein HIP-Äquivalent. Programmierer haben daher nur die Möglichkeit der expliziten Datenbewegung.

Datenanordnung Zweidimensionaler Speicher kann auf CUDA-GPUs mit `cudaMallocPitch` in einer optimierten Anordnung reserviert werden. Dieser Befehl wird derzeit nicht in HIP abgebildet. Gleiches gilt für `cudaArrays`, die in ihrer Anordnung für den Textur-Cache optimiert sind.

Datenlokalität Die Textur-Einheiten und damit der Textur-Cache der GPU werden von HIP zur Zeit nur eingeschränkt unterstützt.

2.3.3 Aufgabensicht

Aufgabengraphen Aufgabengraphen lassen sich in HIP nur über *streams* implementieren, *CUDA Graphs* werden momentan nicht unterstützt.

2.4 HC

2.4.1 Einführung

AMDs HC-API ist ein Dialekt des C++AMP-API, das im August 2012 von der Firma Microsoft veröffentlicht wurde. C++AMP setzte auf der Grafikschnittstelle DirectX 11 auf und ermöglichte die direkte Programmierung von GPUs und CPU-SIMD-Registern mittels C++. 2013 übernahm AMD C++AMP für sein HSA-Programm und erweiterte das API um AMD-spezifische Funktionen und GPU-relevante Features.
HC-Kernel können entweder als separate Funktionen oder Funktoren geschrieben oder als Lambda-Funktionen direkt in den C++-Quelltext eingebettet werden. Der Compiler extrahiert die Kernel aus dem sie umgebenden Quelltext und übersetzt sie in den GPU-Maschinencode, während der restliche Quelltext in den CPU-Maschinencode umgewandelt wird. Der komplette Quelltext wird also in einem Schritt kompiliert.
Unglücklicherweise ist die Dokumentationslage des HC-API sehr schlecht. Es gibt von AMD keine öffentlich zugänglichen Dokumente, die der CUDA-Dokumentation in Umfang und Inhalt nahe kämen. Die in den folgenden Abschnitten präsentierten Erkenntnisse basieren auf der unvollständigen HC-API-Referenz, die ein AMD-Angestellter auf seiner privaten GitHub-Seite zur Verfügung stellt (vgl. [hcr17]), der C++AMP-Spezifikation aus dem Jahr 2012[5] (vgl. [cpp12]) und
Der Quelltext 2.9 zeigt einen in HC geschriebenen Beispielkernel. Device-Funktionen und Kernel werden beide mit dem Attribut `[[hc]]` markiert. Die letzten Zeilen zeigen den Aufruf des Kernels innerhalb des Host-Codes. Auch hier werden die Konzepte in den nächsten Abschnitten detailliert erklärt.

```
int foo() [[hc]] { /* ... */ }

void vec_add [[hc]] (hc::tiled_index<1> idx,
                     hc::array_view<const float> a,
                     hc::array_view<const float> b,
                     hc::array_view<float> c, std::size_t dim)
{
    auto i = idx.global[0];
    if(i < dim) return;

    auto k = foo();
    c[i] = a[i] + b[i] + k;
}

auto global_extent = hc::extent<1>{global_thread_num};
hc::parallel_for_each(global_extent.tile(tile_size),
[=](hc::tiled_index<1> idx)
{
    vec_add(idx, a_view, b_view, c_view, dim);
});
```

Quelltext 2.9: Beispielkernel in HC

[5] Die Features der Ende 2013 veröffentlichten Spezifikation für C++AMP 1.2 werden von HC nicht unterstützt.

2.4.2 Ausführungsmodell

Das HC-Ausführungsmodell ähnelt dem von CUDA sehr stark und unterscheidet sich nur in einigen Details. Auch hier ist die kleinste Ausführungseinheit ein *thread*, die direkt auf einen Hardware-Thread eines Multiprozessors abgebildet wird. Eine Gruppe von 64 *Threads* bildet eine *wavefront* (das Gegenstück zu CUDA-*warps*) und arbeitet synchron die selben Instruktionen ab.
Auf der nächsthöheren Organisationsebene befinden sich *tiles*, die den CUDA-*blocks* entsprechen und ebenfalls aus insgesamt 1.024 *threads* bestehen können. Auch bei HC ist die Größe der *tiles* bis zu dieser Grenze frei wählbar, wird aufgrund der *wavefront*-Größe in der Regel jedoch ein Vielfaches von 64 sein. Jede *tile* wird einem Multiprozessor zugeordnet und von diesem ausgeführt, das Scheduling-Verhalten der GPU bei überzähligen *tiles* entspricht dem von CUDA.
Die *threads* einer *tile* teilen sich den Speicher eines Multiprozessors, der *tile static memory* genannt wird und CUDAs *shared memory* entspricht. Barrieren ermöglichen die Synchronisation zwischen *wavefronts*, zudem gibt es Hardware-Intrinsiken für die Kommunikation innerhalb einer *wavefront*.
CUDAs *grid*-Konzept hat keine direkte HC-Entsprechung, da bei letzterem die Menge der *tiles* keine große Rolle spielt. Stattdessen ist die globale Problemgröße von Relevanz. Diese muss dabei nicht zwangsläufig der Gesamtzahl der *threads* entsprechen. Der Programmierer hat dadurch die Möglichkeit, das Problem mit *tiles* von selbst definierter Größe zu bearbeiten oder nur die globale Problemgröße anzugeben und die Festlegung der *threads* und *tiles* der Laufzeitumgebung zu überlassen.

2.4.3 Datensicht

Datenbewegung Die *explizite* Datenbewegung geschieht über das Definieren des Host-Speichers und eines GPU-Arrays sowie den anschließenden Aufruf des `copy`-Befehls (siehe Quelltext 2.10). Eine manuelle Freigabe des Speichers ist nicht notwendig, da dies automatisch im Destruktor des GPU-Arrays geschieht, sobald der umschließende Kontext verlassen wird.

```
auto a_h = std::vector<float>{};
auto b_h = std::vector<float>{};

a_h.resize(num_elems);
b_h.resize(num_elems);

/* a_h initialisieren */

auto a_d = hc::array<float, 1>{hc::extent<1>{num_elems}};
auto b_d = hc::array<float, 1>{hc::extent<1>{num_elems}};

hc::copy(std::begin(a_h), a_d);

hc::parallel_for_each( /* kernel */ );

hc::copy(b_d, std::begin(b_h))
```

Quelltext 2.10: Explizite Datenbewegung mit HC

Einen Mittelweg zwischen impliziter und expliziter Datenbewegung stellt die direkte Zuordnung eines `hc::array` zu einem Datenbereich auf dem Host dar. Die Host-Daten werden vollständig in das `hc::array` kopiert. Wenn das System, auf dem das Programm ausgeführt wird, einen einheitlichen virtuellen Speicherbereich unterstützt, ist das `hc::array` anschließend auch vom Host aus ansprechbar (siehe Quelltext 2.11).

```
auto a_h = std::vector<float>{};

a_h.resize(num_elems);

/* a_h initialisieren */

auto a_d = hc::array<float, 1>{hc::extent<1>{num_elems},
                               std::begin(a_h)};
auto b_d = hc::array<float, 1>{hc::extent<1>{num_elems},
                               std::begin(b_h)};

hc::parallel_for_each( /* kernel */ );

// wenn vom System unterstützt
func(b_d[42]);
```

Quelltext 2.11: Implizite Datenbewegung mit HC-Arrays

Mit der `hc::array_view`-Struktur ist eine weitere Möglichkeit der *impliziten* Datenbewegung gegeben. `hc::array_view` lässt sich sowohl für ein `hc::array` als auch einen Standard-C++-Container konstruieren. Die Bewegung der auf der jeweils anderen Seite benötigten Daten geschieht im Verborgenen und nur für die tatsächlich angeforderten Teile, nicht das gesamte Feld. Der Zugriff auf die Daten erfolgt dabei über die Methoden der `hc::array_view`, die einen internen Zwischenspeicher besitzt. Ein Zugriff auf das original `hc::array` bzw. den originalen C++-Container bedarf der vorherigen Synchronisierung (siehe Quelltext 2.12). Darüber hinaus ist zu bemerken, dass das ausführende System keinen einheitlichen virtuellen Speicherbereich unterstützen muss, um `hc::array_view` nutzen zu können.

```
auto a_h = std::vector<float>{};
a_h.resize(num_elems);

auto b_d = hc::array<float, 1>{hc::extent<1>{num_elems}};

/* a_h initialisieren */

auto a_view = hc::array_view<float, 1>{hc::extent<1>{num_elems}, a_h};
auto b_view = hc::array_view<float, 1>{b_d};

a_h[42] = ...;     // a_h schreiben
a_view.refresh(); // a_view synchronisieren

hc::parallel_for_each( /* kernel */ );

// a_view und b_view auf Host-Seite schreiben / lesen
for(auto i = 0; i < num_elems; ++i)
    a_view[hc::index<1>{i}] = b_view[hc::index<1>{i}];

a_view.synchronize(); // a_h synchronisieren
... = a_h[42]; // a_h enthält jetzt die neuen Daten
```

Quelltext 2.12: Implizite Datenbewegung mit HC-Sichten

Datenanordnung `hc::array` und `hc::array_view` können bis zu drei Dimensionen abdecken. Die interne Anordnung der Daten wird dabei vor dem Programmierer verborgen. Dies wird einerseits daran deutlich, dass der Zugriff auf ein Feld über den `[]`-Operator nicht durch ganze Zahlen erfolgt, sondern mittels der speziellen Struktur `hc::index`, die ebenfalls bis zu drei Dimensionen umfassen kann. Andererseits gibt es zwei Methoden, die den gekapselten Zeiger zurückliefern:

- `hc::array::accelerator_pointer()` und das `hc::array_view`-Gegenstück geben einen Zeiger auf den zugehörigen Device-Speicher zurück.
- `hc::array::data()` bzw. das `hc::array_view`-Äquivalent liefern einen Zeiger auf ein *linearisiertes* Array.

Aus den obigen Beobachtungen lässt sich daher folgern, dass die Anordnung der Daten im GPU-Speicher nicht unbedingt der Anordnung im Hauptspeicher entsprechen muss und sich unter Umständen auch zwischen verschiedenen HC-Versionen oder GPUs unterscheiden kann.

Datenaffinität Ein `hc::array` ist immer an ein bestimmtes Device gebunden. Sofern der Programmierer bei der Konstruktion eines `hc::arrays` in einem Multi-GPU-System kein Device angibt, entscheidet die Laufzeitumgebung, auf welcher GPU der Speicher reserviert wird. Über den `hc::copy`-Befehl lassen sich Daten direkt zwischen `hc::arrays` auf verschiedenen GPUs bewegen (siehe Quelltext 2.13).

```
auto accelerators = hc::accelerator::get_all();

// Devices aussuchen
auto acc0 = ...;
auto acc1 = ...;

auto acc0_view = acc0.get_default_view();
auto acc1_view = acc1.get_default_view();

auto a_h = std::vector<float>{};
auto b_h = std::vector<float>{};
a_h.resize(num_elems);
b_h.resize(num_elems);

/* a_h und b_h initialisieren */

// a_d auf Device #0 anlegen
auto a_d = hc::array<float, 1>{hc::extent<1>{num_elems},
                               std::begin(a_h), acc0_view};

// b_d auf Device #1 anlegen
auto b_d = hc::array<float, 1>{hc::extent<1>{num_elems},
                               std::begin(b_h), acc1_view};

// Inhalt von a_d zu b_d kopieren
hc::copy(a_d, b_d);
```

Quelltext 2.13: GPU-Speicheraffinität mit HC-Arrays

`hc::array_view` wird dagegen keiner GPU direkt zugeordnet. Bei einem Zugriff auf den auf einer GPU liegenden und von `hc::array_view` verwalteten Speicher durch eine andere GPU wird der Prozess der Datenbewegung grundsätzlich verborgen. Zu einem von ihm gewählten Zeitpunkt kann der Programmierer den Vorgang jedoch durch den Befehl `hc::array_view::synchronize_to()` auslösen.

Datenlokalität HC unterscheidet lediglich zwei Ebenen innerhalb der Speicherhierarchie:
Der *global memory* ist der auf dem Device vorhandene, allen Multiprozessoren zugängliche Speicher. Er verfügt über ein großes Fassungsvermögen, benötigt jedoch viel Zeit für Speicherzugriffe.
Die Multiprozessoren verfügen über programmierbare Caches, die von ihren Hardware-Threads genutzt werden können. Dieser *tile static memory* genannte Speicher ist deutlich kleiner als der globale GPU-Speicher, kann jedoch viel schneller angesteuert werden.
Obwohl HC auf C++AMP basiert und hauptsächlich für GPUs und APUs gedacht ist, werden die in C++AMP enthaltenen (optionalen) Textur-Features nicht unterstützt.

2.4.4 Aufgabensicht

Aufgabengraphen HC-Kernel werden an Datenstrukturen vom Typ `hc::accelerator_view` übermittelt. Wie der Name impliziert, handelt es sich bei diesen Strukturen um *logische* Sichten auf einen Beschleuniger. Der Programmierer kann mehrere `hc::accelerator_views` pro GPU erzeugen.
Alle Kernel, die einer bestimmten `hc::accelerator_view` zugeordnet werden, werden in der Reihenfolge ihrer Übermittlung ausgeführt. Kernel, die zu verschiedenen Sichten gehören, können dagegen in anderer Reihenfolge oder parallel ausgeführt werden. Die Synchronisation zwischen von einander abhängigen Kerneln, die verschiedenen `hc::accelerator_views` übermittelt wurden, erfolgt durch `hc::completion_future`-Objekte. Diese entsprechen in ihrem Verhalten weitestgehend den `std::future`-Objekten der C++-Standardbibliothek, wurden jedoch um die Methode `then` erweitert, die eine Ausführung eines nachfolgenden Befehls ermöglicht, sobald der aktuelle Kernel beendet ist. Dieser Ansatz hat jedoch einen Nachteil: `then` kann nur ein einziges Mal pro `completion_future` ausgeführt werden, eine Verkettung oder Parallelisierung mehrerer nachfolgender Befehle gestaltet sich also recht schwierig.
Der Quelltext 2.14 zeigt die Implementierung des Beispiel-Graphen in Abbildung 2.2 mit HC.
Eine auf Graphen spezialisierte Schnittstelle existiert nicht.

```
auto accelerators = hc::accelerator::get_all();

auto acc = ...;
auto view0 = acc.create_view();
auto view1 = acc.create_view();

// A ausführen
auto futureA = hc::parallel_for_each(view0, /* kernel A */);

// B wird nach A ausgeführt
auto futureB = hc::parallel_for_each(view0, /* kernel B */);

// C ausführen, sobald A fertig ist
auto futureC = hc::completion_future{};
futureA.then([&]()
{
    futureC = hc::parallel_for_each(view1, /* kernel C */);
});

// D wird nach B ausgeführt, sobald C fertig ist
futureC.then([&]()
{
    hc::parallel_for_each(view0, /* kernel D */);
});
```

Quelltext 2.14: Aufgabengraph mit HC

2.5 SYCL

2.5.1 Einführung

Khronos' SYCL-Standard ging aus einer Compiler-Erweiterung der Firma Codeplay hervor, die für den Einsatz auf PlayStation-3-Spielekonsolen konzipiert wurde. Diese Erweiterung ermöglichte die Markierung von C++-Funktionen mit Compiler-Direktiven, die dadurch automatisch auf den in der PlayStation 3 enthaltenen Beschleunigern ausgeführt wurden. (vgl. [LPD16])
SYCL ist syntaktisch vollständig mit dem C++-Standard kompatibel. Die Kernel bedürfen also keiner speziellen Annotation, wie das etwa in CUDA mit der Markierung `__global__` bzw. bei HC mit der Markierung `[[hc]]` der Fall ist. Kernel werden als Lambda-Funktionen oder als Funktoren geschrieben und können weitere Funktionen aufrufen. Es ist die Aufgabe des Compilers, Kernel und aufgerufen Funktionen zu erkennen und für das Ziel-Device zu übersetzen.
Um die Portabilität zu gewährleisten, wird der Quelltext in mehreren Phasen übersetzt. Der Quelltext für den Host wird einmal vom System-Compiler kompiliert, der Device-spezifische Code einmal für jedes Ziel-Device durch den jeweiligen Device-Compiler. Code, der sowohl auf AMD als auch NVIDIA-GPUs zur Ausführung gebracht werden soll, benötigt also einen Device-Compiler, der den AMD-GPU-Befehlssatz beherrscht und zusätzlich einen weiteren Device-Compiler, der den Befehlssatz für NVIDIA-GPUs kennt. Der so generierte Maschinen-Code wird in die ausführbare Datei eingebettet und zur Laufzeit ausgewählt. (vgl. [syc18], S. 262)
SYCL kann ebenfalls OpenCL-Kernel verarbeiten, die dann – wie bei dem OpenCL-API – zur Laufzeit kompiliert werden.
Zur Zeit existieren drei SYCL-Implementierungen, die bereits (auf jeweils anderer Hardware) nutzbar sind, sich jedoch noch stark in der Entwicklung befinden:
Codeplays Implementierung *ComputeCpp* ist sowohl in einer kommerziellen Variante für den Embedded- und Automotive-Bereich als auch in einer kostenlosen Version für Intel-CPUs sowie Intel- und NVIDIA-GPUs verfügbar. OpenCL-Implementierungen, die Khronos' *intermediate representations*[6] SPIR oder SPIR-V unterstützen, können ebenfalls mit *ComputeCpp* angesprochen werden. (vgl. [com19])
Die von der Firma Xilinx vorangetriebene Referenzimplementierung *triSYCL* kann SYCL-Kernel auf CPUs verschiedener Hersteller (über OpenMP) und Xilinx' eigenen FPGAs ausführen. (vgl. [tri18])
Intel arbeitet gegenwärtig daran, SYCL-Unterstützung direkt in den clang-Compiler einzubauen. Diese namenlose[7] Implementierung kann mit OpenCL-2.1-Implementierungen zusammenarbeiten, die derzeit nur für manche Intel-CPUs und -GPUs vorhanden sind. (vgl. [int19])
Der Quelltext 2.15 zeigt das Vektoradditions-Beispiel in SYCL. Auffällig ist, dass Device-Funktionen und Kernel im Gegensatz zu den anderen Spracherweiterungen keine speziellen Annotationen benötigen. Die nächsten Abschnitte erläutern die verwendeten Konzepte genauer.

[6] Eine abstrakte Maschinensprache, die von Compilern vor der Übersetzung in die eigentliche Maschinensprache verwendet wird. Das bekannteste Beispiel ist die *Low Level Virtual Machine*, besser bekannt als LLVM.
[7] Der offizielle Titel der Implementierung lautet *SYCL Compiler and Runtimes*.

```
int foo() { /* ... */ }

struct vec_adder
{
    cl::sycl::accessor<float, 1,
                       cl::sycl::access::mode::read,
                       cl::sycl::access::target::global_buffer> a;
    cl::sycl::accessor<float, 1,
                       cl::sycl::access::mode::read,
                       cl::sycl::access::target::global_buffer> b;
    cl::sycl::accessor<float, 1,
                       cl::sycl::access::mode::discard_write,
                       cl::sycl::access::target::global_buffer> c;
    std::size_t dim;

    void operator()(cl::sycl::nd_item<1> my_item)
    {
        auto i = my_item.get_global_id(0);
        if(i >= dim) return;

        auto k = foo();
        c[i] = a[i] + b[i] + k;
    }
};

queue.submit(cl::sycl::handler& cgh)
{
    auto acc_a = buf_a.get_access<cl::sycl::access::mode::read,
                                  cl::sycl::target::global_buffer>();
    auto acc_b = buf_b.get_access<cl::sycl::access::mode::read,
                                  cl::sycl::target::global_buffer>();
    auto acc_c =
    ↪  buf_c.get_access<cl::sycl::access::mode::discard_write,
                                  cl::sycl::target::global_buffer>();

    auto adder = vec_adder { acc_a, acc_b, acc_c, dim };

    cgh.parallel_for(cl::sycl::nd_range<1>{
                        cl::sycl::range<1>{global_thread_num},
                        cl::sycl::range<1>{group_size}},
                     adder);
}
```

Quelltext 2.15: Beispielkernel in SYCL

2.5.2 Ausführungsmodell

SYCL übernimmt OpenCLs Ausführungsmodell ohne weitere Anpassungen. Da es sich hier nicht um ein reines GPU-API handelt und andere Beschleuniger ebenfalls berücksichtigt werden, wird bei SYCL und OpenCL von GPU-spezifischen Konzepten wie *threads* abstrahiert.
Führt ein Beschleuniger einen SYCL-Kernel aus, wird ein Index-Raum definiert, der bis zu drei Dimensionen umfassen kann. Für jeden Punkt in diesem Raum wird der Kernel einmal ausgeführt. Diese Kernel-Instanz heißt *work-item* und entspricht konzeptionell den *threads* der anderen Spracherweiterungen. (vgl. [syc18], S. 23)
Work-items werden in *work-groups* gesammelt, der Entsprechung zu den *groups* bzw. *tiles* der anderen Spracherweiterungen. Die *work-items* einer *work-group* werden parallel vom Beschleuniger ausgeführt, während verschiedene *work-groups* auch seriell ausgeführt werden können. (vgl. [syc18], S. 23)
Work-items der selben *Work-group* können untereinander über den *local memory* kommunizieren, der CUDAs *shared memory* bzw. HCs *tile static memory* entspricht. Die dabei notwendige Synchronisierung erfolgt durch verschiedene Barrierentypen.
Aufgrund des abstrakten Hardware-Modells gibt es kein Äquivalent zu *warps* und *wavefronts* und somit auch keine standardisierten Intrinsiken oder Befehle, die die Kommunikation innerhalb dieser Hardware-spezifischen Threadgruppen ermöglichen. Eine Nutzung von herstellerspezifischen Erweiterungen (wie bei OpenCL) ist für SYCL-Kernel nicht vorgesehen, hier ist nur der „Umweg“ über OpenCL-Kernel möglich. Das Schreiben hochperformanter Kernel setzt jedoch gelegentlich die Nutzung solcher Intrinsiken voraus, weshalb SYCL gegenüber den anderen Spracherweiterungen im Nachteil ist.

2.5.3 Datensicht

Datenbewegung Die von der SYCL-Spezifikation vorgesehene Datenbewegung ist grundsätzlich *implizit*. Dabei kommt ein zweistufiges System zum Einsatz. Zunächst wird der zu bearbeitende Speicherbereich des Hosts einem *buffer* zugeordnet. Der *buffer* übernimmt während seiner Lebensdauer die Kontrolle über besagten Host-Speicher und reserviert einen entsprechenden Bereich auf dem Device (bzw. den Devices, siehe Affinitäts-Abschnitt). Der ursprüngliche Host-Container bzw. -Zeiger ist währenddessen in einem nicht definierten Zustand. Mitunter kann es notwendig sein, schon während der Lebensdauer des *buffer*s den eigentlichen Host-Container zu verändern oder zu lesen. In diesem Fall ermöglicht das SYCL-API die manuelle Synchronisierung mittels *mutual exclusions*. (vgl. [syc18], Abschnitt 4.7.2)
Der Zugriff auf den so verwalteten Speicher geschieht sowohl auf Host- als auch auf Device-Seite über *accessor*-Instanzen, die über die `get_access`-Methode des *buffer*s akquiriert werden. Dabei muss der Programmierer angeben, ob der Zugriff lesend, schreibend (mit oder ohne Beibehalten der vorherigen Daten) oder sowohl lesend als auch schreibend erfolgen soll. Dies ermöglicht der SYCL-Laufzeitumgebung Optimierungen und die Abhängigkeitserkennung, die für Aufgabengraphen nötig ist (siehe Abschnitt 2.5.4). (vgl. [syc18], Abschnitt 4.7.6)
Sobald der Destruktor des *buffer*s erreicht wird, werden die verwalteten Daten wieder mit dem ursprünglichen Host-Container synchronisiert. Der Quelltext 2.16 zeigt die implizite Datenbewegung mit SYCL.

```
struct some_kernel
{
    cl::sycl::accessor<float, 1,
                       cl::sycl::access::mode::read_only> a;
    cl::sycl::accessor<float, 1,
                       cl::sycl::access::mode::discard_write> b;

    void operator()(cl::sycl::nd_item<1> work_item)
    { /* kernel */ }
};

auto a_h = std::vector<float>{};
auto b_h = std::vector<float>{};

a_h.resize(num_elems);
b_h.resize(num_elems);

/* a_h initialisieren */

/* neuer Scope - SYCL-Objekte existieren nur innerhalb der
   geschweiften Klammern */
{
    auto a_d = cl::sycl::buffer<float>{a_h.data(),
                                       cl::sycl::range<1>{num_elems}};
    auto b_d = cl::sycl::buffer<float>{b_h.data(),
                                       cl::sycl::range<1>{num_elems}};
    // a_h und b_h sind ab hier undefiniert

    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto acc_a = a_d.get_access<
                        cl::sycl::access::mode::read_only>();
        auto acc_b = b_d.get_access<
                        cl::sycl::access::mode::discard_write>();

        auto kernel = some_kernel{acc_a, acc_b};
        cgh.parallel_for(cl::sycl::nd_range<1>{
                            cl::sycl::range<1>{global_size},
                            cl::sycl::range<1>{group_size}},
                         kernel);
    });
    // a_h und b_h sind immer noch undefiniert
} // Destruktoren von a_d und b_d werden hier aufgerufen
// b_h enthält ab hier die neuen Daten
```

Quelltext 2.16: Implizite Datenbewegung mit SYCL

SYCL unterstützt ebenfalls die *explizite* Datenbewegung. Dafür stehen zwei Wege bereit.
Der erste Weg ähnelt stark der impliziten Datenbewegung. Hier wird lediglich ein anderer Konstruktor der Klasse `buffer` benutzt, der als Parameter keinen Zeiger auf einen Speicherbereich des Hosts erhält, sondern einen Ein- und einen Ausgabeiterator eines C++-Containers. Im obigen Beispiel könnte man diese beispielsweise durch die Funktionen `std::begin` und `std::end` der Standardbibliothek abfragen. (vgl. [syc18], Abschnitt 4.7.2.3)
Der zweite Weg stellt mit der Methode `copy` der Klasse `handler` einen expliziteren Befehl bereit. Aus Sicht des Programmierers handelt es sich dabei um einen speziellen Kernel, der von der SYCL-Laufzeitumgebung dann in die entsprechenden Kopierfunktionen des OpenCL-API umgewandelt wird. Wie bei der impliziten Datenbewegung erfolgt der Datenzugriff über die `accessor`-Klasse; der Quelltext 2.17 illustriert den Vorgang. (vgl. [syc18], Abschnitt 4.8.6)

```
auto a_h = std::vector<float>{};
auto b_h = std::vector<float>{};

a_h.resize(num_elems);
b_h.resize(num_elems);

/* a_h initialisieren */

auto a_d = cl::sycl::buffer<float>{cl::sycl::range<1>{num_elems}};
auto b_d = cl::sycl::buffer<float>{cl::sycl::range<1>{num_elems}};

queue.submit([&](cl::sycl::handler& cgh)
{
    auto acc_a = a_d.get_access<
                    cl::sycl::access::mode::discard_write>();
    cgh.copy(a_h.data(), acc_a);
});

queue.submit([&](cl::sycl::handler& cgh)
{
    auto acc_a = a_d.get_access<cl::sycl::access::mode::read_only>();
    auto acc_b = b_d.get_access<
                    cl::sycl::access::mode::discard_write>();

    /* kernel - wie im impliziten Beispiel */
});

queue.submit([&](cl::sycl::handler& cgh)
{
    auto acc_b = b_d.get_access<cl::sycl::access::read_only>();
    cgh.copy(acc_b, b_h.data());
});
```

Quelltext 2.17: Explizite Datenbewegung mit SYCL - copy-Befehl

Datenanordnung Die SYCL-Spezifikation macht keine Angaben darüber, wie der den *buffer*n zugrundeliegende Speicher angeordnet ist. Stattdessen werden Device-seitige Zeiger-Klassen, die sich über den *accessor* anfordern lassen, explizit als *implementation defined* bezeichnet. Es ist daher anzunehmen, dass eine gute SYCL-Implementierung die Datenanordnung für die jeweilige Hardware optimiert. (vgl. [syc18], S. 132)
SYCL bietet neben dem für alle Datentypen geeigneten *buffer* zusätzlich eine `image`-Klasse, die für die möglicherweise auf der Hardware vorhandenen Textur-Einheiten geeignet ist. (vgl. [syc18], S. 94)

Datenaffinität *Buffer*- und *image*-Objekte sind auf allen im System vorhandenen Devices sichtbar. Um die Synchronisierung zwischen Devices (möglicherweise von verschiedenen Herstellern) zu gewährleisten, kapseln diese Objekte unter Umständen mehrere Speicherreservierungen auf dem Host und den Devices. Eventuelle Abhängigkeiten zwischen Device-Zugriffen werden von der Laufzeitumgebung erkannt und im Hintergrund synchronisiert. Bei einer parallelen Verarbeitung des selben Objekts durch mehrere Devices kann der Programmierer explizit einen atomaren Zugriff definieren. (vgl. [syc18], S. 24)

Datenlokalität SYCLs Speicherhierarchie ist vierstufig. Auf der obersten Ebene liegt der *global memory*, der für alle *work-items* sichtbar ist, er entspricht also den gleichnamigen Speicherebenen in CUDA und HC.
Constant memory ist ebenfalls für alle *work-items* sichtbar, jedoch – wie der Name andeutet – nur lesbar. Konzeptionell stellt er damit das Äquivalent zu CUDAs gleichnamigem Speicher dar.
Auf der Ebene der *work-groups* befindet sich der *local memory*, der den einer *work-group* zugehörigen *work-items* die Kommunikation ermöglicht. Er entspricht CUDAs *shared memory* und HCs *tile static memory*.
Der Speicher eines einzelnen *work-items* wird als *private memory* bezeichnet. Variablen, die innerhalb des Kernels definiert werden, werden automatisch in diesem Speicherbereich abgelegt. *Private memory* ist damit konzeptionell wie ein Register aufzufassen. (vgl. [syc18], S. 27)

2.5.4 Aufgabensicht

Aufgabengraphen Aufgabengraphen sind ein inhärentes Prinzip des SYCL-Standards. Grundsätzlich sind alle Kernel asynchron zueinander; sofern Abhängigkeiten bestehen, ist es Aufgabe der SYCL-Laufzeitumgebung, diese zu erkennen und die Kernel-Ausführung entsprechend zu serialisieren. (vgl. [syc18], Abschnitt 3.4.1.2)
Der in der Abbildung 2.2 gezeigte Beispielgraph lässt sich in SYCL wie in Quelltext 2.18 gezeigt implementieren.

```
queue.submit([&](cl::sycl::handler& cgh)
{
    cgh.parallel_for(cl::sycl::nd_range<1>{...}, A);
});

// B wird nach A und parallel zu C ausgeführt
queue.submit([&](cl::sycl::handler& cgh)
{
    cgh.parallel_for(cl::sycl::nd_range<1>{...}, B);
});

// C wird nach A und parallel zu B ausgeführt
queue.submit([&](cl::sycl::handler& cgh)
{
    cgh.parallel_for(cl::sycl::nd_range<1>{...}, C);
});

// D wird nach B und C ausgeführt
queue.submit([&](cl::sycl::handler& cgh)
{
    cgh.parallel_for(cl::sycl::nd_range<1>{...}, D);
});
```

Quelltext 2.18: Aufgabengraph mit SYCL

2.6 Zusammenfassung und Bewertung

Zusammenfassend lässt sich feststellen, dass die einzelnen Spracherweiterungen ähnliche Fähigkeiten bieten. Gleichwohl bleibt an dieser Stelle festzuhalten, dass der gewählte Kriterienkatalog nicht alle Anwendungsfälle dieser APIs abdeckt; eine Untersuchung unter anderen Schwerpunkten könnte daher zu einer gänzlich anderen Bewertung führen.

Ein wichtiger Unterschied zwischen CUDA/HIP einerseits und HC und SYCL andererseits liegt in ihrer Modernität. CUDA/HIP ist host-seitig ein reines C-API und verlangt dem Programmierer dadurch bei der Speicherverwaltung erheblich mehr Arbeit und Aufmerksamkeit ab. Die C-Natur des API führt außerdem dazu, dass eine Integration in bestehenden C++-Code vergleichsweise aufwendig ist, da z.B. die Interaktion mit Containern nur über Zeiger möglich ist.

HC und SYCL lassen sich dagegen in Kombination mit modernem C++ nutzen und orientieren sich in vielen Konzepten an der Standardbibliothek, was ihre Integration in bestehenden C++-Code recht einfach macht. Besonders der einfache Übergang von Standard-Containern zu den jeweiligen Speicherverwaltungssystemen der APIs ist hier hervorzuheben.

Die Umsetzungen der Aufgabengraphen, wie sie bei CUDA/HIP und SYCL zu finden sind, sind für den Programmierer einfach verständlich (in SYCLs Fall sogar ein implizites Sprach-Feature) und gut auf komplexere Graphen skalierbar. HCs Ansatz mit `futures` ist hier klar unterlegen, da die Implementierung komplexerer Graphen deutlich mehr Aufwand erfordert.

Die in diesem Abschnitt vorgestellten Vergleichskriterien und ihre Erfüllung durch die Spracherweite-

rungen sind in der Tabelle 2.1 zusammengefasst.

Kriterium	CUDA	HIP	HC	SYCL
explizite Datenbewegung	ja	ja	ja	ja
implizite Datenbewegung	ja	nein	ja	ja
Datenanordnung (1D-Arrays)	Zeiger, entspricht Host	Zeiger, entspricht Host	opak, Anordnung verborgen	opak, Anordnung verborgen
Datenanordnung (2D-Arrays)	Zeiger, für GPU optimiert	nein	opak, Anordnung verborgen	opak, Anordnung verborgen
Datenanordnung (3D-Arrays)	Zeiger, für GPU optimiert	Zeiger, für GPU optimiert	opak, Anordnung verborgen	opak, Anordnung verborgen
Datenanordnung (Texturen)	opak, Anordnung verborgen	opak, Anordnung verborgen	nein	opak, Anordnung verborgen
Datenaffinität	explizite Befehle bei expliziter Bewegung, implizit bei impliziter Bewegung	explizite Befehle	implizit; explizite Befehle möglich	implizit
Datenlokalität	globaler Speicher, lokaler Speicher, programmierbare Caches	globaler Speicher, lokaler Speicher, programmierbare Caches (Texturen eingeschränkt)	globaler Speicher, lokaler Speicher	globaler Speicher, lokaler Speicher
Aufgabengraphen	Asynchronitäts-API, Graph-API	Asynchronitäts-API	Asynchronitäts-API (eingeschränkt)	automatisch

Tabelle 2.1: Zusammenfassung der Spracherweiterungs-Features

3 Konzepte und Methoden

3.1 Theoretische Konzepte

Um von den Spracherweiterungen und ihren unterschiedlichen Konzepten in Bezug auf die Ausführung von Kerneln und die Speicherhierarchie abstrahieren zu können, wird in diesem Abschnitt ein theoretisches GPU-Modell vorgestellt. Dieses umfasst alle für das Verständnis der Benchmarks notwendigen Eigenschaften einer GPU und lässt sich einfach auf die konkreten Konzepte der Spracherweiterungen abbilden.

3.1.1 Arbeitsgruppen und -einheiten

Bei der Ausführung von Kerneln sind zwei Ebenen zu unterscheiden:

- Eine **Arbeitseinheit** ist eine logische Instanz, die auf einen Hardware-Thread eines Multiprozessors abgebildet werden kann.
- **Arbeitsgruppen** umfassen eine unbestimmte Zahl von Arbeitseinheiten und werden auf die Multiprozessoren der GPU verteilt. Arbeitseinheiten innerhalb einer Gruppe sind untereinander möglicherweise asynchron, können aber durch Barrieren synchronisiert werden. Durch die Verteilung der Arbeitsgruppen auf die Multiprozessoren sind auch die Arbeitsgruppen untereinander asynchron und können nicht synchronisiert werden.

Die Tabelle 3.1 zeigt die Abbildung der hier vorgestellten Konzepte auf die äquivalenten Konzepte der verschiedenen Plattformen.

Modell	CUDA / HIP	HC	SYCL
Arbeitsgruppe	block	tile	work group
Arbeitseinheit	thread	thread	work item

Tabelle 3.1: Abbildung der Ausführungskonzepte des Modells auf die Entsprechungen der Spracherweiterungen

3.1.2 Speicherhierarchie

Das theoretische GPU-Modell kennt drei Ebenen der Speicherhierarchie:

- Die höchste Ebene ist der **globale Speicher**. Er entspricht dem globalen (GDDR- oder HBM-) Speicher auf realen GPUs, ist im Modell jedoch unendlich groß, während die Zugriffszeit sehr langsam ist.
- Die nächste Ebene ist der **lokale Speicher**. Dieser hat in der realen Welt unterschiedlich benannte Entsprechungen (siehe Tabelle 3.2), meint aber immer den schnellen Speicher, der einer Arbeitsgruppe zugeordnet ist. Er ist in der Kapazität beschränkt und deutlich kleiner als der globale Speicher, während er wesentlich schnellere Zugriffe ermöglicht.

- Auf der untersten Ebene befinden sich die **Register**, die jeder Ausführungseinheit in unbegrenzter Zahl zugeordnet sind. Wie in der realen Welt sind sie nicht direkt vom Programmierer ansprechbar. Der Modell-Compiler platziert in ihnen lokale Variablen der Arbeitseinheiten.

Modell	CUDA / HIP	HC	SYCL
globaler Speicher	global memory	global memory	global memory
lokaler Speicher	shared memory	`tile_static` memory	local memory
Register	register	register	-

Tabelle 3.2: Abbildung des Speicherkonzepte des Modells auf die Entsprechungen der Spracherweiterungen

3.1.3 Kernelsprache

Die in den vorherigen Abschnitten entwickelten Konzepte werden für die theoretische Beschreibung der verwendeten Benchmarks in eine fiktive Kernel-Sprache integriert. Bei dieser Sprache handelt es sich um normales C++ mit folgenden Erweiterungen:

- Das Schlüsselwort `local` markiert Variablen und Arrays, die im lokalen Speicher liegen.
- Der Befehl `synchronize()` wirkt als Barriere innerhalb einer Arbeitsgruppe. Es wird garantiert, dass alle vorherigen Zugriffe auf den (globalen und lokalen) Speicher und Rechenoperationen abgeschlossen wurden, bevor die Arbeitseinheiten die Befehle nach der Barriere ausführen können.
- Das Schlüsselwort `num_groups` ist eine Variable, die die Anzahl aller Arbeitsgruppen enthält.
- Das Schlüsselwort `group_idx` dient der eindeutigen Identifikation einer Arbeitsgruppe in Form einer ganzen Zahl. Die Indices sind fortlaufend, d.h. die erste Arbeitsgruppe hat den Index 0, die zweite den Index 1 und die n-te Arbeitsgruppe den Index $n - 1$.
- Das Schlüsselwort `group_dim` ist eine Variable, die die Anzahl der Arbeitseinheiten einer Gruppe beinhaltet.
- Das Schlüsselwort `unit_idx` dient der eindeutigen Identifikation einer Arbeitseinheit innerhalb einer Arbeitsgruppe. Die erste Einheit einer Gruppe trägt den Index 0, die zweite den Index 1 und die n-te Einheit den Index $n - 1$.
- Die globale Position einer Arbeitseinheit, also die eindeutige Identifikation über Gruppengrenzen hinweg, lässt sich mit Hilfe der vorgegebenen Schlüsselworte berechnen:
  `global_id = group_idx * group_dim + unit_idx`
- Vektoren werden in der Form `typZAHL` notiert, z.B. `int2` oder `float4`. Der Zugriff auf die Komponenten erfolgt über den Punktoperator und die Angabe des Felds. Beispielsweise sind die Komponenten des Vektors `a` vom Typ `float3` durch `a.x`, `a.y` und `a.z` ansprechbar.

Alle Indices sind im vorgestellten Modell eindimensional, was für die hier vorgestellten Benchmarks ausreicht.

3.2 zcopy-Benchmark

Der zcopy-Benchmark geht auf einen Beitrag des Nutzers *njuffa*[8] im NVIDIA-Forum zurück. Er hat das Ziel, die tatsächlich erreichbare Speicherbandbreite des globalen GPU-Speichers zu ermitteln, indem ein Kernel ausschließlich kombinierte Lese- und Schreibzugriffe (16 Byte pro Thread) durchführt. (vgl. [nju17])
Der Benchmark wurde für diese Arbeit übernommen und angepasst:

- Es wird sowohl die kombinierte Lese- und Schreibgeschwindigkeit als auch die reine Schreibgeschwindigkeit gemessen, da die Kombination eventuell zusätzlichen Overhead im Speicher-Controller der GPU erzeugen kann.
- Während der originale Benchmark nur auf 16 Byte pro Thread zugreift, wurden im Rahmen dieser Arbeit auch die Bandbreiten für die Speichergrößen 4 Byte, 8 Byte und 32 Byte ermittelt.

Quelltext 3.1 zeigt den Pseudo-Code des eingesetzten Kernels am Beispiel von 16 Byte pro Thread.

```
void read_write(double2* a, double2* b, size_t num_elems)
{
    int stride = num_groups * group_dim;
    int global_id = group_dim * group_idx + unit_idx;

    for(int i = global_id; i < num_elems; i += stride)
        b[i] = a[i];
}

void write(double2* b, size_t num_elems)
{
    int stride = num_groups * group_dim;
    int global_id = group_dim * group_idx + unit_idx;

    for(int i = global_id; i < num_elems; i += stride)
        b[i] = double2{0.0, 0.0};
}
```

Quelltext 3.1: zcopy-Benchmark

3.3 Reduction-Benchmark

3.3.1 Theoretischer Hintergrund

Die Reduktion eines Arrays ist eines der klassischen Konzepte der parallelen Programmierung. Mathematisch verbirgt sich dahinter das Reduzieren einer Menge von Zahlen auf eine kleinere Menge von Zahlen durch eine Funktion. Als Beispiel sei eine Menge von Zahlen $A = \{a_1, a_2, \ldots, a_n\}$ gegeben. Durch die Summenfunktion lässt sich das Ergebnis $b = \sum_{i=1}^{n} a_i$ ermitteln – in Mengen ausgedrückt wurde

[8] Vermutlich der NVIDIA-Angestellte Norbert Juffa.

A auf $B = \{b\}$ reduziert. Auf die gleiche Weise ließe sich die Menge $C = \{c\}$ durch eine Multiplikationsreduktion $c = \prod_{i=1}^{n} a_i$ berechnen.
Ein serieller Reduktionsalgorithmus ließe sich in Form einer Schleife schreiben, die über die Menge der Zahlen iteriert und das Ergebnis der Funktion $\oplus$ für alle aufeinander folgenden Zahlen berechnet:

1: $result \leftarrow 0$
2: **for** $i \leftarrow 1, n$ **do**
3: $\quad result \leftarrow result \oplus a_i$
4: **end for**

Da die Reihenfolge der Anwendung der Funktion $\oplus$ in diesem Beispiel nicht relevant ist, lässt sich der Algorithmus einfach parallelisieren, indem eine Menge von p Arbeitseinheiten $\oplus$ zunächst auf p Paare (a_i, a_j) anwendet, wobei a_i und a_j jeweils genau einmal zugeordnet werden (a_i wird also niemals zweimal verwendet). Im nächsten Schritt führen $\frac{p}{2}$ Einheiten die Funktion erneut auf den Ergebnissen des vorherigen Schritts aus, usw. Die Abbildung 3.1 visualisiert den Vorgang für acht Elemente $a_1, \ldots, a_8$ auf das Ergebnis b durch vier Ausführungseinheiten $E_1, \ldots, E_4$.

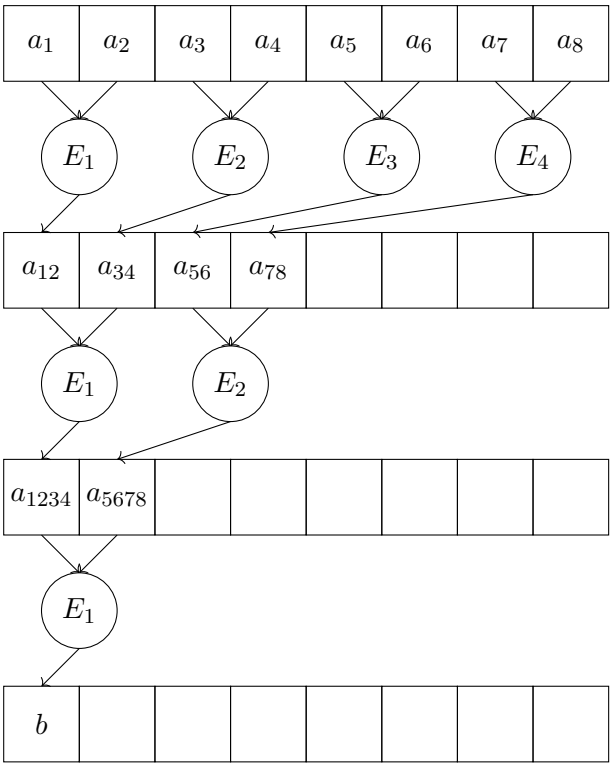


Abbildung 3.1: Visualisierung einer parallelen Reduktion

3.3.2 GPU-Implementierung

Im GPGPU-Kontext ist die Reduktion früh untersucht worden. So stellte Harris 2007 eine optimierte Reduktion für CUDA vor (vgl. [Har07]). Eine für NVIDIAs Kepler-Architektur optimierte Variante der Reduktion wurde 2014 von Luitjens präsentiert (vgl. [Lui14]). Da sich Luitjens' Implementierung NVIDIA-spezifischer Hardware-Intrinsiken bedient, die nicht auf jeder im Rahmen dieser Arbeit untersuchten Plattform verfügbar sind, wird hier ein anderer Ansatz verwendet.

Der Kernel, der die Reduktion durchführt, besteht aus zwei Stufen. Während der ersten Stufe iterieren x Arbeitsgruppen mit jeweils p Arbeitseinheiten über den globalen Speicher mit n Elementen. Die Arbeitseinheiten führen die Reduktion für jedes Elementepaar aus, wobei jeder Iterationsschritt $x \cdot p$ Elemente vom vorherigen Schritt entfernt auf den Speicher zugreift. Unter der Annahme, dass n ein Vielfaches von p ist, werden $y = \frac{n}{p}$ Arbeitsgruppen gebraucht. Gilt $x \leq y$ unter der Bedingung, dass y ein Vielfaches von x ist, führt jede Arbeitseinheit die Reduktionsoperation im ersten Schritt genau y-mal aus. Dieses Zugriffsmuster ist als *(grid-)stride loop* bekannt und ein häufig genutztes Konzept der GPGPU-Programmierung (vgl. [har13] für eine Einführung und Erklärung der Vorteile).
Während der zweiten Stufe werden die bisherigen Ergebnisse der einzelnen Arbeitseinheiten in den lokalen Speicher geladen. Dann führt die Hälfte der Arbeitseinheiten die Reduktionsoperation auf jeweils einem Elementepaar des lokalen Speichers aus, dann ein Viertel auf der Arbeitseinheiten auf den neuen Ergebnissen, usw. Am Ende gibt es pro Arbeitsgruppe genau ein Teilergebnis der Reduktionsoperation (also x Ergebnisse), das wieder in den globalen Speicher geschrieben wird.
Durch einen erneuten Aufruf des Kernels mit einer Arbeitsgruppe und $\frac{x}{2}$ Arbeitseinheiten lässt sich in einem weiteren Schritt das Gesamtergebnis der Reduktion berechnen. Quelltext 3.2 zeigt den Pseudo-Code des Kernels.

```
void block_reduce(const int* data, int* result, size_t num_elems)
{
    local int shared[p];

    int global_id = group_idx * group_dim + unit_idx;
    int tsum = data[global_id]; // vermeide neutrales Element
    int grid_size = group_dim * num_blocks;
    global_id += grid_size;

    // 1. Stufe: grid-stride loop
    while((global_id + 3 * grid_size) < num_elems)
    {
        tsum += data[global_id] + data[global_id + grid_size] +
                data[global_id + 2 * grid_size] +
                data[global_id + 3 * grid_size];
        i += 4 * grid_size;
    }

    // verbleibende Elemente, falls n kein Vielfaches von p ist
    while(global_id < num_elems)
    {
        tsum += data[global_id];
        global_id += grid_size;
    }

    // schreibe Ergebnis in lokalen Speicher
    shared[unit_idx] = tsum;
    synchronize();

    // 2. Stufe: Reduktion innerhalb der Gruppe
    for(int bs = group_dim, bsup = (group_dim + 1) / 2;
        bs > 1;
        bs /= 2, bsup = (bs + 1) / 2)
    {
        bool cond = unit_idx < bsup // erste Gruppenhälfte
            && unit_idx + bsup < group_dim
            && (group_idx * group_dim + unit_idx + bsup) < num_elems;

        if(cond) shared[unit_idx] += shared[unit_idx + bsup];
        synchronize();
    }

    // Ergebnis in globalen Speicher schreiben
    if(unit_idx == 0) result[group_idx] = shared[0];
}
```

Quelltext 3.2: Reduction-Benchmark

3.4 N-Body-Benchmark

3.4.1 Vorbemerkung

Eine effiziente Implementierung einer N-Body-Simulation mit quadratischer Komplexität für GPUs wurde 2007 von Nyland et al. vorgestellt. Der in dieser Arbeit verwendete Benchmark folgt dieser Implementierung, die theoretische Beschreibung ist (gekürzt) ebenfalls der Arbeit von Nyland et al. entnommen. (vgl. [NHP07])

3.4.2 Einführung

N-Body-Simulationen wenden numerische Methoden an, um die Entwicklung eines Systems vieler miteinander interagierender Körper zu approximieren. N-Body-Probleme sind in den Naturwissenschaften zahlreich vertreten, beispielsweise bei der Simulation vieler Galaxien oder Sterne und deren Wechselwirkungen durch die Schwerkraft.
Eine N-Body-Simulation aller Körperpaare ist eine *brute-force*-Technik, die alle paarweisen Interaktionen auswertet. Aufgrund der quadratischen Komplexität $\mathcal{O}(n^2)$ ist die Rechenlast bei großen Systemen sehr hoch: ein System mit 16.384 Körpern, das 20 Zeitschritte pro Sekunde simuliert, berechnet über fünf Milliarden Interaktionen pro Sekunde. Aus diesem Grund ist eine solche Simulation ein interessantes Ziel für den Einsatz paralleler Beschleuniger.

3.4.3 Mathematischer Hintergrund

In der in dieser Arbeit verwendeten Simulation wird das Gravitationspotential aller Körper berechnet. Die in den folgenden Formeln vorkommenden Vektoren werden in der Form $\vec{a}, \vec{b}, \vec{c}$ notiert und sind grundsätzlich dreidimensional.
In einem System aus N Körpern mit der Startposition $\vec{x}_i$ und der Geschwindigkeit $\vec{v}_i$ ($1 \leq i \leq N$) ergibt sich die auf den Körper i durch die Anziehungskraft des Körpers j wirkende Kraft $\vec{f}_{ij}$ wie folgt:

$$\vec{f}_{ij} = G\frac{m_i m_j}{|\vec{r}_{ij}|^2} \cdot \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|},$$

wobei m_i und m_j die Massen der Körper i und j sind, $\vec{r}_{ij} = \vec{x}_j - \vec{x}_i$ der Vektor vom Körper i zum Körper j ist und G die Gravitationskonstante darstellt. Der linke Faktor – der *Betrag* der Kraft – ist proportional zum Produkt der Massen und schrumpft mit zunehmendem Abstand zwischen den Körpern i und j. Der rechte Faktor ist die *Richtung* der Kraft. Aufgrund der anziehenden Wirkung zwischen den Körpern i und j ist dieser Vektor ist ein Einheitsvektor.
Die gesamte auf den Körper i wirkende Kraft $\vec{F}_i$, die sich aus den Interaktionen mit den anderen $N-1$ Körpern ergibt, ist die Summe aller Interaktionen:

$$\vec{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \vec{f}_{ij} = Gm_i \cdot \sum_{\substack{1 \leq j \leq N \\ j \neq i}} \frac{m_j \vec{r}_{ij}}{|\vec{r}_{ij}|^3}.$$

Wenn sich Körper einander nähern, wächst die zwischen ihnen wirkende Kraft bis ins Unendliche, was bei der numerischen Integration unerwünscht ist. In astrophysikalischen Simulationen werden daher Kol-

lisionen zwischen Körpern grundsätzlich ausgeschlossen. Aus diesem Grund wird ein Schwächungsfaktor $\varepsilon^2 > 0$ hinzugefügt und der Nenner wie folgt umgeschrieben:

$$\vec{F_i} \approx Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\sqrt{|\vec{r}_{ij}|^2 + \varepsilon^2}^3}.$$

Die Bedingung $j \neq i$ wird bei der Summe nicht länger benötigt, da $\vec{f}_{ii} = 0$, wenn $\varepsilon^2 > 0$. Der Schwächungsfaktor stellt die Interaktion zwischen zwei Plummer-Massepunkten dar, also Massen, die sich wie kugelförmige Galaxien verhalten (vgl. [Aar03] und [DI93]). Effektiv beschränkt der Schwächungsfaktor die Größe der Kräfte zwischen Körpern, was bei der numerischen Integration wünschenswert ist.
Um über die Zeit zu integrieren, muss die Beschleunigung $\vec{a}_i = \frac{\vec{F}_i}{m_i}$ auf die Position und Geschwindigkeit des Körpers i angewendet werden, wodurch sich die Berechnung weiter vereinfacht:

$$\vec{a}_i \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\sqrt{|\vec{r}_{ij}|^2 + \varepsilon^2}^3}$$

Der numerische Integrator, der die Positionen und Geschwindigkeiten aktualisiert und auch in dieser Arbeit zum Einsatz kommt, ist ein Leapfrog-Verlet-Integrator (vgl. [Ver67]). Er ist auf dieses Problem anwendbar und berechnungseffizient, d.h. die Genauigkeit ist im Verhältnis zum Rechenaufwand hoch. Die Wahl der Integrationsmethode für N-Body-Simulationen hängt vom beobachteten System ab. Der Integrator wird in den Messungen mit erfasst, in dieser Arbeit jedoch nicht gesondert beschrieben, da er eine lineare Komplexität von $\mathcal{O}(n)$ besitzt und mit zunehmendem n bedeutungslos wird.

3.4.4 GPU-Implementierung

Den vorgestellten Algorithmus kann man sich als Berechnung jedes Feldes $\vec{f}_{ij}$ in einem Gitter der Größe $N \times N$ vorstellen. In diesem Fall ist die Gesamtkraft $\vec{F}_i$ bzw. die Beschleunigung $\vec{a}_i$ eines Körpers i die Summe aller Einträge der Zeile i. Jeder Eintrag kann unabhängig von den anderen berechnet werden, was einen möglichen Parallelisierungsgrad von $\mathcal{O}(n^2)$ ergibt. Dieser Ansatz hätte eine Speicherkomplexität von ebenfalls $\mathcal{O}(n^2)$ zur Folge, was die Performanz durch die verfügbare Speicherbandbreite limitieren würde. Stattdessen werden einige Berechnungen serialisiert, um durch die Wiederverwendungen von Daten die benötigte Speicherbandbreite zu verringern und die höchstmögliche Leistung der arithmetischen Einheiten zu erreichen.
Das Gitter wird dazu in quadratische Kacheln gleicher Größe aufgeteilt, die aus p Zeilen und p Spalten bestehen. $2p$ Körperbeschreibungen werden benötigt, um alle p^2 Interaktionen innerhalb der Kachel zu berechnen, von denen p wiederverwendet werden können. Diese Beschreibungen können dadurch im lokalen Speicher oder in den Registern der auf der GPU verbauten Multiprozessoren gehalten werden. Der Effekt der Interaktionen auf die p Körper innerhalb der Kachel wird als Aktualisierung von p Beschleunigungsvektoren gespeichert.
Um die Daten effizient wiederverwenden zu können, erfolgen die Berechnungen in einer Zeile in sequentieller Reihenfolge, während die einzelnen Zeilen parallel berechnet werden. Abbildung 3.2 zeigt links die Ausführungsstrategie und rechts die Ein- und Ausgabewerte einer Kachel.
Der Pseudo-Code in Quelltext 3.3 dient der Berechnung der Krafteinwirkung auf einen Körper i durch die Interaktion mit dem Körper j und aktualisiert die Beschleunigung a_i des Körpers i. Die Beschreibungen der Körper enthalten die Koordinaten in den Feldern `x`, `y` und `z` sowie die Körpermasse im Feld

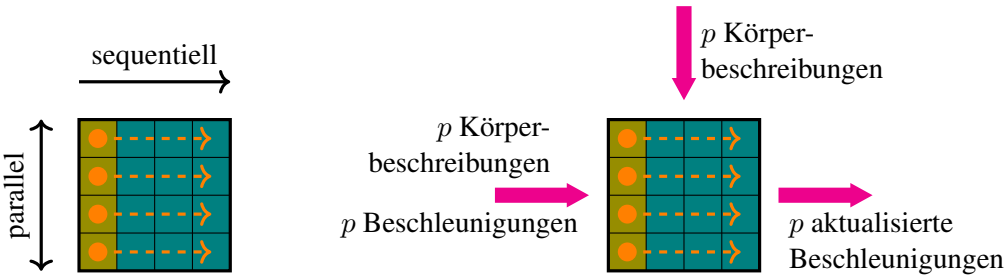


Abbildung 3.2: Schema einer Kachel

w. Die Berechnung einer Interaktion benötigt 20 FLOPs.

```
const float epsilon2 = epsilon * epsilon;

float3 body_body_interaction(float4 i, float4 j, float3 ai)
{
    // 3 FLOPs
    float3 r = j.xyz - i.xyz;

    // 6 FLOPs
    float dist_sqr = skalarprodukt(r, r) + epsilon2;

    // 4 FLOPs (2x Multiplikation, Wurzel, Inversion
    float inv_dist_cube = 1.0 / sqrt(dist_sqr * dist_sqr * dist_sqr);

    // 1 FLOP
    float s = j.w * inv_dist_cube;

    // 6 FLOPs -- ai und r sind Vektoren, s ist skalar
    ai += r * s;

    return ai;
}
```

Quelltext 3.3: Berechnung der Interaktion zwischen zwei Körpern

Eine Kachel wird von p Threads ausgewertet, wobei jeder Thread die Beschleunigung eines Körpers als Ergebnis seiner Interaktion mit p anderen Körpern aktualisiert. Das heißt, dass p Körperbeschreibungen aus dem globalen Speicher der GPU in den lokalen Speicher des Multiprozessors geladen werden. Jeder Thread innerhalb des Blocks wertet p aufeinanderfolgende Interaktionen aus, was p aktualisierte Beschleunigungen ergibt. Der Pseudo-Code in Quelltext 3.4 zeigt diese Berechnung. Der Parameter `my_position` enthält die Position des Körpers, den der aktuelle Thread auswertet, während das im lokalen Speicher liegende Array `sh_position` die Beschreibungen der p interagierenden Körper enthält.

```
float3 tile_calculation(float4 my_position, float3 accel)
{
    local float4 sh_position[p];
    for(int i = 0; i < p; ++i)
        accel = body_body_interaction(my_position, sh_position[i],
                                      accel);

    return accel;
}
```

Quelltext 3.4: Berechnung der Interaktionen einer Kachel mit $p \times p$ Elementen

Im hier verwendeten Benchmark verarbeitet eine Gruppe von p Arbeitseinheiten sequentiell eine Reihe von $\frac{N}{p}$ Kacheln der Größe $p \times p$. Die Wiederverwendung von Daten wird effektiver, je länger eine Zeile wird. Die Zeilenlänge beeinflusst gleichzeitig die Größe der aus dem globalen in den lokalen Speicher kopierten Daten. Die Kachelgröße selbst bestimmt somit den Bedarf an Registern und lokalem Speicher. Vor der Verarbeitung einer Kachel lädt jede Arbeitseinheit einen Körper in den lokalen Speicher und synchronisiert mit den anderen Arbeitseinheiten innerhalb der Arbeitsgruppe. Somit liegen vor der Verarbeitung jeder Kachel p aufeinanderfolgende Körper im lokalen Speicher.

Die Abbildung 3.3 zeigt eine Arbeitsgruppe, die nacheinander mehrere Kacheln verarbeitet. In horizontaler Richtung erstreckt sich die Zeit, während die Parallelität in vertikaler Richtung zu sehen ist. Die fett gezeichneten Linien rahmen die Kacheln ein und zeigen gleichzeitig die Zeitpunkte, an denen der lokale Speicher befüllt wird und die Synchronisierung der Arbeitseinheiten erfolgt. Alle p Arbeitseinheiten einer Gruppe berechnen innerhalb einer Kachel die Kräfte, die auf p Körper wirken (eine Einheit pro Körper); bei $\frac{N}{p}$ Kacheln pro Arbeitsgruppe berechnet damit jede Einheit alle N Interaktionen pro Körper.

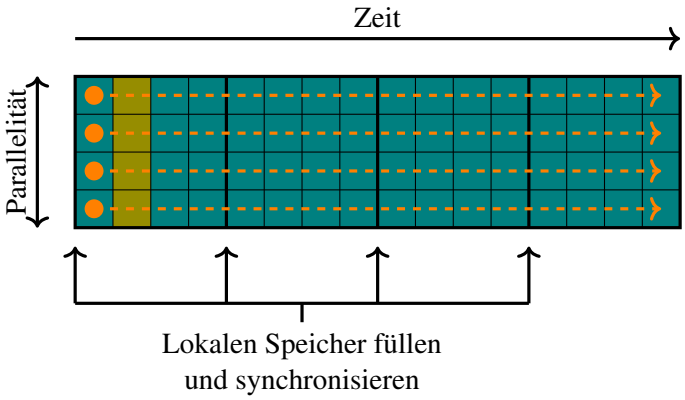


Abbildung 3.3: Berechnungen einer Arbeitsgruppe

Quelltext 3.5 zeigt den Pseudo-Code, der diese Berechnung für GPUs umsetzt. Die Parameter `x` und `a` sind Zeiger auf den globalen Speicher und enthalten die Positionen (`x`) und Beschleunigungen (`a`) der Körper. Die Iteration über die Kacheln erfordert zwei Synchronisierungspunkte. Der Erste stellt sicher, dass alle Felder des lokalen Speichers befüllt wurden, bevor die Berechnung beginnt. Der zweite Synchronisationspunkt garantiert, dass alle Berechnungen abgeschlossen wurden, bevor die nächste Kachel

verarbeitet wird.

```
void tile_calculation(float4* x, float3* a)
{
    local float4 sh_position[p];

    float3 acc = {0.0, 0.0, 0.0};
    int global_id = group_idx * group_dim + unit_idx;

    float4 my_position = x[global_id];

    for(int i = 0, tile = 0; i < N; i += p, ++tile)
    {
        int idx = tile * group_dim + unit_idx;
        sh_position[unit_idx] = x[idx];
        synchronize();

        acc = tile_calculation(my_position, acc);
        synchronize();
    }

    // Ergebnis für folgenden Integrationsschritt abspeichern
    a[global_id] = acc;
}
```

Quelltext 3.5: Berechnung aller N Interaktionen für p Körper innerhalb einer Arbeitsgruppe mit p Arbeitseinheiten

Der gezeigte Pseudo-Code berechnet die Beschleunigung von p Körpern eines Systems, die durch ihre Interaktion mit allen N Körpern im System hervorgerufen wird. Um die Beschleunigung aller N Körper zu berechnen, werden mehrere Arbeitsgruppen eingesetzt. Da es p Arbeitseinheiten pro Arbeitsgruppe und eine Arbeitseinheit pro Körper gibt, ist die Zahl der benötigten Arbeitsgruppen $\frac{N}{p}$. Im Gesamtsystem gibt es somit N Arbeitseinheiten, die jeweils N Berechnungen durchführen. Insgesamt werden auf diese Weise N^2 Interaktionen berechnet. Eine Visualisierung dieses Vorgangs findet sich in Abbildung 3.4. Die vertikale Dimension zeigt die Parallelisierung, die durch $\frac{N}{p}$ unabhängige Arbeitsgruppen mit jeweils p Arbeitseinheiten erreicht wird. Die horizontale Dimension zeigt die sequentielle Verarbeitung von N Berechnungen durch jede Arbeitseinheit. Eine Arbeitsgruppe lädt ihren lokalen Speicher alle p Schritte neu, um p Positionen zu teilen.

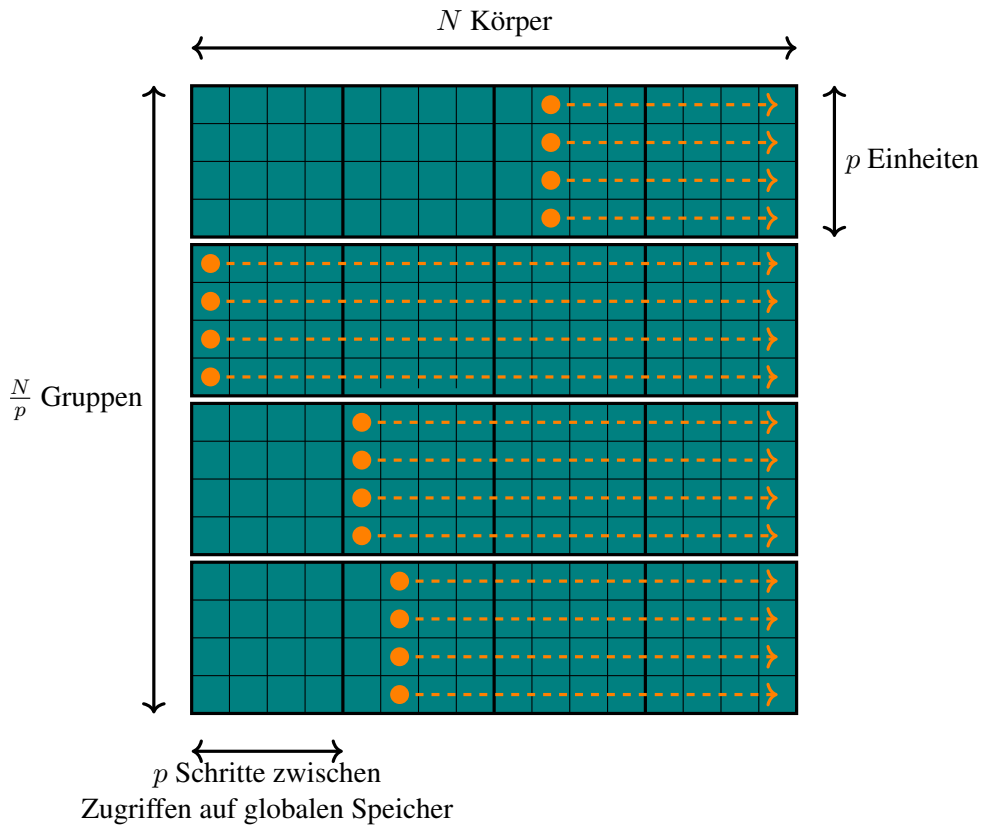


Abbildung 3.4: Berechnung aller Interaktionen

4 Messungen auf NVIDIA-GPUs

4.1 Verwendete Hard- und Software

Die hier gezeigten Benchmark-Ergebnisse wurden auf einem Knoten der `gpu2`-Partition des HPC-Systems Taurus gemessen, der über zwei GPUs des Modells Tesla K20x (mit aktiviertem ECC) verfügt. Die Messungen fanden innerhalb der SCS5-Umgebung statt, in der die folgende Software verwendet wurde:

- `CUDA/10.0.130` (Modulsystem)
- `GCC/7.3.0-2.30` (Modulsystem)
- HIP (Version 1.5.19061, ROCm-GitHub-Repository)
- ComputeCpp (Codeplays SYCL-Implementierung, Version 1.0.5 für Ubuntu 14.04)

4.2 zcopy

4.2.1 Vorüberlegungen

Um eine optimale Ausnutzung der Speicherbandbreite zu erreichen, ist ein genauer Blick auf die verwendete Architektur erforderlich. Die K20x-GPUs verfügen auf jedem Multiprozessor bei einfacher Genauigkeit über 192 Hardware-Threads. Ein Multiprozessor kann somit sechs *Warps* parallel ausführen. Ein häufiges Muster der CUDA-Programmierung ist die Verwendung von Blockgrößen, die eine Zweierpotenz darstellen. Aus diesem Grund werden auf Blockgrößen untersucht, die ein Vielfaches von 64 sind (bis 1.024). Zusätzlich werden auf der K20x Blockgrößen betrachtet, die ein Vielfaches von 192 sind (bis 768), um eventuelle Effekte zu entdecken.
Die K20x-GPU besitzt eine L1-Cacheline-Größe von 128 Byte (vgl. [pas18], Abschnitt 1.4.3.2). Sofern alle Threads eines *Warps* benachbarte 4-Byte-Sequenzen laden, lassen sich diese Zugriffe in einer einzigen Cacheline vereinen, was die pro Thread verwendete Speicherbandbreite drastisch reduziert und eine insgesamt höhere Auslastung erlaubt.
Um eine höhere Auslastung des Speicher-Controllers zu erreichen und die Anzahl der notwendigen Schleifendurchläufe innerhalb des Kernels zu verringern, wodurch sich die Zahl abhängiger Instruktionen verringert, werden die 4 Byte pro Thread in diesem Benchmark vervierfacht. Jeder Thread lädt also 16 Byte in Form eines Elements vom Typ `float4`. Die Anhang befindlichen Quelltexte A.1 (CUDA), B.1 (HIP) und D.1 (SYCL) zeigen die Kernel-Implementierungen der verschiedenen Spracherweiterungen.

4.2.2 Messmethoden

Der Speicher der GPU wurde zunächst mit zwei gleich großen Datenfeldern A und B befüllt, die jeweils etwas weniger als die Hälfte des GPU-Speichers einnahmen und n `float4`-Elemente umfassten. Diese wurden mit Nullen (A) bzw. *NaN* (B) initialisiert. Anschließend wurden die Kernel für jede Block-Größe jeweils zehn Mal ausgeführt, wobei für jeden Kernel-Durchlauf die benötigte Zeit über die Event-Funktionalitäten der verschiedenen Spracherweiterungen gemessen wurde. Der minimale Zeitbedarf $t_{\min}$ (in s) jedes Kernels wird als Grundlage für die Berechnung der Bandbreite B (in $\mathrm{GiB\,s^{-1}}$) genommen:

$$B = \frac{k \cdot \text{sizeof}(\texttt{float4}) \cdot n}{t_{\text{min}}},$$

wobei der Faktor k die Werte $2 \cdot 10^{-9}$ (kombinierter Lese- und Schreibzugriff) oder $1 \cdot 10^{-9}$ (Schreibzugriff) annehmen kann.
Der Quelltext 4.1 zeigt die verwendeten Compiler-Flags.

```
# CUDA-Compiler
nvcc -std=c++14 -O3 -gencode arch=compute_35,code=sm_35

# HIP-Compiler
hipcc -O3 -std=c++14 -gencode arch=compute_35,code=sm_35

# SYCL-Compiler
compute++ -std=c++17 -O3 -sycl-driver -sycl-target ptx64
```

Quelltext 4.1: Compiler-Flags für zcopy

4.2.3 Ergebnisse

Ein Blick auf die mit CUDA ermittelten Ergebnisse zeigt, dass die K20x-GPU bei Speicherzugriffen von größeren Thread-Blöcken als auch von vielen Thread-Blöcken profitiert. Dies gilt sowohl für die „generischen“ Blockgrößen, die ein Vielfaches von 64 darstellen (siehe die Abbildungen 4.1 und 4.2), als auch die auf die Kepler-Architektur angepassten Blockgrößen (siehe die Abbildungen 4.3 und 4.4). Außerdem bestätigt sich die Annahme, dass mit der angepassten Blockgröße eine etwas bessere Bandbreite erzielt werden kann (siehe Abbildung 4.5). Dies hilft insbesondere dem kombinierten Lese- und Schreibvorgang, der dadurch bei einer großen Blockzahl nahezu die selbe Bandbreite wie der reine Schreibvorgang erreicht (siehe Abbildung 4.6).
Die mit HIP und SYCL ermittelten Ergebnisse zeigen ein ähnliches Bild (siehe die angehängten Abbildungen in den Abschnitten E.2 (HIP) und E.3 (SYCL)).
Hinsichtlich der erreichten Speicherbandbreite bestehen zwischen den Spracherweiterungen keine nennenswerten Unterschiede, wie die Abbildung 4.7 zeigt.
Die K20x-GPU erreicht etwa 75% der theoretisch möglichen Bandbreite (siehe Abbildung 4.8). Dies ist durch das aktivierte ECC (vgl. [bes18], Abschnitt 8.2.1) und eventuell vorhandene Ungenauigkeiten der Event-Systeme der Spracherweiterungen zu erklären[9]. Für den Reduction-Benchmark wird daher eine tatsächlich erreichbare Obergrenze von $188\,\text{GiB}\,\text{s}^{-1}$ angenommen.

4.3 Reduction

4.3.1 Implementierung

Die Implementierungen des Reduktionskernels sind dieser Arbeit angehängt und befinden sich in den Quelltexten A.2 (CUDA), B.2 (HIP) und D.2 (SYCL).

[9] Der Einsatz geeigneter Profiler könnte aufgrund der besseren Zeitauflösung noch kleinere Änderungen der Ergebnisse bewirken. Da es aber für SYCL derzeit keine Profiler gibt, die auf NVIDIA-Hardware messen können, wurde an dieser Stelle darauf verzichtet.

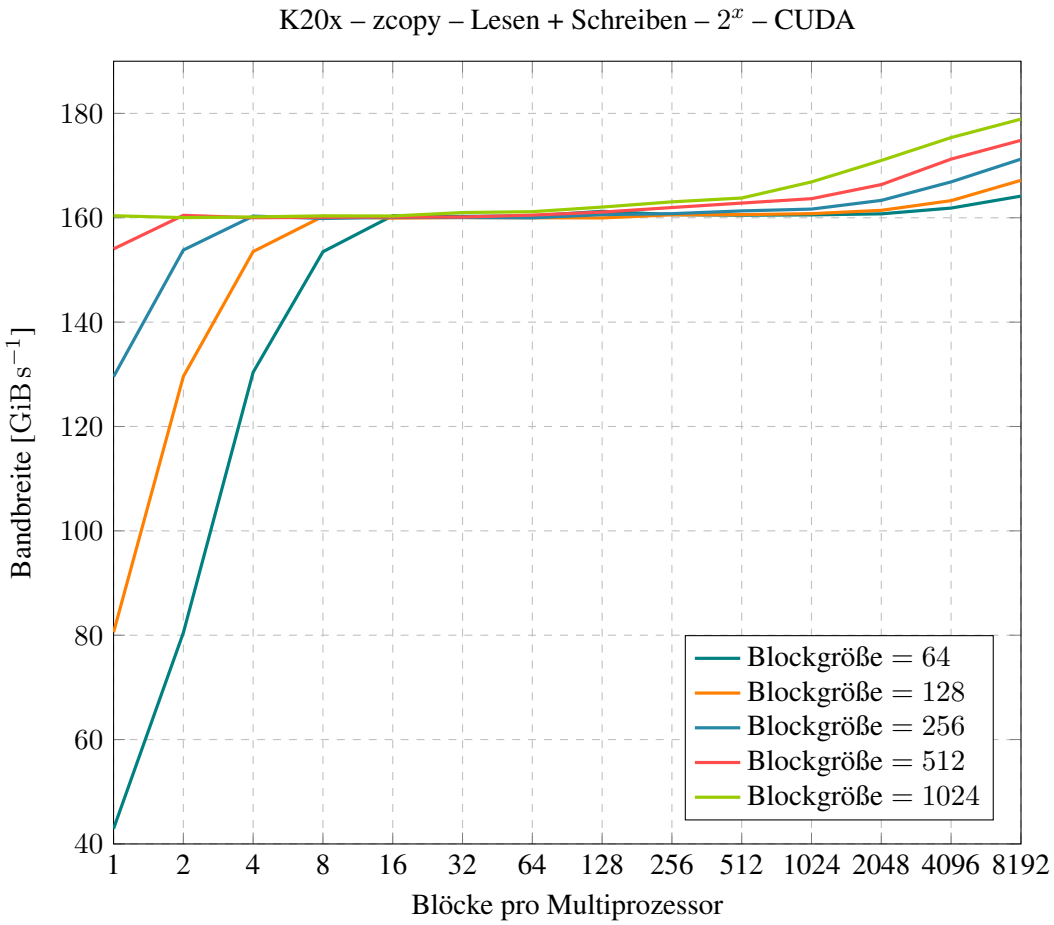


Abbildung 4.1: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Lesen und Schreiben, CUDA)

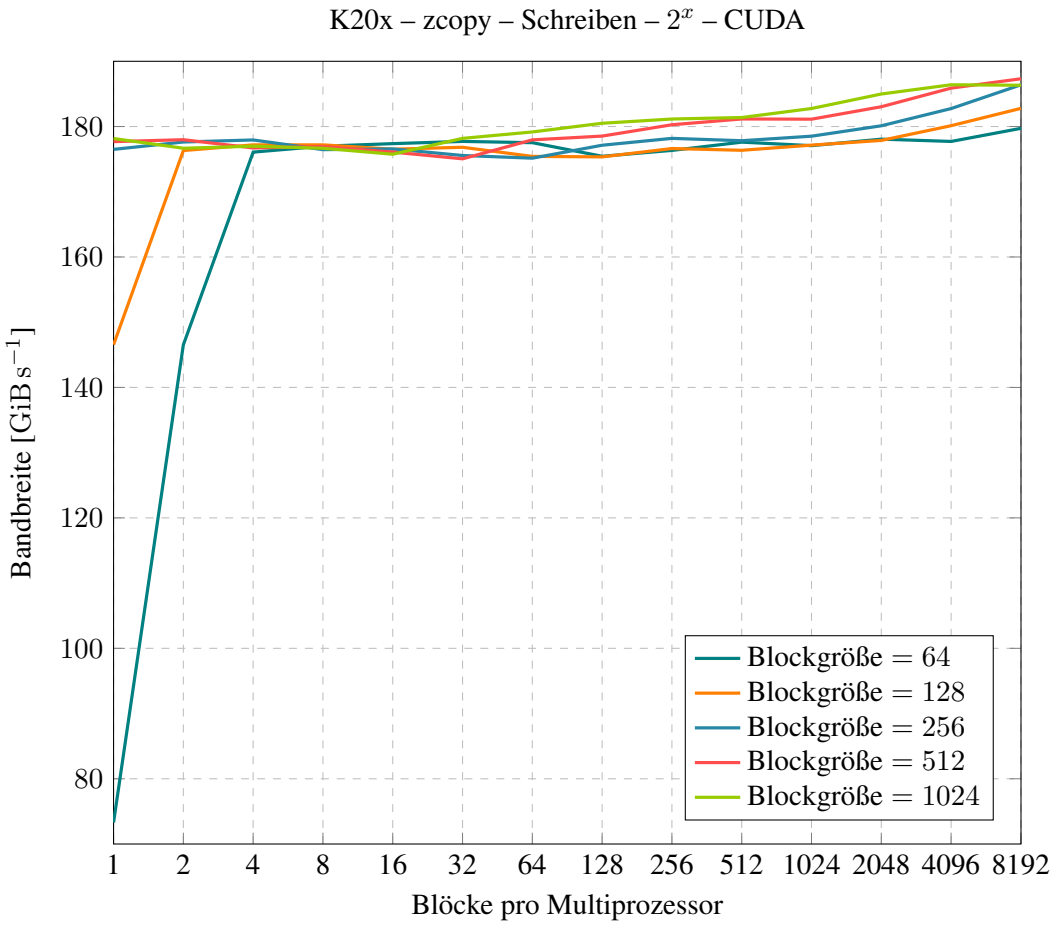


Abbildung 4.2: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Schreiben, CUDA)

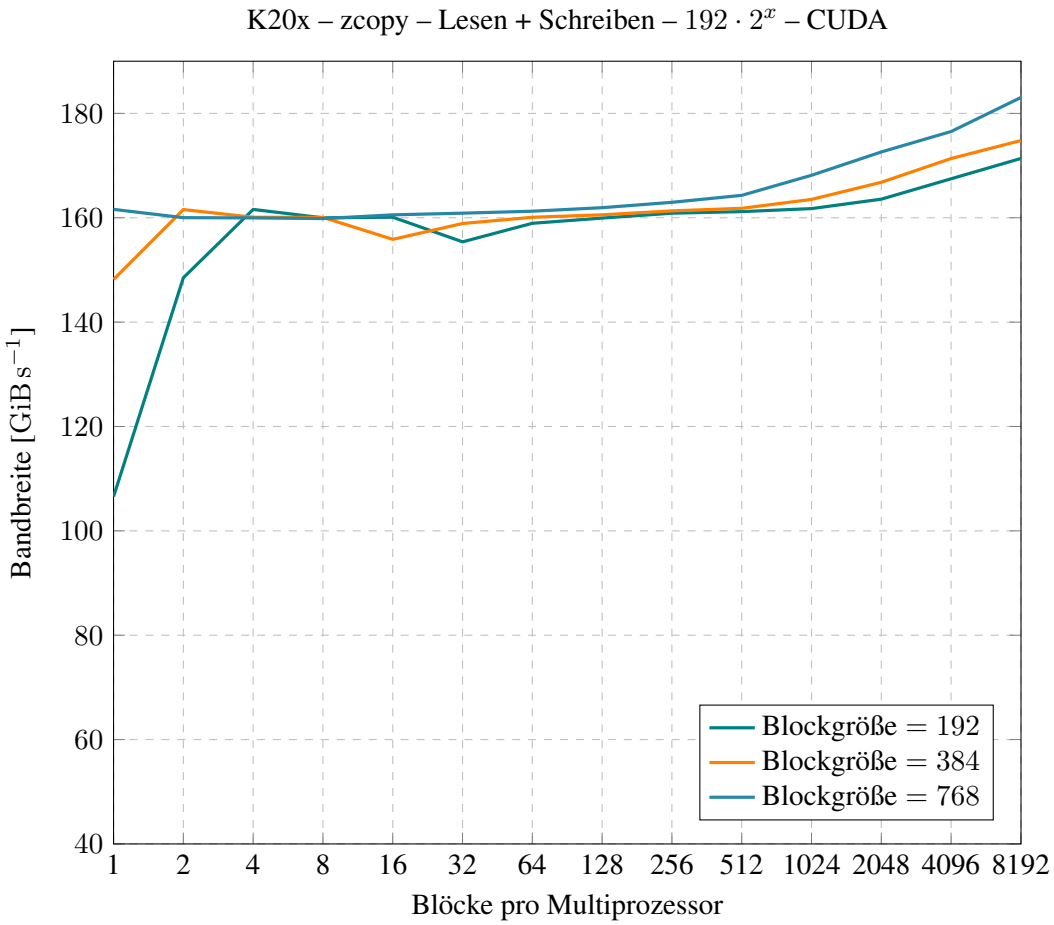


Abbildung 4.3: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Lesen und Schreiben, CUDA)

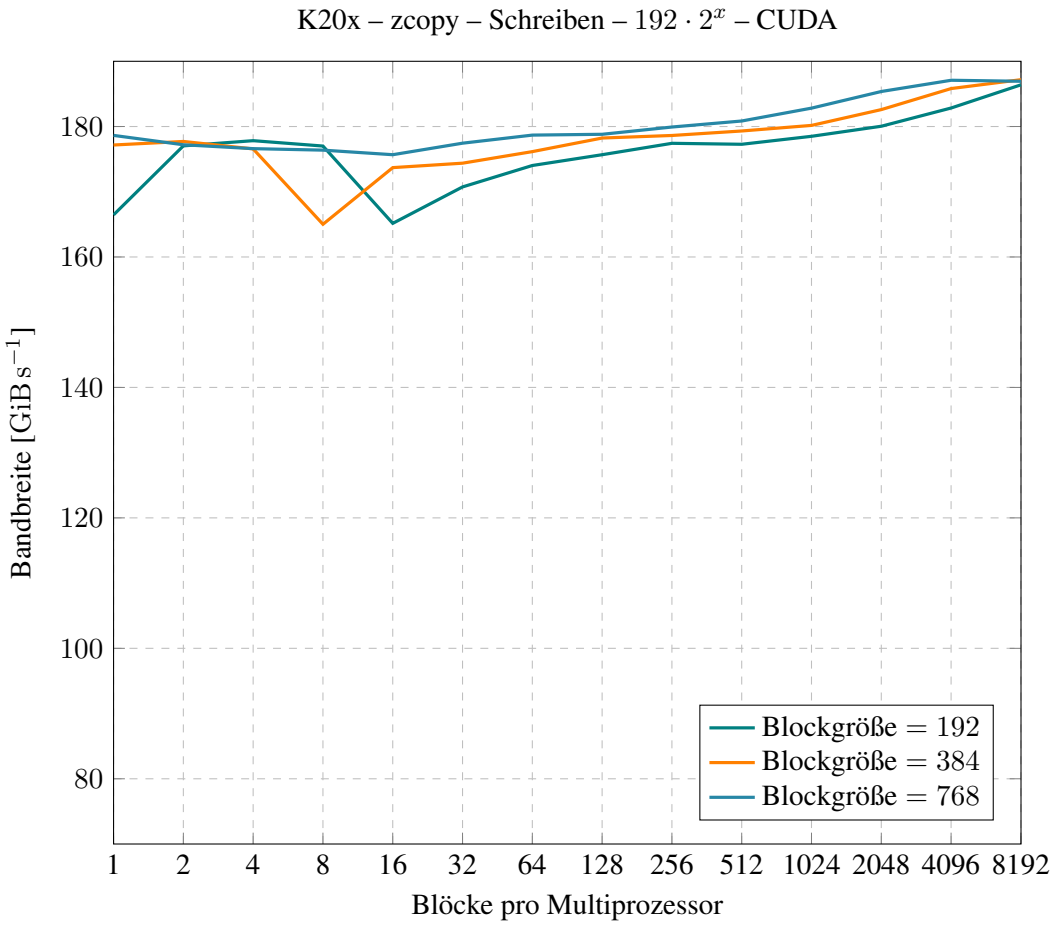


Abbildung 4.4: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Schreiben, CUDA)

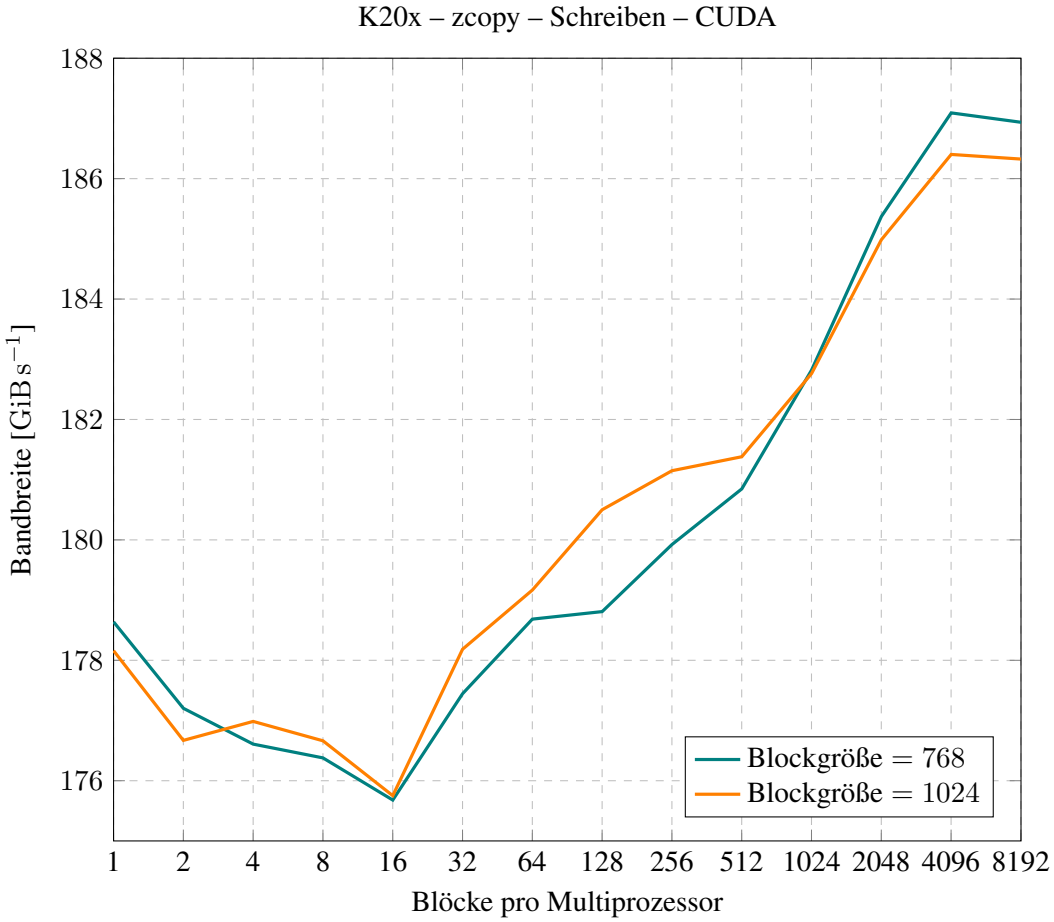


Abbildung 4.5: zcopy: K20x-Bandbreite für Zweierpotenz- ($n = 117440512$) und 192er-Größen ($n = 88080384$, Schreiben, CUDA)

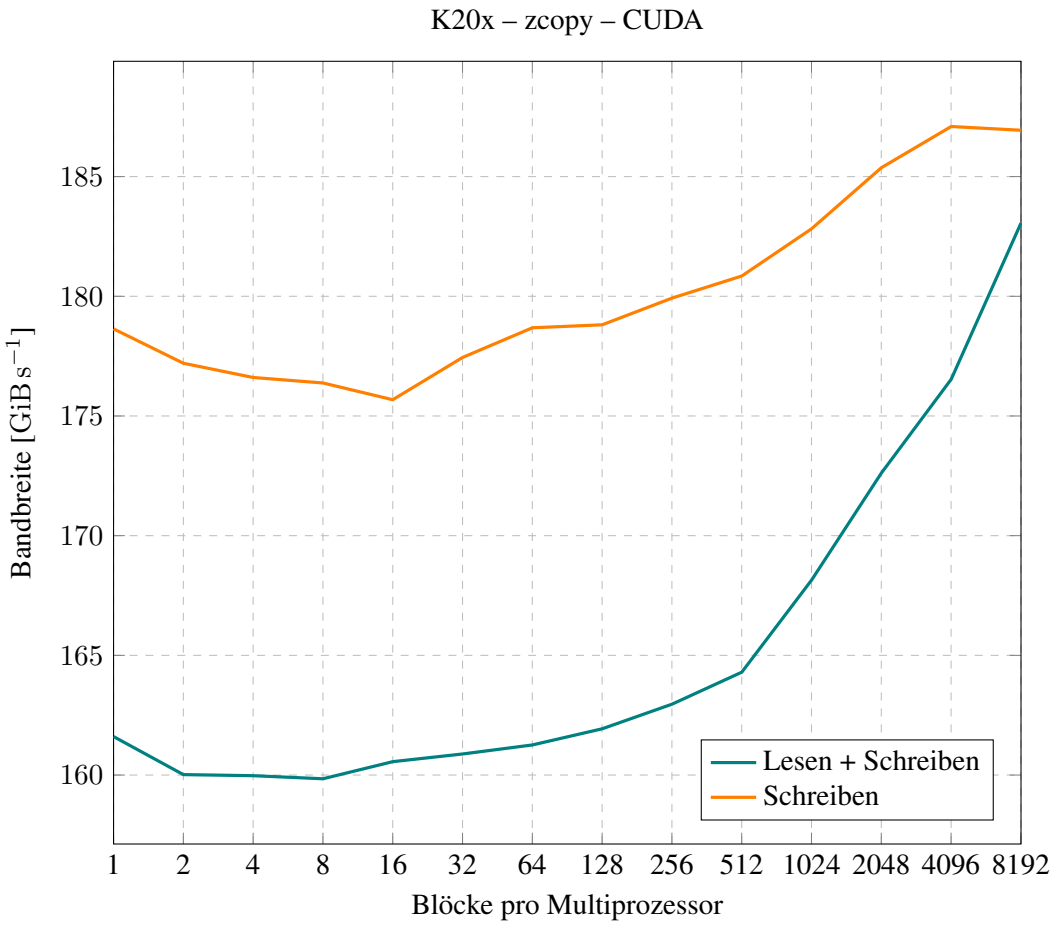


Abbildung 4.6: zcopy: K20x-Bandbreite für 768er-Blöcke ($n = 88080384$, CUDA)

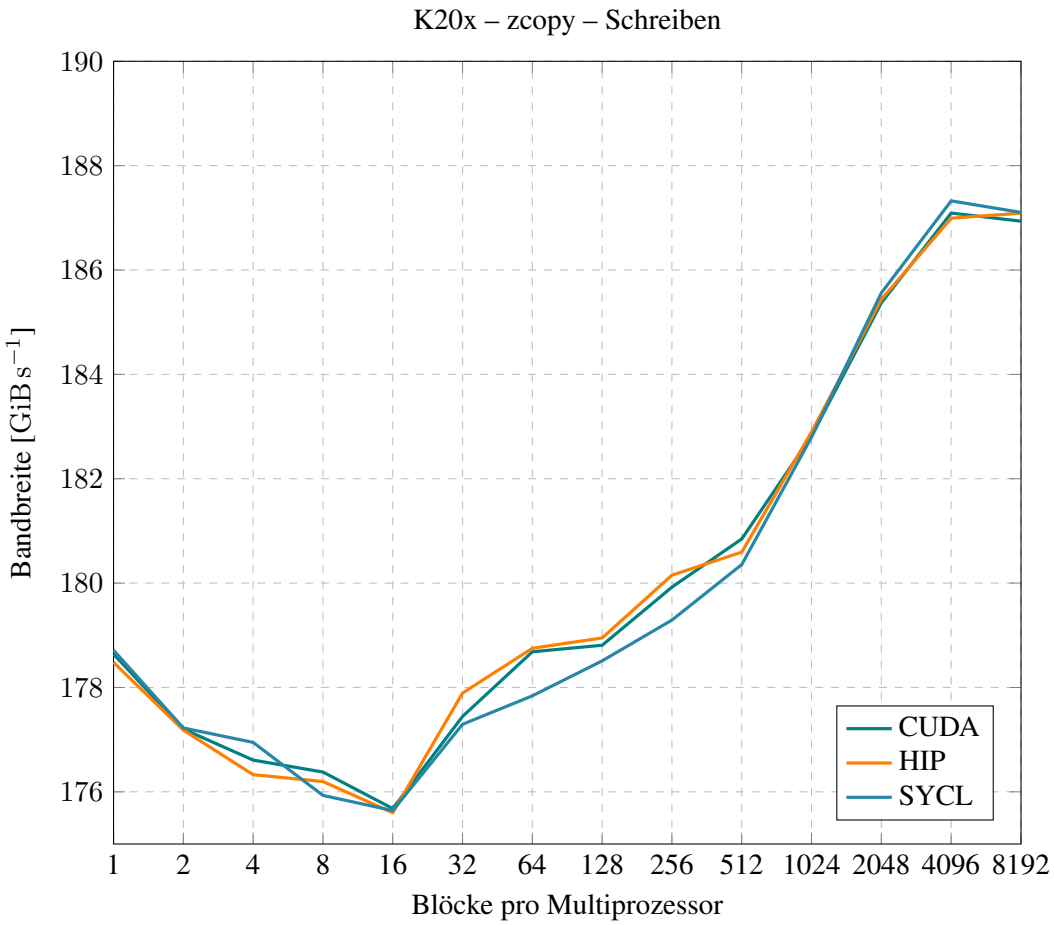


Abbildung 4.7: zcopy: K20x-Bandbreite für 768er-Blöcke ($n = 88080384$, Schreiben)

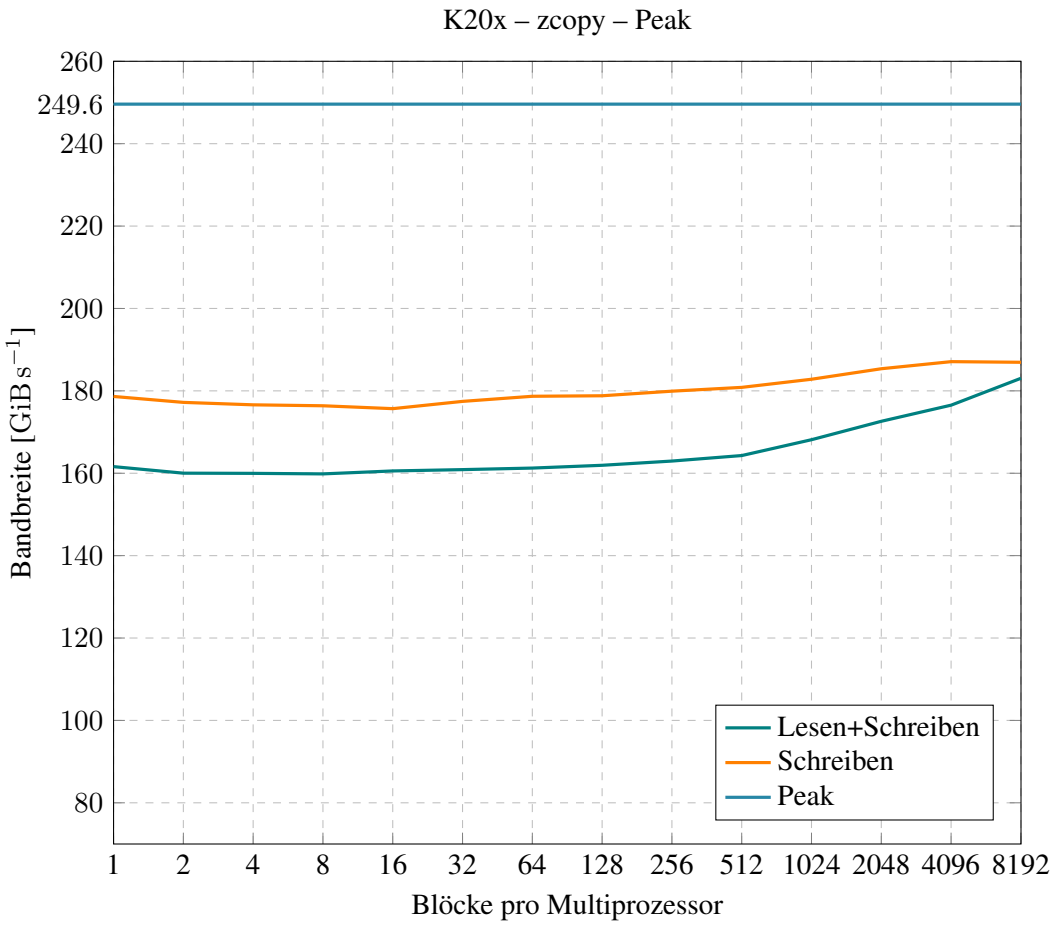


Abbildung 4.8: zcopy: Theoretische und praktische K20x-Bandbreite für 768er-Blöcke ($n = 88080384$)

4.3.2 Messmethoden

Zunächst wurde die zu reduzierende Array-Größe auf die größte in den GPU-Speicher passende Elementezahl festgesetzt. Durch das Ausprobieren mehrerer Block-Größen zwischen 64 und 1.024 einerseits und der Variation der Block-Zahl pro Multiprozessor andererseits wurde dann eine optimierte Kernel-Konfiguration ermittelt.
Im nächsten Schritt wurde die ermittelte optimierte Konfiguration auf Arrays verschiedener Größe ausprobiert, um das Skalierungsverhalten festzustellen.
Jede für den beschriebenen Ablauf nötige Kernel-Ausführung erfolgte zehn Mal hintereinander. Für jede Ausführung wurde über die den Spracherweiterungen zugehörigen Events die Laufzeit ermittelt; die minimale Laufzeit diente als Grundlage für die Berechnung der Bandbreite.
Die Compiler-Flags sind dieselben wie die für den zcopy-Benchmark in Abschnitt 4.2.2 aufgeführten.

4.3.3 Ergebnisse

Der Blick auf die Abbildung 4.9 zeigt, dass 256, 512 und 1.024 Threads die besten Block-Größen bilden. Die selben Ergebnisse zeigen sich für HIP (siehe Abbildung E.16) und SYCL (siehe Abbildung E.17). Die SYCL-Variante zeigt zudem ein effizienteres Verhalten für Blöcke mit 128 Threads. Da 256 Threads über alle Spracherweiterungen hinweg die beste Auswahl an Block-pro-Multiprozessor-Konfigurationen bieten, wurde im weiteren Benchmark-Verlauf diese Größe ausgewählt. Zwischen 8 und 1.024 Blöcken pro Multiprozessor kommen für diese Größe verschiedene Konfigurationen in Frage, die sich für große Arrays nahezu gleichwertig verhalten. Da die nächste Benchmark-Stufe auch weitaus kleinere Arrays umfasst, wurde hier die kleinste sinnvolle Konfiguration gewählt: 8 Blöcke pro Multiprozessor.
Die zcopy-Ergebnisse können hier bestätigt werden: zwischen den Erweiterungen bestehen allenfalls marginale Bandbreitenunterschiede. Zudem kann die tatsächlich erreichbare Bandbreite, die im zcopy-Benchmark ermittelt wurde, auch bei der Reduktion nahezu erreicht werden (siehe Abbildung 4.10).

4.4 N-Body

4.4.1 Implementierung

Die theoretische Funktion 3.3, die die Interaktion zwischen zwei Körpern berechnet, wurde in allen Spracherweiterungen umgesetzt. Durch den Einsatz von *fused multiply-add* (FMA)-Operationen werden die benötigten FLOPs für die Berechnung des Skalarprodukts sowie der Beschleunigung verringert. Die inverse Wurzel wird durch die `rsqrt`-Funktion berechnet. Die Quelltexte A.3 (CUDA), B.3 und D.3 (SYCL) im Anhang dieser Arbeit zeigen die konkrete Implementierung.
Die theoretischen Funktionen 3.4 und 3.5 wurden zusammengefasst, da erstere nur aus einer kurzen Schleife besteht. Überdies wurde der Compiler angewiesen, die Schleife auszurollen (siehe auch den nächsten Abschnitt). Diese Implementierungen finden sich in den angehängten Quelltexten A.4 (CUDA), B.4 (HIP) bzw. D.4 (SYCL).
Ein gesonderter Vergleich ist zwischen CUDA und SYCL nötig, da das experimentelle Backend für NVIDIA-GPUs der ComputeCpp-Implementierung die Funktion `rsqrtf` für die reziproke Quadratwurzel nicht unterstützt. Da die äquivalente Berechnung `1.f / sqrtf` deutlich langsamer ist, wurde stattdessen eine schnellere, weniger genaue Implementierung der Funktion `rsqrtf` aus dem Quelltext

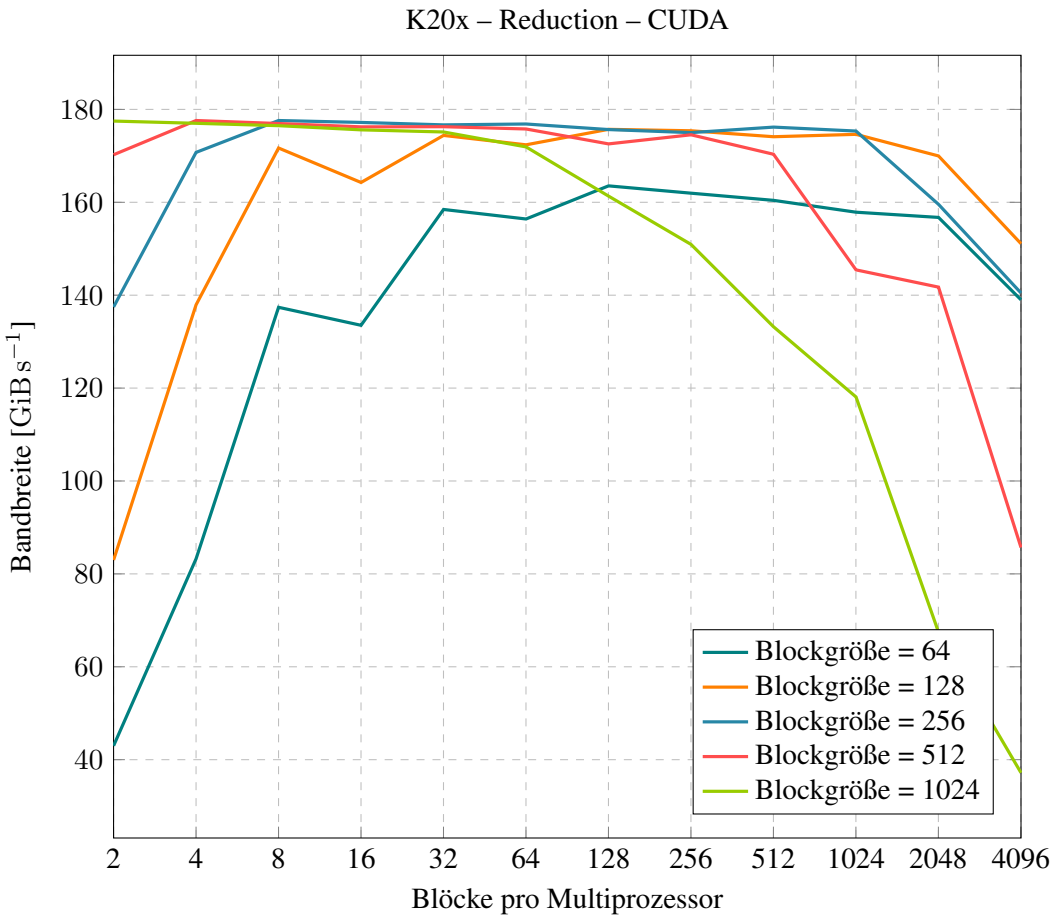


Abbildung 4.9: Reduction: Bandbreite der K20x (CUDA)

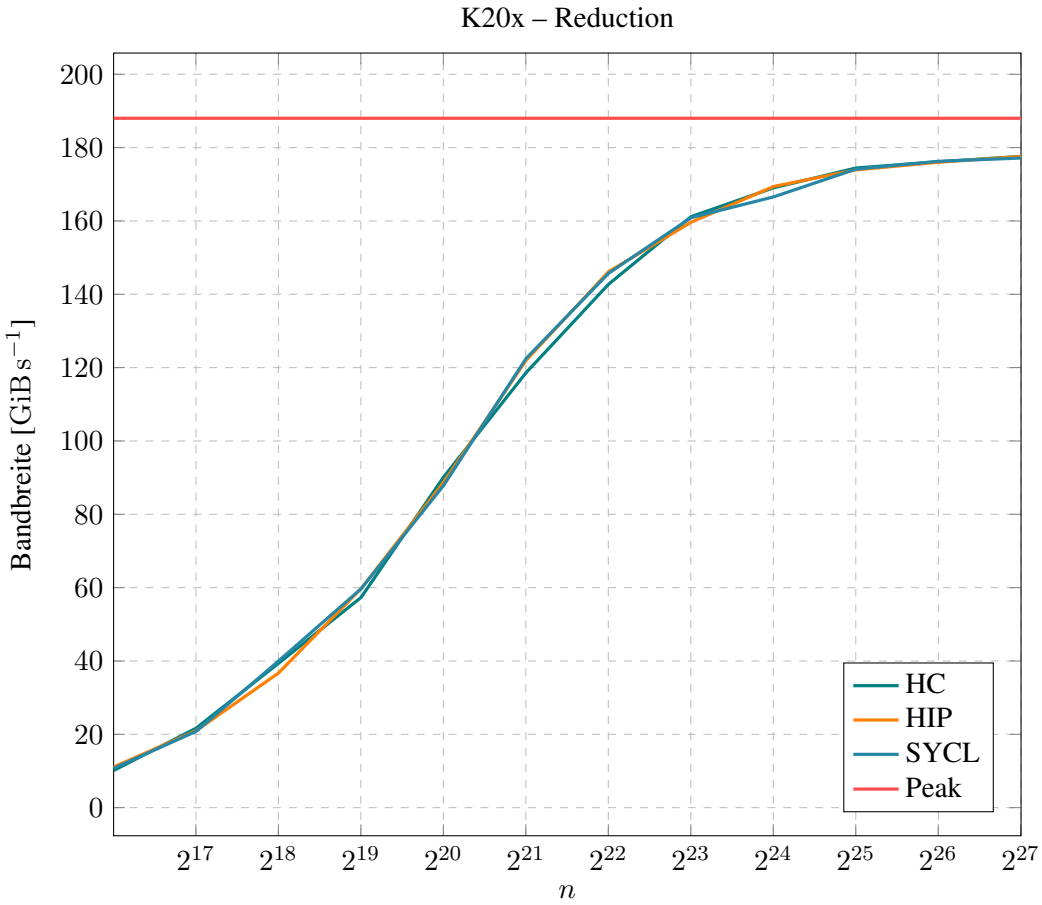


Abbildung 4.10: Reduction: Bandbreite der K20x (acht 256er-Blöcke pro Multiprozessor)

des Spiels *Quake 3 Arena* übernommen (siehe Quelltext 4.2 für eine normale C++-Implementierung).

```
auto Q_rsqrt(float number) -> float
{
    auto x2 = number * 0.5f;
    auto y = number;
    auto i = *(reinterpret_cast<std::int32_t*>(&y));

    i = 0x5f3759df - (i >> 1);

    y = *(reinterpret_cast<float*>(&i));
    y *= 1.5f - (x2 * y * y);

    return y;
}
```

Quelltext 4.2: Quake-3-Implementierung der rsqrt-Funktion

4.4.2 Messmethoden

Der N-Body-Benchmark wurde für jede verwendete Block-Größe über 10 Zeitschritte (i) ausgeführt. Anhand der dafür benötigten Laufzeit t (in Sekunden) lassen sich die erreichten *floating-point operations per second* (FLOPS) nach dem folgenden Schema berechnen:

$$
\begin{aligned}
I &= n \cdot n \cdot i \\
I_s &= \frac{I}{t} \\
\text{FLOPS} &= I_s \cdot 20,
\end{aligned}
$$

wobei I die Zahl der Interaktionen darstellt und I_s die Interaktionen pro Sekunde. Die Berechnung einer Interaktion benötigt 20 FLOPs.
Die Compiler-Flags sind mit denen des zcopy-Benchmarks identisch (siehe Abschnitt 4.2.2).

4.4.3 Optimierung und Auswertung

Eine einfache Optimierung ist das Ausrollen der Schleife, die nacheinander die Interaktionen berechnet. Dadurch erhöht sich zwar der Registerbedarf pro Thread, der mögliche Grad an *instruction-level parallelism* (ILP) wächst aber ebenfalls, da es nun weniger Instruktionen gibt, die von vorherigen Instruktionen abhängig sind. Zudem verringert sich der Overhead, der durch Verzweigungsinstruktionen anfällt.
Diese Effekte sind deutlich in der Abbildung 4.11 zu sehen: durch die Bestimmung eines besseren Ausrollfaktors lassen sich in diesem Benchmark bei einer festen Kachelgröße von $p = 256$ knapp 600 GFLOPS mehr Durchsatz gewinnen. Dieses Verhalten kann auch bei der Implementierung mit HIP (siehe Abbildung E.20) beobachtet werden, nicht jedoch bei SYCL (siehe Abbildung 4.12). Hier ist zu vermuten, dass der Compiler den Ausrollfaktor ignoriert oder nicht zu verarbeiten weiß.

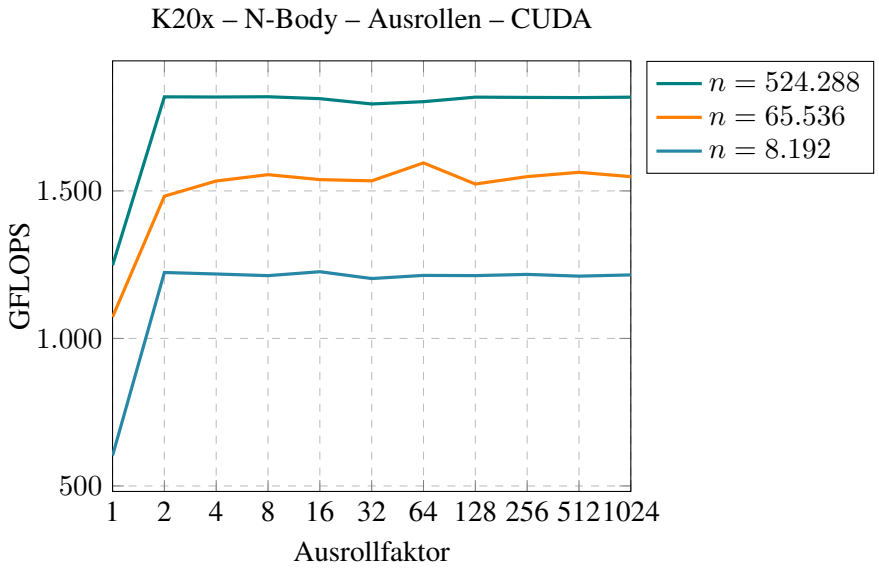


Abbildung 4.11: N-Body: Performanzgewinn durch das Ausrollen der Schleife (CUDA)

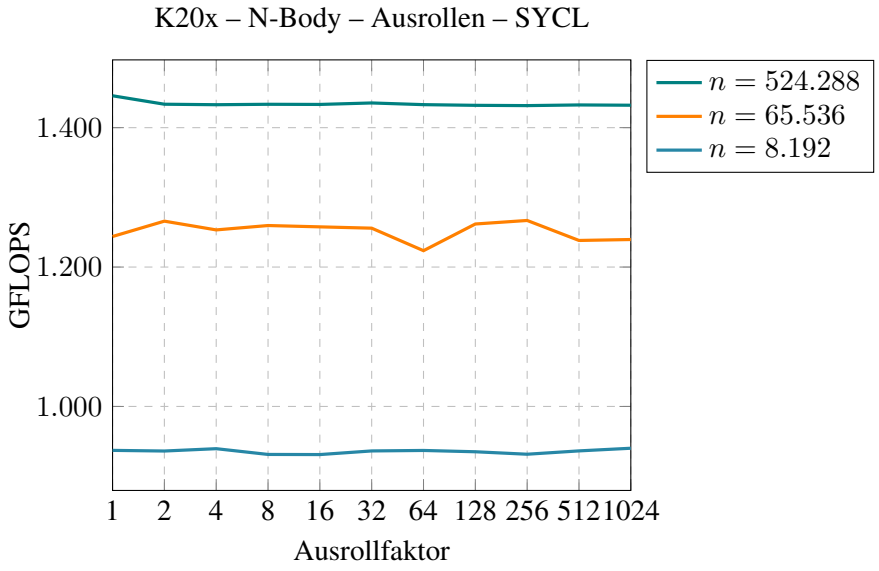


Abbildung 4.12: N-Body: Performanzgewinn durch das Ausrollen der Schleife (SYCL)

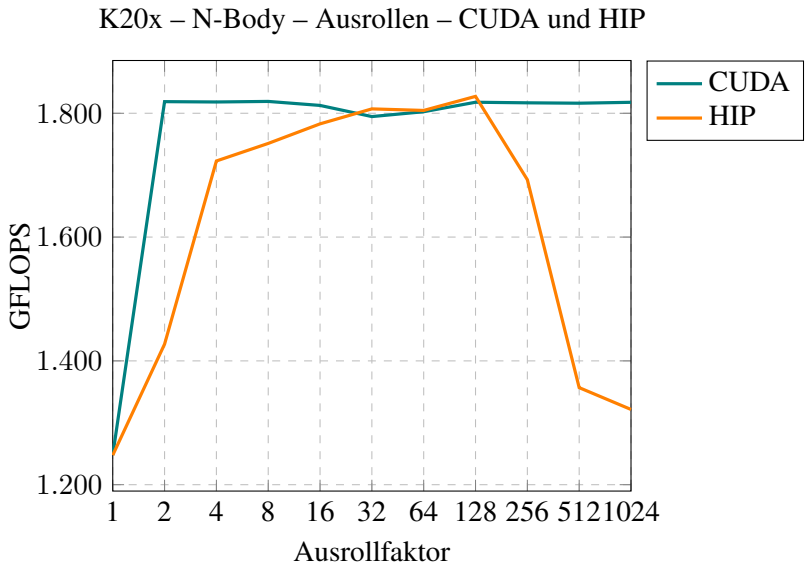


Abbildung 4.13: N-Body: Vergleich der Ausrollfaktoren zwischen CUDA und HIP ($n = 524.288$)

Es ist außerdem festzustellen, dass der Ausrollfaktor auf HIP einen ähnlichen Einfluss wie auf CUDA hat. HIP erreicht jedoch bei kleinen und großen Faktoren weniger FLOPS als CUDA, während mittlere Faktoren ähnliche Ergebnisse erzielen (siehe Abbildung 4.13). Möglicherweise verhindern das Ummanteln des CUDA-API durch das HIP-API oder der Aufruf des `nvcc`-Compilers durch das `hipcc`-Skript einige kleinere Compiler-Optimierungen, die bei nativem CUDA-Code möglich wären.
Interessant ist ebenfalls, dass ein relativ großer Ausrollfaktor von 128 bei HIP noch positive Auswirkungen auf die Performanz hat. Normalerweise wäre durch den hohen Registerverbrauch bereits hier eine schlechtere Leistung zu erwarten, da auf einem Multiprozessor weniger Blöcke gleichzeitig Platz finden. Hier wäre zu untersuchen, inwieweit die Register wiederverwendet werden und wie sich dies auf die Auslastung der Multiprozessoren auswirkt.
Anhand dieser Messung wurde für den weiteren Verlauf ein Ausrollfaktor von 64 festgelegt.
Der nächste performanzrelevante Faktor ist die Größe der Kacheln selbst. Aus den in der Abbildung 4.14 dargestellten Messergebnissen wird ersichtlich, dass die gewählte Kachelgröße auf die Leistung einen erheblichen Einfluss haben kann. Auch dieses Verhalten ist bei HIP (siehe Abbildung E.21) und SYCL (siehe Abbildung E.22) feststellbar. Für den weiteren Messverlauf wird daher eine Kachelgröße von $p = 512$ angenommen.
Mit der experimentell ermittelten Konfiguration lässt sich ein direkter Vergleich zwischen CUDA und HIP einerseits sowie CUDA (mit `Q_rsqrt`) und SYCL andererseits anstellen. Die Abbildung 4.15 zeigt, dass die Performanz bei CUDA und HIP nahezu identisch ist. Der Blick in den generierten Maschinen-Code zeigt, dass der `nvcc`-Compiler und der `hcc`-Wrapper um `nvcc` dasselbe Ergebnis erzeugen (siehe Quelltexte 4.3 und 4.4).
Die Abbildung 4.16 zeigt die Ergebnisse des Vergleichs zwischen CUDA (mit `Q_rsqrt`) und SYCL. Beide Implementierungen erreichen ähnliche Ergebnisse, wobei CUDA zumeist etwas schneller arbeitet. Die Quelltexte 4.5 (CUDA) und 4.6 (SYCL) zeigen außerdem merkliche Unterschiede zwischen den generierten Maschinen-Codes, was möglicherweise CUDAs bessere Ergebnisse erklärt.

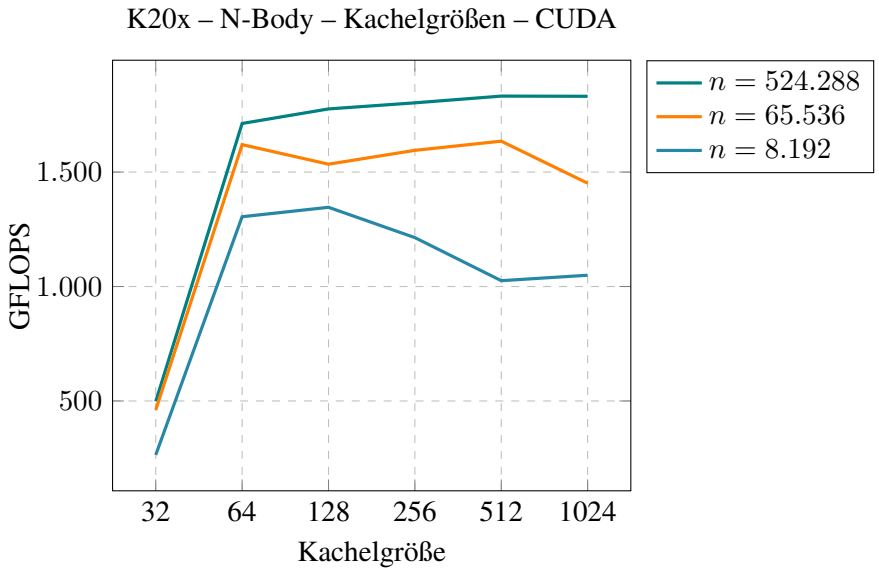


Abbildung 4.14: N-Body: Performanz bei verschiedenen Kachelgrößen (CUDA)

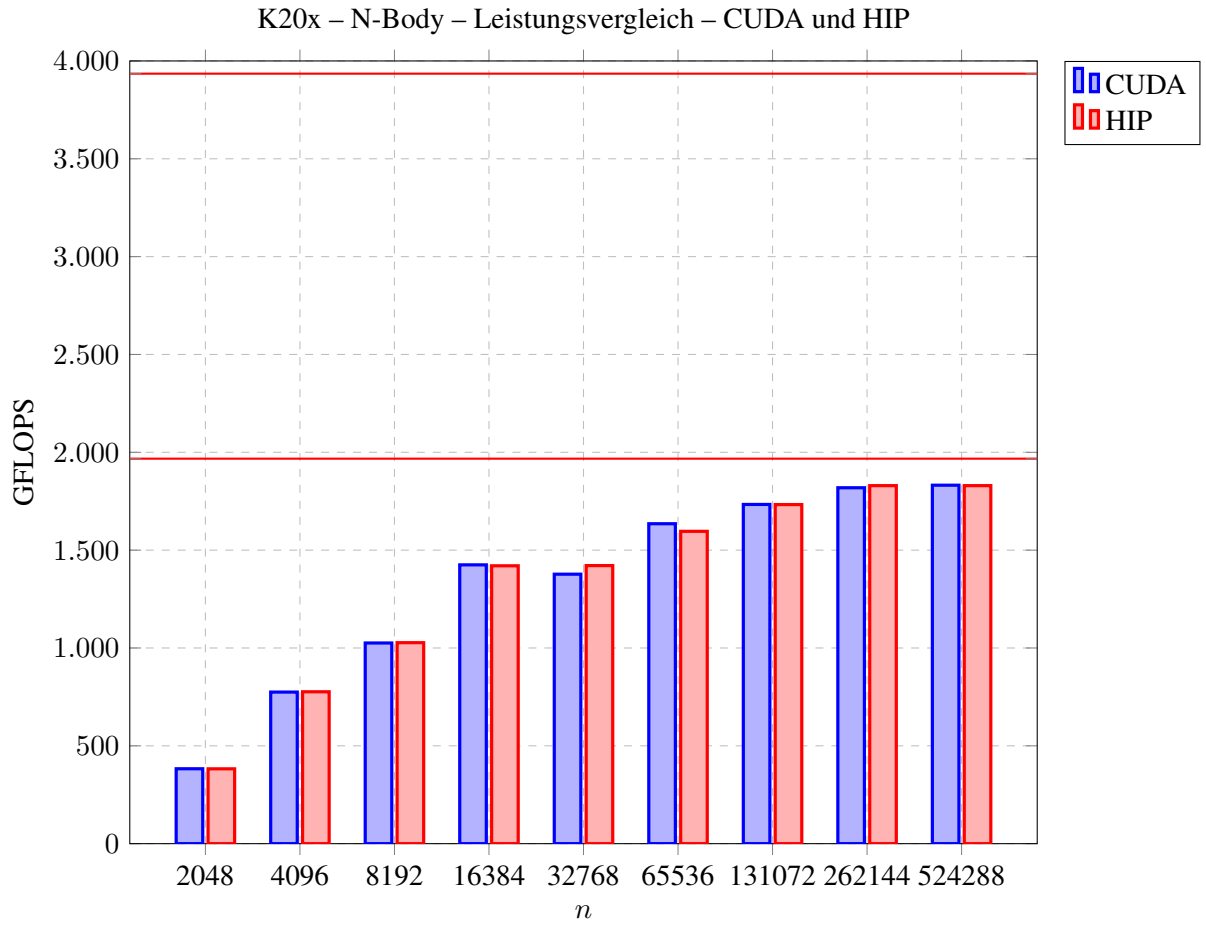


Abbildung 4.15: N-Body: Leistungsvergleich zwischen CUDA und HIP (Peak bei 1.967,5 (ohne FMA) bzw. 3.935 GFLOPS)

```
shl.b32 %r29, %r42, 4;
add.s32 %r31, %r22, %r29;
ld.shared.v4.f32 {%f50, %f51, %f52,
                  %f53}, [%r31];
sub.f32 %f58, %f50, %f35;
sub.f32 %f59, %f51, %f36;
sub.f32 %f60, %f52, %f37;
mov.f32 %f61, 0f358637BE;
fma.rn.f32 %f62, %f60, %f60, %f61;
fma.rn.f32 %f63, %f59, %f59, %f62;
fma.rn.f32 %f64, %f58, %f58, %f63;
mul.f32 %f65, %f64, %f64;
mul.f32 %f66, %f64, %f65;
rsqrt.approx.f32 %f67, %f66;
mul.f32 %f68, %f53, %f67;
fma.rn.f32 %f69, %f58, %f68, %f21590;
fma.rn.f32 %f70, %f59, %f68, %f21589;
fma.rn.f32 %f71, %f60, %f68, %f21588;
```

Quelltext 4.3: N-Body: Maschinencode des CUDA-Kernels

```
shl.b32 %r29, %r42, 4;
add.s32 %r31, %r22, %r29;
ld.shared.v4.f32 {%f50, %f51, %f52,
                  %f53}, [%r31];
sub.f32 %f58, %f50, %f35;
sub.f32 %f59, %f51, %f36;
sub.f32 %f60, %f52, %f37;
mov.f32 %f61, 0f358637BE;
fma.rn.f32 %f62, %f60, %f60, %f61;
fma.rn.f32 %f63, %f59, %f59, %f62;
fma.rn.f32 %f64, %f58, %f58, %f63;
mul.f32 %f65, %f64, %f64;
mul.f32 %f66, %f64, %f65;
rsqrt.approx.f32 %f67, %f66;
mul.f32 %f68, %f53, %f67;
fma.rn.f32 %f69, %f58, %f68, %f21590;
fma.rn.f32 %f70, %f59, %f68, %f21589;
fma.rn.f32 %f71, %f60, %f68, %f21588;
```

Quelltext 4.4: N-Body: Maschinencode des HIP-Kernels

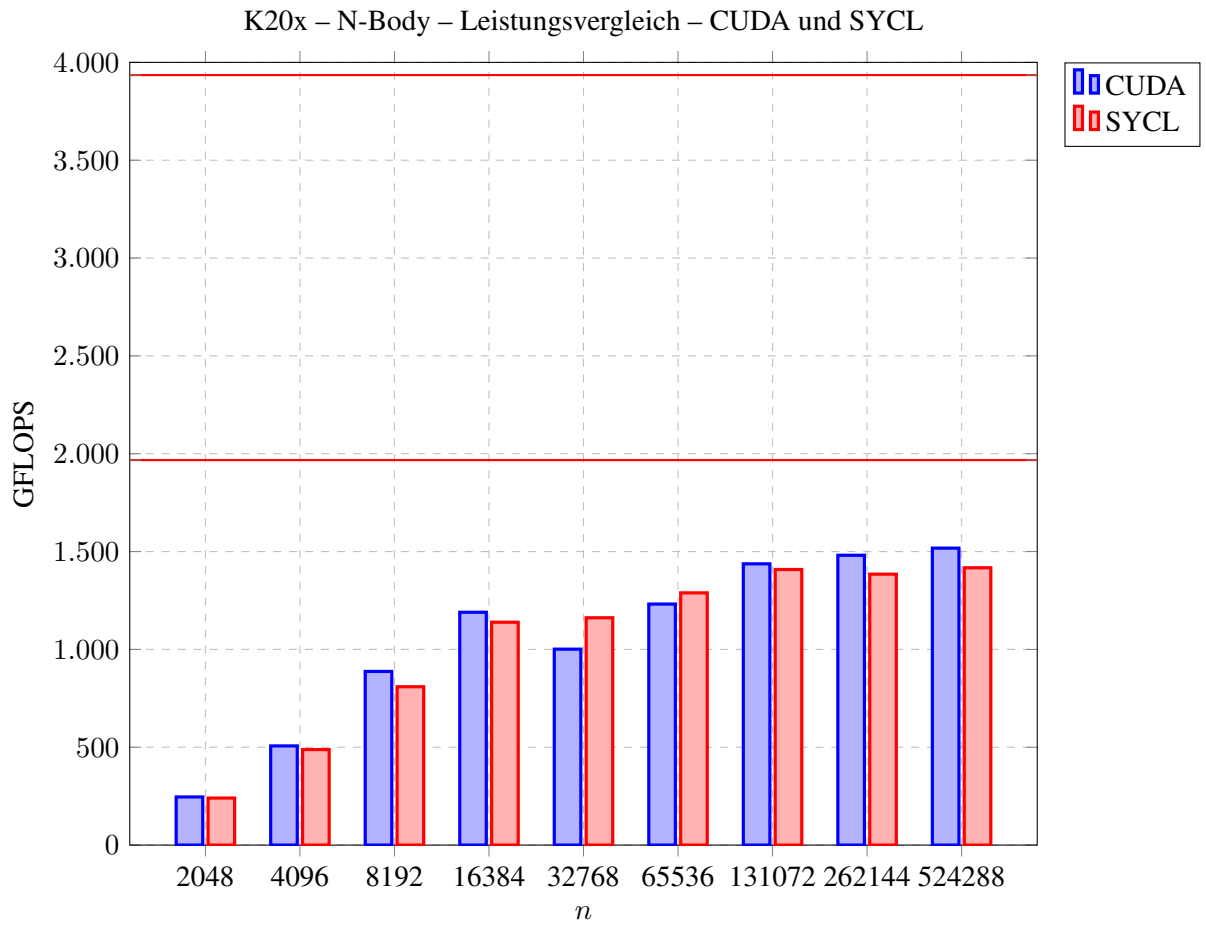


Abbildung 4.16: N-Body: Leistungsvergleich zwischen CUDA (mit Q_rsqrt) und SYCL (Peak bei 1.967,5 (ohne FMA) bzw. 3.935 GFLOPS)

```
ld.shared.v4.f32 {%f50, %f51, %f52,
                  %f53}, [%r31];
sub.f32 %f58, %f50, %f35;
sub.f32 %f59, %f51, %f36;
sub.f32 %f60, %f52, %f37;
mov.f32 %f61, 0f358637BE;
fma.rn.f32 %f62, %f60, %f60, %f61;
fma.rn.f32 %f63, %f59, %f59, %f62;
fma.rn.f32 %f64, %f58, %f58, %f63;
mul.f32 %f65, %f64, %f64;
mul.f32 %f66, %f64, %f65;
mul.f32 %f67, %f66, 0f3F000000;
mov.b32 %r32, %f66;
shr.s32 %r33, %r32, 1;
mov.u32 %r34, 1597463007;
sub.s32 %r35, %r34, %r33;
mov.b32 %f68, %r35;
mul.f32 %f69, %f67, %f68;
mul.f32 %f70, %f68, %f69;
mov.f32 %f71, 0f3FC00000;
sub.f32 %f72, %f71, %f70;
mul.f32 %f73, %f68, %f72;
mul.f32 %f74, %f53, %f73;
fma.rn.f32 %f75, %f58, %f74, %f13403;
fma.rn.f32 %f76, %f59, %f74, %f13402;
fma.rn.f32 %f77, %f60, %f74, %f13401;
```

Quelltext 4.5: N-Body: Maschinencode des CUDA-Kernels (mit Q_rsqrt)

```
ld.shared.v4.f32 {%f33, %f34, %f35,
                  %f36}, [%rd25];
sub.rn.f32 %f37, %f33, %f1;
sub.rn.f32 %f38, %f34, %f2;
sub.rn.f32 %f39, %f35, %f3;
fma.rn.f32 %f40, %f39, %f39,
           0f358637BE;
fma.rn.f32 %f41, %f38, %f38, %f40;
fma.rn.f32 %f42, %f37, %f37, %f41;
mul.rn.f32 %f43, %f42, %f42;
mul.rn.f32 %f44, %f42, %f43;
mul.rn.f32 %f45, %f44, 0fBF000000;
mov.b32 %r13, %f44;
shr.s32 %r14, %r13, 1;
sub.s32 %r16, %r15, %r14;
mov.b32 %f46, %r16;
mul.rn.f32 %f47, %f45, %f46;
mul.rn.f32 %f48, %f47, %f46;
add.rn.f32 %f49, %f48, 0f3FC00000;
mul.rn.f32 %f50, %f49, %f46;
mul.rn.f32 %f51, %f36, %f50;
fma.rn.f32 %f76, %f39, %f51, %f76;
fma.rn.f32 %f75, %f38, %f51, %f75;
fma.rn.f32 %f74, %f37, %f51, %f74;
```

Quelltext 4.6: N-Body: Maschinencode des SYCL-Kernels

5 Messungen auf AMD-GPUs

5.1 Verwendete Hard- und Software

Die hier gezeigten Benchmark-Ergebnisse wurden auf einem Rechner mit der folgenden Hardware gemessen:

- CPU: AMD Ryzen Threadripper 1950X
  - 16 Kerne
  - 32 virtuelle Kerne
  - maximaler Takt: $3{,}4\,\mathrm{GHz}$
- GPU: AMD Radeon RX Vega 64
  - 64 Multiprozessoren
  - 64 Kerne pro Multiprozessor (insgesamt 4.096 Kerne)
  - maximaler Takt: $1.536\,\mathrm{MHz}$
  - $8\,\mathrm{GiB}$ HBM2-Speicher
  - Speicherbusbreite: $2.048\,\mathrm{bit}$
  - Speicherbandbreite: $483{,}3\,\mathrm{GiB\,s^{-1}}$
  - Speichertakt: $945\,\mathrm{MHz}$
  - kein ECC
- RAM: $64\,\mathrm{GiB}$

Das verwendete Betriebssystem war Ubuntu 16.04 mit der Linux-Kernel-Version 4.15. Für die GPGPU-Programmierung kamen die mit der ROCm-Version 2.1.96 mitgelieferten Implementierungen von HIP und HC zum Einsatz.

5.2 zcopy

5.2.1 Vorüberlegungen

Die in Abschnitt 4.2.1 für NVIDIA-GPUs gemachten Überlegungen können nicht ohne Anpassungen für AMD-GPUs übernommen werden, da sich die Multiprozessor-Architekturen unterscheiden.
Jeder Multiprozessor einer Vega-GPU besteht aus vier SIMD-Einheiten. Eine SIMD-Einheit verfügt über 16 Vektor-ALUs (VALUs), die synchron auf jeweils einem *work-item* arbeiten. Die vier SIMD-Einheiten arbeiten parallel eine *wavefront* ab, die 64 *work-items* umfasst.
Anders ausgedrückt kann jeder Multiprozessor genau 64 Threads abarbeiten, wobei diese auf dem Multiprozessor noch einmal in Gruppen von 16 Threads aufgeteilt werden. Ausgehend von NVIDIAs *half-warps* könnte man diese Gruppen auch als *quarter-wavefronts* bezeichnen, wenngleich dieser Begriff offiziell nicht existiert.
Die Multiprozessoren verfügen über einen L1-Cache, der eine Cacheline-Größe von 64 Byte aufweist. Um jede SIMD-Einheit effizient zu befüllen, muss jeder Thread genau 4 Bytes lesen, was einem `float`

entspräche. Um den Speicher-Controller besser auszulasten, wird dieser Wert auf 8 Byte pro Thread erhöht (Datentyp `hc::short_vector::float_2`). Die transportierte Datenmenge pro Wavefront beträgt somit $8 \cdot 64 = 512$ Bytes und entspricht damit der der CUDA-Variante ($16 \cdot 32 = 512$), bei der ein `float4`-Vektor verwendet wurde.

5.2.2 Messmethoden

Die Messmethoden entsprechen dem in Abschnitt 4.2.2 für NVIDIA-GPUs geschilderten Vorgehen.
Die im Quelltext 5.1 zeigen die verwendeten Compiler-Flags sowie die Festsetzung der Multiprozessor-Taktrate.

```
# HC-Compiler
hcc `hcc-config --cxxflags --ldflags` -O3 -std=c++17 \
  -amdgpu-target=gfx900

# HIP-Compiler (-amdgpu-target wird nicht unterstützt)
hipcc -O3 -std=c++17

# Taktrate
rocm-smi --setsclk 7
```

Quelltext 5.1: Compiler-Flags und Taktrate für zcopy

5.2.3 Ergebnisse

Wie das Ergebnis für den kombinierten Lese- und Schreibvorgang zeigt (siehe Abbildung 5.1), stagniert die Bandbreite der HC-Kernel mit steigender Tile-Zahl. Auch scheint eine mittlere Anzahl an Threads pro Tile vorteilhaft sein.
Ein interessanter Effekt ist für den reinen Schreibvorgang zu beobachten (siehe Abbildung 5.2): Während Tiles mit 64 oder 256 Threads eine nahezu konstante Bandbreite ermöglichen, brechen alle anderen Größen stark ein, bevor sie sich mit steigender Tile-Zahl wieder in die Richtung des Maximums bewegen. Eine Erklärung lässt sich dafür nur schwer finden, da es keine auf den ersten Blick schlüssige Begründung für das Verhalten der 128er-Tiles gibt, die sich mit dem Verhalten der großen Tiles in Einklang bringen lässt. Denkbar ist ein Fehler im Scheduler oder eine in dieser Konfiguration schlechte Auslastung des Speichercontrollers; hier wären weitergehende Untersuchungen erforderlich.
Die HIP-Variante verhält sich wie die HC-Implementierung, die Ergebnisse sind dieser Arbeit in Abschnitt E.1 angehängt.
Ein Leistungsunterschied besteht in der erreichten Bandbreite: HIP ist hier messbar schneller, wenn auch nur um wenige $\mathrm{GB\,s^{-1}}$. Dies ist sowohl für den kombinierten Lese- und Schreib-Kernel als auch den reinen Schreibvorgang der Fall, wie die Abbildungen 5.3 und 5.4 zeigen.
Wie die Abbildung 5.5 zeigt, lassen sich etwa 90% der theoretisch möglichen Bandbreite auch praktisch nutzen.

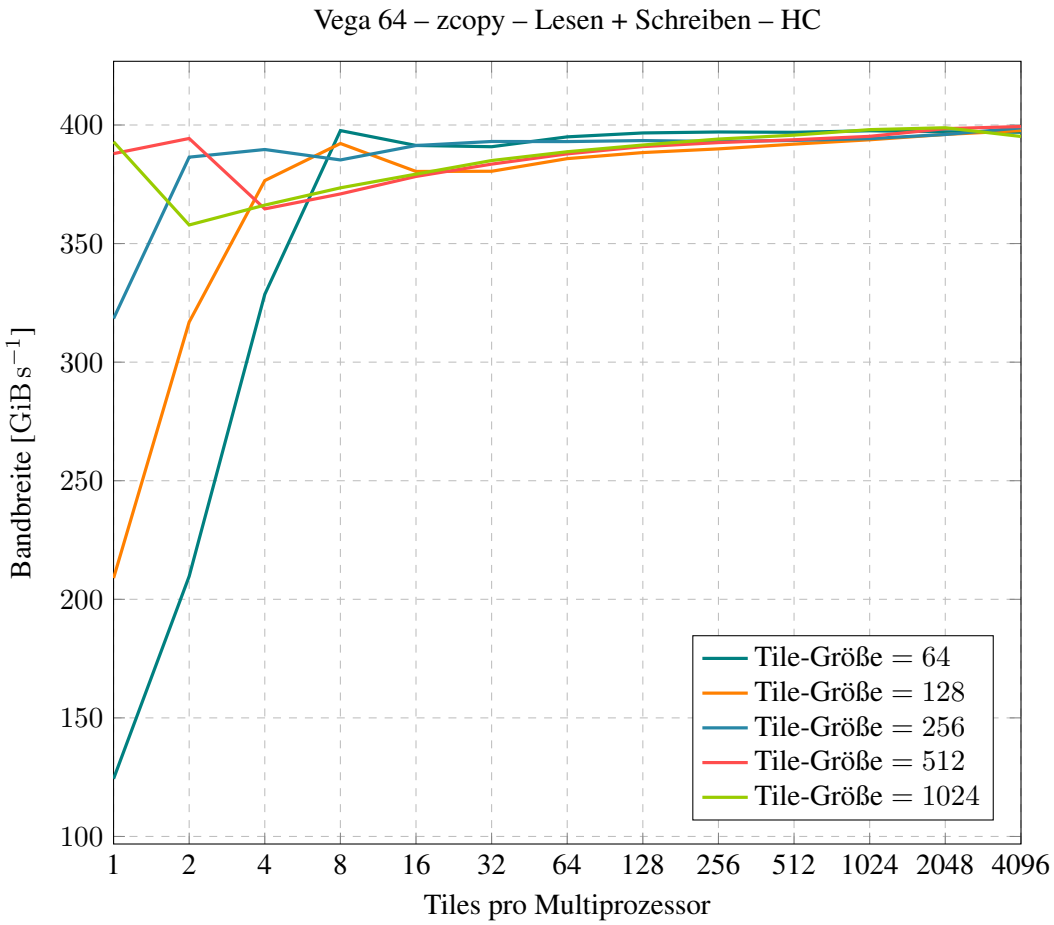


Abbildung 5.1: zcopy: Bandbreite der Vega 64 ($n = 268435456$, Lesen und Schreiben, HC)

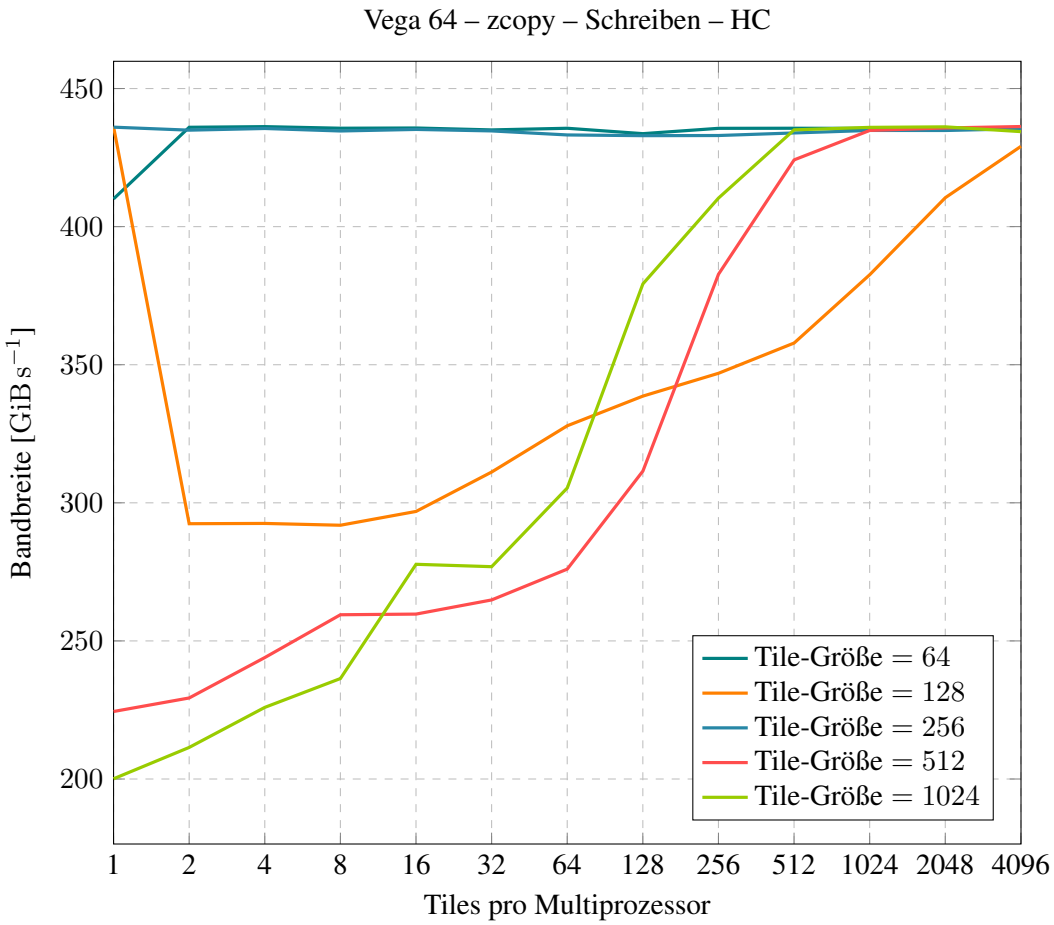


Abbildung 5.2: zcopy: Bandbreite der Vega 64 ($n = 268435456$, Schreiben, HC)

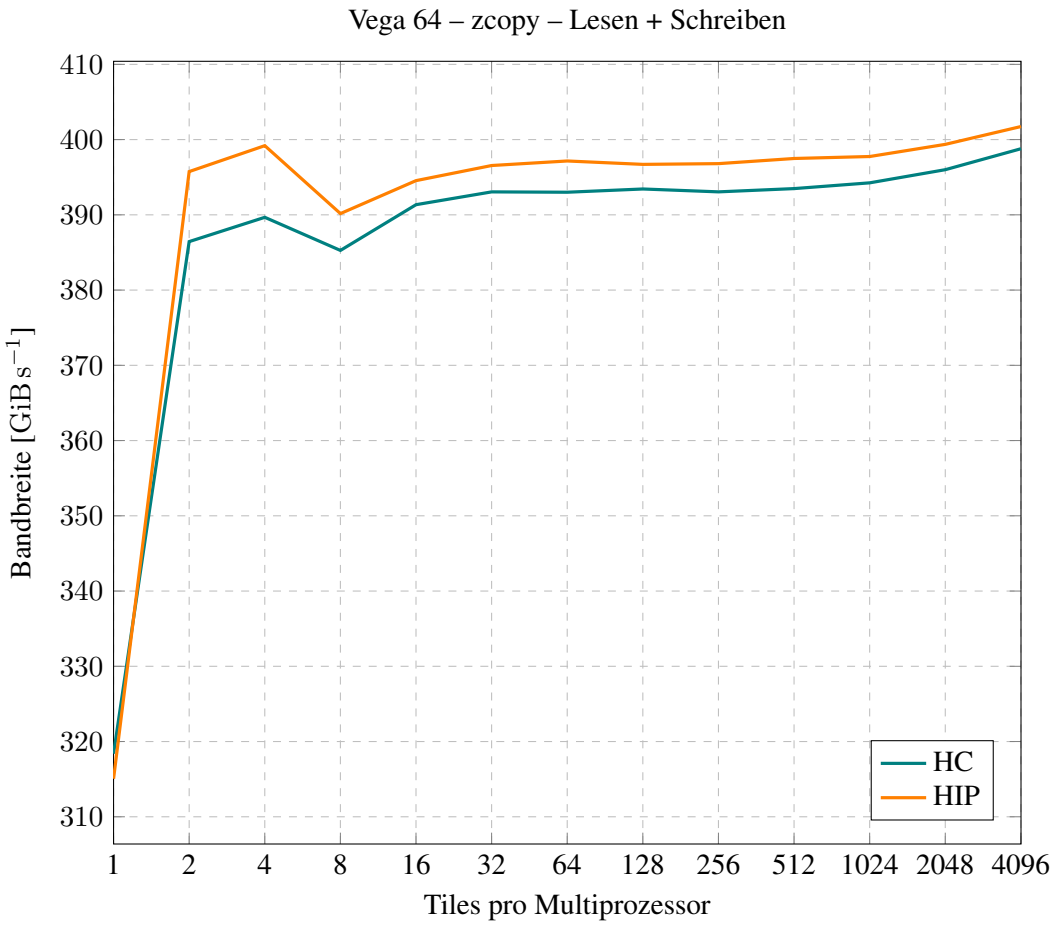


Abbildung 5.3: zcopy: Bandbreite der Vega 64 ($n = 268435456$, 256er-Blöcke, Lesen+Schreiben)

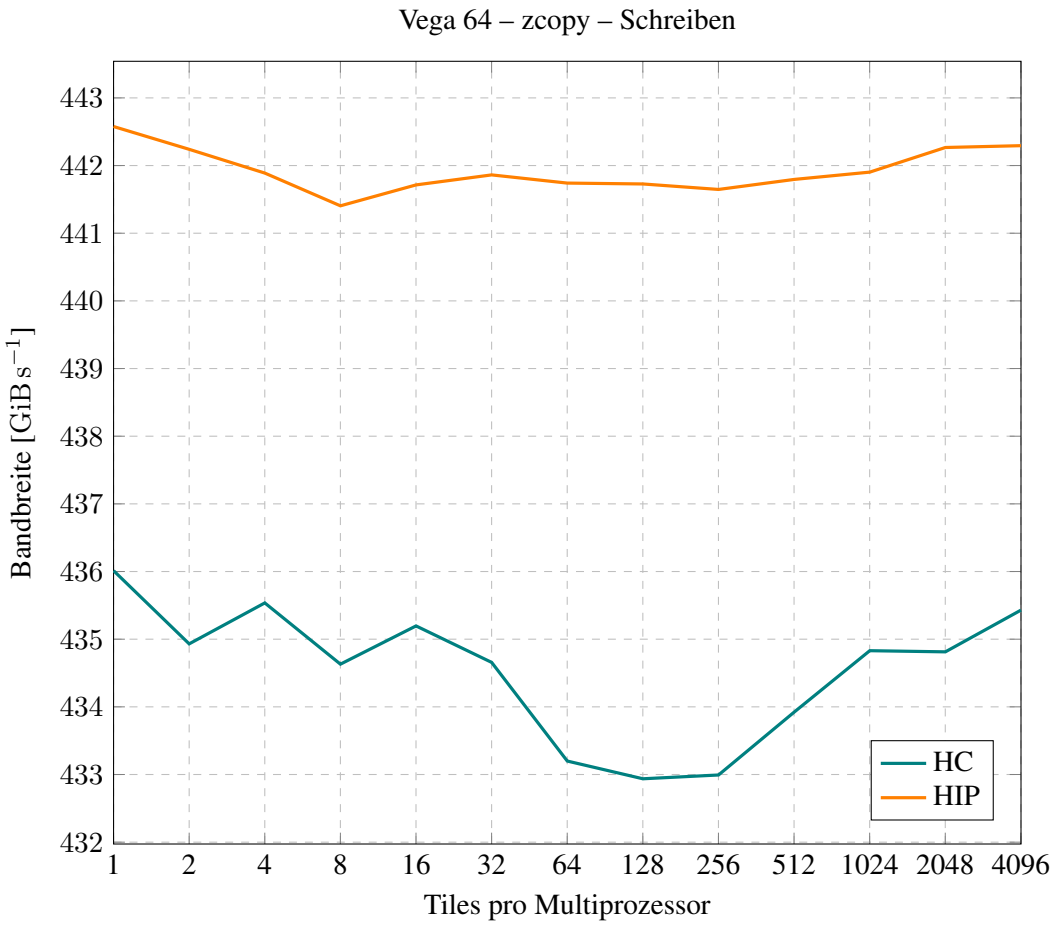


Abbildung 5.4: zcopy: Bandbreite der Vega 64 ($n = 268435456$, 256er-Blöcke, Schreiben)

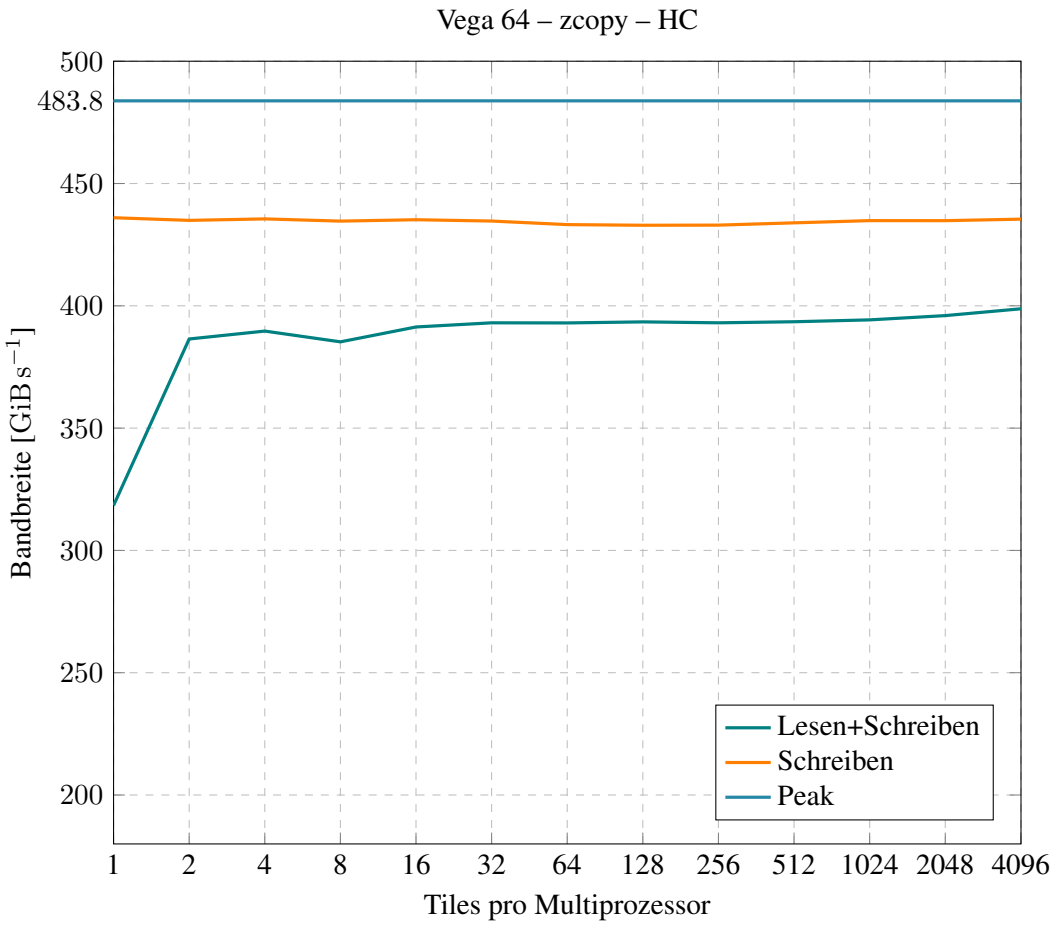


Abbildung 5.5: zcopy: Theoretische und praktische Bandbreite der Vega 64 ($n = 268435456$, HC, 256er-Tiles)

5.3 Reduction

5.3.1 Implementierung

Die Implementierungen des Reduktionskernels sind dieser Arbeit angehängt und befinden sich in den Quelltexten C.2 (HC) und B.2 (HIP).

5.3.2 Messmethoden

Die Messmethoden entsprechen den in Abschnitt 4.3.2 für die NVIDIA-Implementierungen geschilderten, die Compiler-Flags und GPU-Einstellungen sind dieselben wie für den zcopy-Benchmark (siehe Abschnitt 5.2.2).

5.3.3 Ergebnisse

Wie in der Abbildung 5.6 zu sehen ist, sind 256 Threads die beste Tile-Größe der HC-Implementierung, während zwei Tiles dieser Größe die beste Anzahl pro Multiprozessor darstellen. Dies gilt auch für HIP (siehe Abbildung E.15).
Im direkten Vergleich der Spracherweiterungen wird sichtbar, dass sich die zcopy-Ergebnisse bei der Reduktion bestätigen lassen. Wieder erreichen HC und HIP eine ähnliche Bandbreite, wobei HIP konstant und messbar bessere Werte erreicht. Beide Implementierungen kommen außerdem sehr nah an die tatsächlich erreichbare Bandbreite heran, die im zcopy-Benchmark ermittelt wurde (siehe Abbildung 5.7).

5.4 N-Body

5.4.1 Implementierung

Die Implementierungen in HC und HIP folgen denselben Prinzipien, die für die Implementierungen für NVIDIA-GPUs in Abschnitt 4.4.1 aufgeführt wurden.

5.4.2 Messmethoden

Auch die Messmethoden entsprechen denen für NVIDIA-GPUs (siehe Abschnitt 4.4.2). Die Compiler-Flags und GPU-Einstellungen sind mit denen für den zcopy-Benchmark identisch (siehe Abschnitt 5.2.2).

5.4.3 Optimierung und Auswertung

Der durch das Ausrollen der Schleifen erzielte Effekt ist deutlich in den Abbildungen 5.8 und E.18 zu sehen: durch die Bestimmung eines besseren Ausrollfaktors lassen sich in diesem Benchmark bei einer festen Kachelgröße von $p = 256$ knapp 1.000 GFLOPS mehr Durchsatz gewinnen. Anhand dieser Messung wurde für den weiteren Verlauf ein Ausrollfaktor von 8 festgelegt.
Der nächste performanzrelevante Faktor ist die Größe der Kacheln selbst. Aus den in den Abbildungen 5.9 und E.19 dargestellten Messergebnissen wird ersichtlich, dass die Kachelgröße für den Benchmark weniger wichtig ist; relevante Unterschiede sind nur bei großen Kachelgrößen und wenigen Elementen sichtbar. Das ist ein wesentlicher Unterschied zu den in Abschnitt 4.4.3 für NVIDIA-GPUs gemessenen Ergebnissen, die von der Kachelgröße stark beeinflusst wurden. Für den weiteren Messverlauf wird daher eine Kachelgröße von $p = 256$ angenommen.

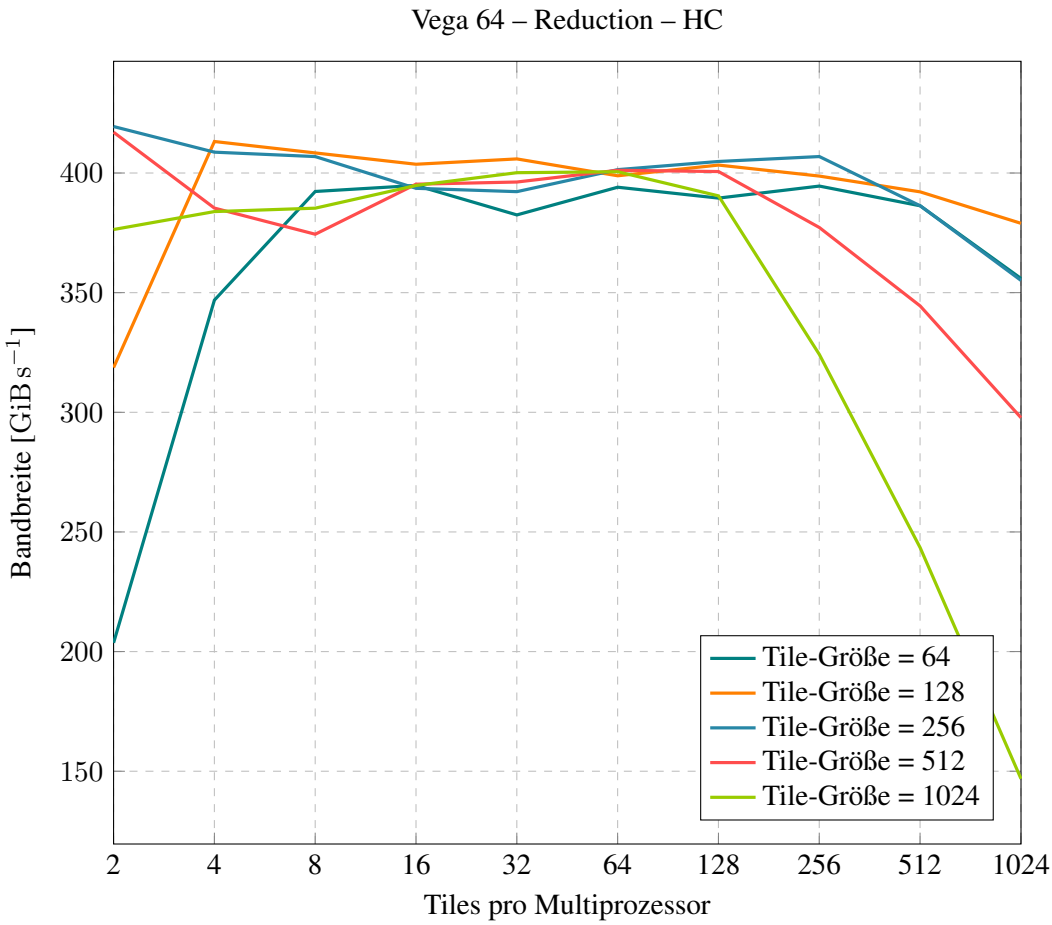


Abbildung 5.6: Reduction: Bandbreite der Vega 64 (HC)

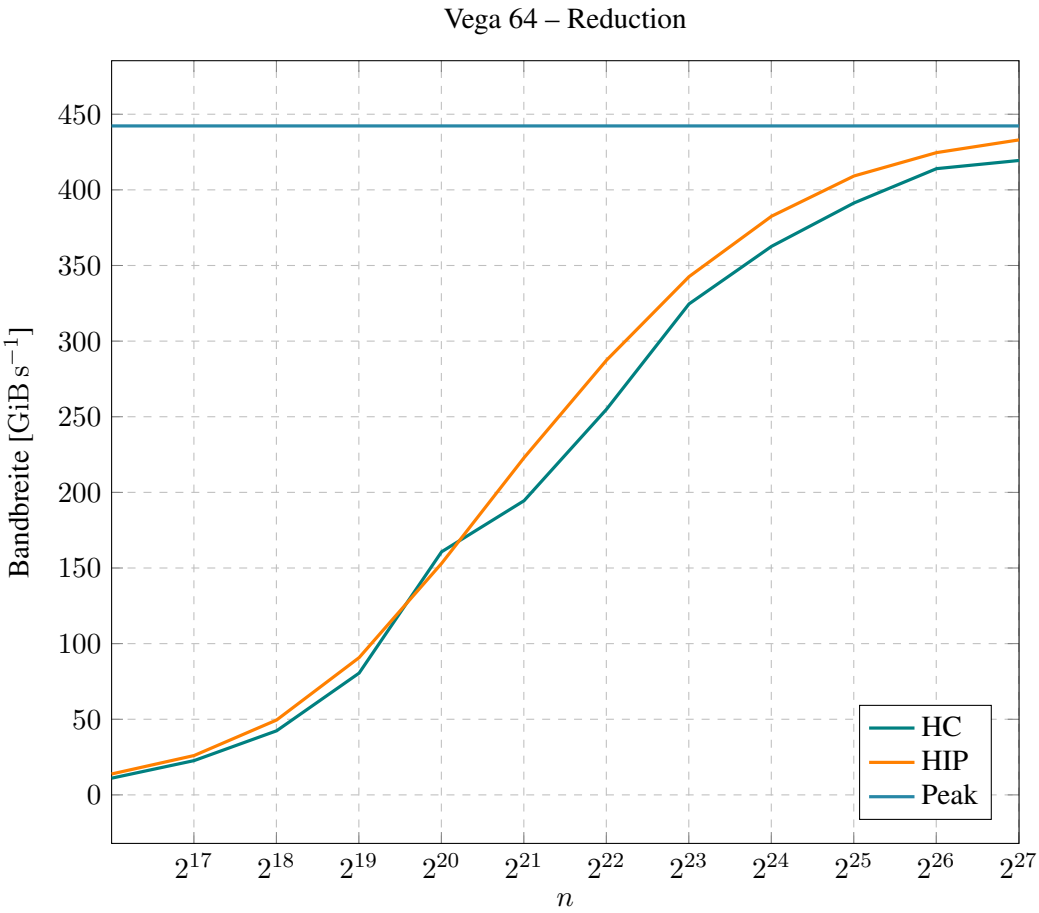


Abbildung 5.7: Reduction: Bandbreite der Vega 64 (zwei 256er-Tiles pro Multiprozessor)

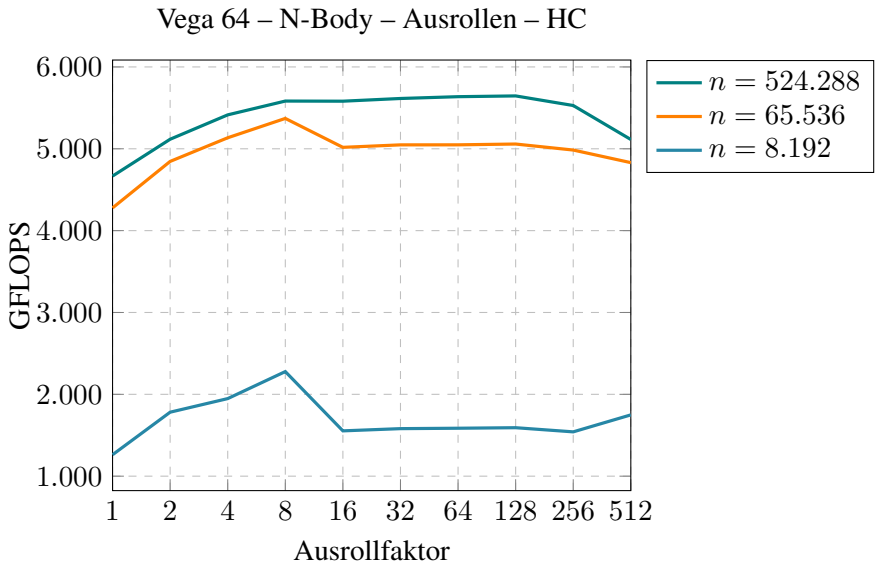


Abbildung 5.8: N-Body: Performanzgewinn durch das Ausrollen der Schleife (HC)

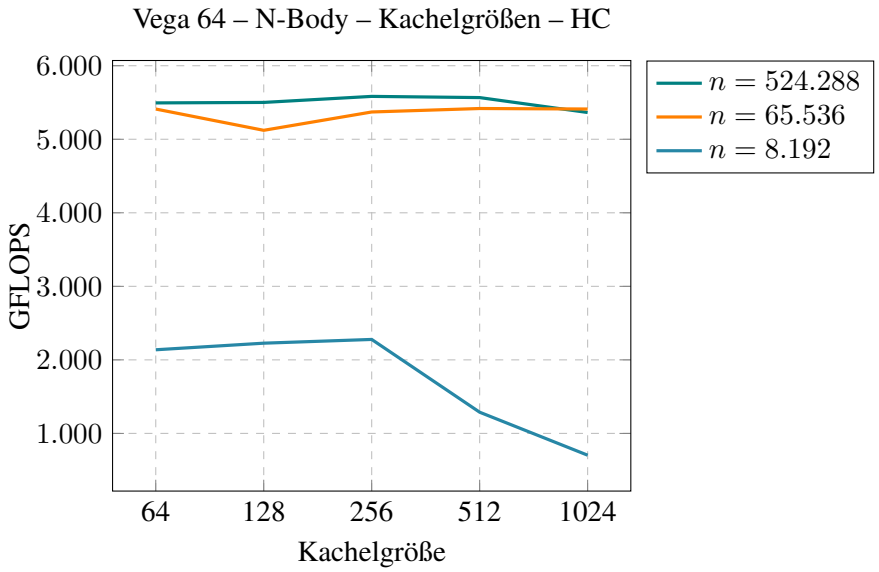


Abbildung 5.9: N-Body: Performanz bei verschiedenen Kachelgrößen (HC)

Mit der experimentell ermittelten Konfiguration lässt sich ein direkter Vergleich zwischen HC und HIP anstellen. Die Abbildung 5.10 zeigt, dass die Performanz bei beiden APIs nahezu identisch ist. Der Blick in den generierten Maschinen-Code zeigt, dass der `hcc`-Compiler in der Lage ist, für beide Varianten ein identisches Ergebnis zu erzeugen (siehe Quelltexte 5.2 und 5.3).

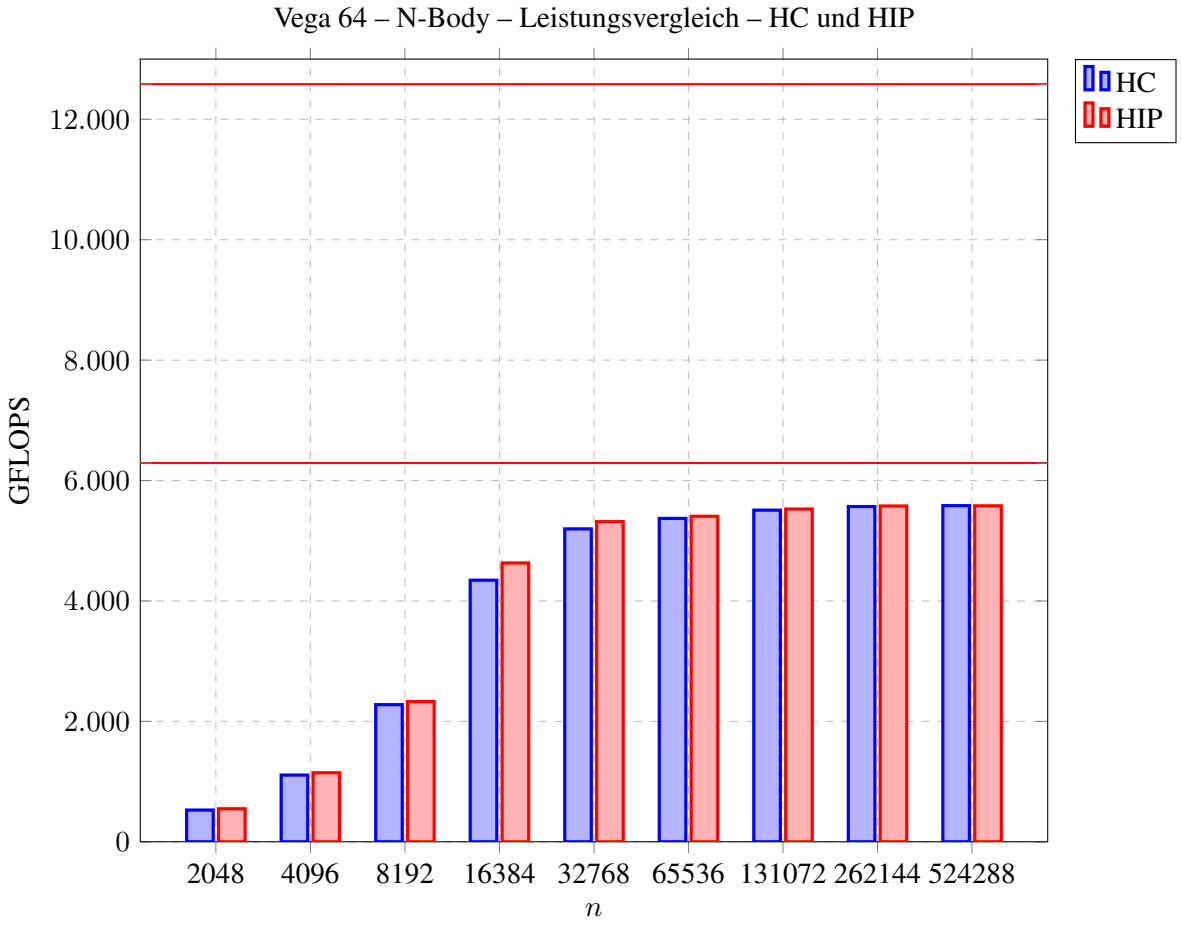


Abbildung 5.10: N-Body: Leistungsvergleich zwischen HC und HIP (Peak bei 6.291,5 (ohne FMA) bzw. 12.583 GFLOPS)

```
ds_read2_b64 v[14:17], v10 offset1:1
v_add_u32_e32 v9, 64, v9
s_waitcnt lgkmcnt(0)
v_sub_f32_e32 v16, v16, v5
v_sub_f32_e32 v15, v15, v4
v_fma_f32 v18, v16, v16, s20
v_sub_f32_e32 v14, v14, v3
v_fma_f32 v18, v15, v15, v18
v_fma_f32 v18, v14, v14, v18
v_mul_f32_e32 v19, v18, v18
v_mul_f32_e32 v18, v18, v19
v_cmp_gt_f32_e32 vcc, s21, v18
v_mov_b32_e32 v19, s22
v_cndmask_b32_e32 v20, 1.0, v19, vcc
v_mul_f32_e32 v18, v18, v20
v_rsq_f32_e32 v18, v18
v_mov_b32_e32 v20, s23
v_cndmask_b32_e32 v21, 1.0, v20, vcc
v_mul_f32_e32 v18, v21, v18
v_mul_f32_e32 v17, v17, v18
v_fma_f32 v18, v14, v17, v11
v_fma_f32 v15, v15, v17, v12
v_fma_f32 v16, v16, v17, v13
```

Quelltext 5.2: N-Body: Maschinencode des HC-Kernels

```
ds_read2_b64 v[14:17], v9 offset1:1
v_add_u32_e32 v10, 64, v10
s_waitcnt lgkmcnt(0)
v_sub_f32_e32 v16, v16, v5
v_sub_f32_e32 v15, v15, v4
v_fma_f32 v18, v16, v16, s16
v_sub_f32_e32 v14, v14, v3
v_fma_f32 v18, v15, v15, v18
v_fma_f32 v18, v14, v14, v18
v_mul_f32_e32 v19, v18, v18
v_mul_f32_e32 v18, v18, v19
v_cmp_gt_f32_e32 vcc, s17, v18
v_mov_b32_e32 v19, s18
v_cndmask_b32_e32 v20, 1.0, v19, vcc
v_mul_f32_e32 v18, v18, v20
v_rsq_f32_e32 v18, v18
v_mov_b32_e32 v20, s19
v_cndmask_b32_e32 v21, 1.0, v20, vcc
v_mul_f32_e32 v18, v21, v18
v_mul_f32_e32 v17, v17, v18
v_fma_f32 v18, v14, v17, v11
v_fma_f32 v15, v15, v17, v12
v_fma_f32 v16, v16, v17, v13
```

Quelltext 5.3: N-Body: Maschinencode des HIP-Kernels

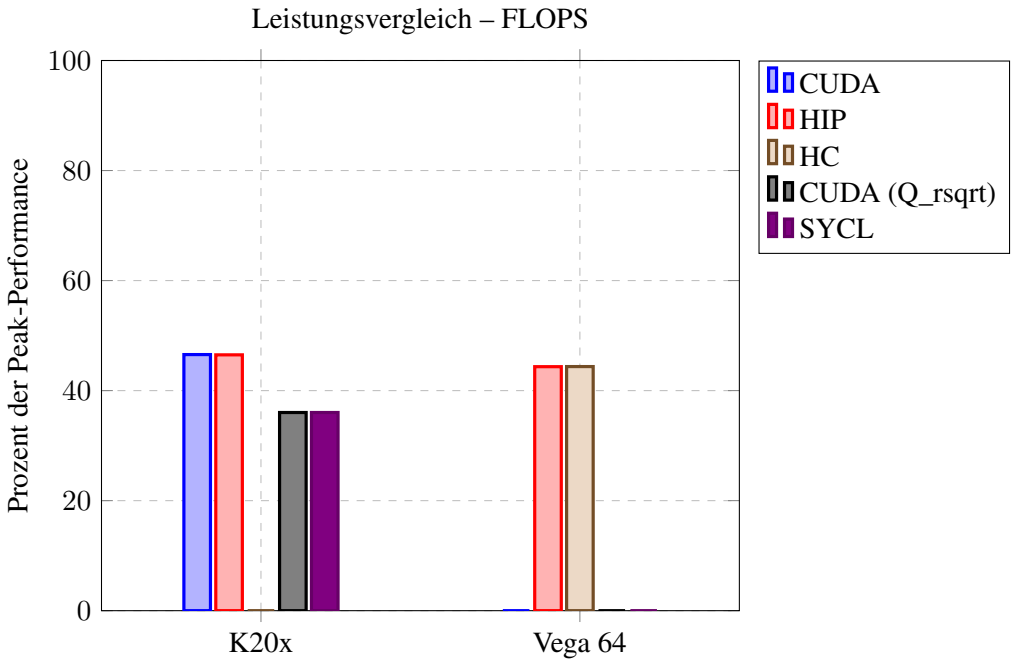


Abbildung 6.1: Prozentualer Leistungsvergleich (FLOPS)

6 Fazit

6.1 Zusammenfassung und Empfehlungen

Es wurde gezeigt, dass die Spracherweiterungen sich in ihrer Mächtigkeit recht ähnlich und auf derselben Hardware nicht signifikant besser oder schlechter als die Alternativen sind. Auch plattformübergreifend fallen die Unterschiede bei den prozentual erreichbaren FLOPS (siehe Abbildung 6.1) und Bandbreiten (siehe Abbildung 6.2) gering aus, lediglich HIP auf der Vega 64 erreicht hier geringfügig bessere Werte. CUDA ist angesichts seiner gegenwärtigen dominanten Stellung und der langjährigen Entwicklung des Ökosystems für NVIDIA-GPUs nahezu alternativlos. Gleichwohl ist mit SYCL ein interessanter Wettbewerber aufgetreten, der viele gute Ansätze von CUDA übernimmt und um eigene Techniken erweitert. Sofern der SYCL-Standard durch Hardware- und Software-Hersteller die notwendige Unterstützung erfährt, kann er zu einer guten, plattformübergreifenden Alternative zu CUDA werden und verdient daher weitere Untersuchungen.
Der Einsatz von HC an Stelle von HIP, wenn AMD-GPUs die einzige unterstützte Hardware-Plattform sein sollen, bietet zur Zeit keine Vorteile. Beide Spracherweiterungen erreichen ähnliche Ergebnisse auf derselben Hardware, HIP ist darüber hinaus einfacher auf NVIDIA-GPUs portierbar und verfügt mit den `hipStreams` über ein besseres System, mit dem sich Aufgabengraphen implementieren lassen. Der einzige Punkt, der für HC spricht, ist das moderne C++-Interface.

6.2 Ausblick

Weitere Aspekte der Spracherweiterungen, die in dieser Arbeit nicht behandelt wurden, bedürfen noch der Untersuchung. Insbesondere das Feld der Multi-GPU-Programmierung verdient mehr Beachtung. Der SYCL-Standard kennt z.B. das Konzept von Peer-to-Peer-Kopien zwischen direkt verbundenen GPUs nicht. Hier wäre zu prüfen, inwieweit die vorhandenen Implementierungen von dieser Fähig-

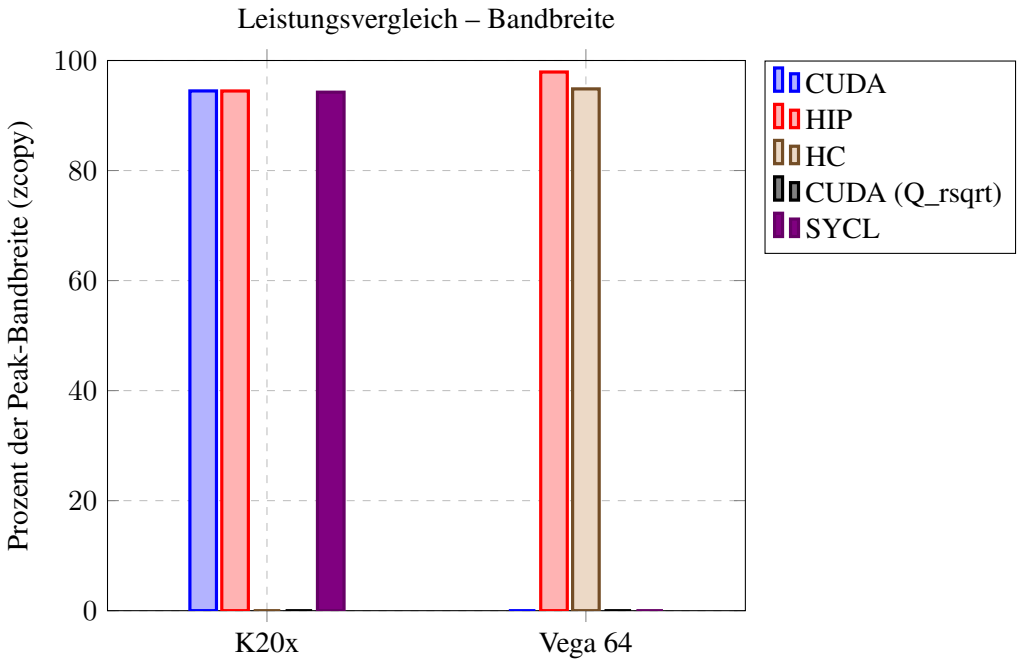


Abbildung 6.2: Prozentualer Leistungsvergleich (Bandbreite)

keit Gebrauch machen (können). Auch der direkte Zugriff auf den globalen Speicher von auf anderen Rechnern bzw. Knoten befindlicher GPUs, wie er im CUDA-Umfeld über *remote direct memory access* (RDMA) möglich ist, ist in den anderen Spracherweiterungen möglicherweise gar nicht vorhanden oder hinter abstrakten Konzepten versteckt.

Ein weiteres Thema ist die Untersuchung der Spracherweiterungen auf Hardware, die im klassischen HPC eher selten anzutreffen ist. Codeplays eigentliches Geschäft findet zu großen Teilen im Embedded-Bereich statt. Eine Untersuchung der SYCL-Implementierung auf Systemen mit sehr niedrigem Energiebedarf könnte daher interessante Erkenntnisse bieten. Ähnlich verhält es sich mit FPGAs, die experimentell über die von Xilinx vorangetriebene triSYCL-Implementierung angesprochen werden können. Hier wäre beispielsweise zu prüfen, wie portabel die erreichte Performanz zwischen verschiedenen Beschleunigerfamilien wie GPUs und FPGAs ist.

Aus einer nichttechnischen Perspektive ist der Ausgang des gegenwärtig vor dem Obersten Gerichtshof der Vereinigten Staaten verhandelten Prozesses zwischen den Firmen Oracle und Google von großer Relevanz. Oracle sieht eine Verletzung seiner Urheberrechte in der durch Google für die Android-Plattform erstellten Kopie des Java-API. Da HIP ebenfalls eine Kopie eines anderen API ist, wird das Urteil auch auf diesen Fall anwendbar sein. (vgl. [Grü19])

Literatur

[Aar03] AARSETH, Sverre J.: *Gravitational N-Body Simulations*. 1. Auflage. Cambridge University Press, 2003

[ARG17] ALIAGA, José I. ; REYES, Ruymán ; GOLI, Mehdi: SYCL-BLAS: Leveraging Expression Trees for Linear Algebra. In: *Proceedings of the 5th International Workshop on OpenCL*, 2017

[ASA+18] AFZAL, Ayesha ; SCHMITT, Christian ; ALHADDAD, Samer ; GRYNKO, Yevgen ; TEICH, Jürgen ; FÖRSTNER, Jens ; HANNIG, Frank: Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study. In: *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors*, 2018

[bes18] *CUDA C Best Practices Guide*. Oktober 2018. – `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, zuletzt abgerufen am 21. Februar 2019

[CK17] COPIK, Marcin ; KAISER, Hartmut: Using SYCL as Implementation Framework for HPX.Compute. In: *Proceedings of the 5th International Workshop on OpenCL*, 2017

[com19] *ComputeCpp CE – Overview*. 2019. – `https://developer.codeplay.com/computecppce/latest/overview`, zuletzt abgerufen am 10. Februar 2019

[CPB16] CARDOSO DA SILVA, Hércules ; PISANI, Flávia ; BORIN, Edson: A Comparative Study of SYCL, OpenCL, and OpenMP. In: *2016 International Symposium on Computer Architecture and High Performance Computing Workshops*, 2016, S. 61–66

[cpp12] *C++ AMP: Language and Programming Model – Version 1.0*. August 2012

[cud11] *CUDA Toolkit 4.0 Readiness For CUDA Applications*. März 2011

[cud14] *NVIDIA CUDA Toolkit V6.0 – Release Notes for Windows, Linux, and Mac OS*. Februar 2014

[cud18a] *CUDA C Programming Guide*. Oktober 2018. – `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, zuletzt abgerufen am 26. Januar 2019

[cud18b] *NVIDIA CUDA Toolkit 10.0.130 – Release Notes for Windows, Linux, and Mac OS*. Oktober 2018

[cud18c] *NVIDIA CUDA Toolkit 9.0.176 – Release Notes for Windows, Linux, and Mac OS*. März 2018

[Dam17] DAMKROGER, Trish: *Unleashing High-Performance Computing Today and Tomorrow*. November 2017. – `https://itpeernetwork.intel.com/unleashing-high-performance-computing`, zuletzt abgerufen am 18. Januar 2019

[DI93] DYER, Charles C. ; IP, Peter S.S̃.: Softening in N-Body simulations of collisionless systems. In: *Astrophysical Journal, Part 1* 409 (1993), Mai, Nr. 1, S. 60–67

[DKO17] DOUMOULAKIS, Anastasios ; KERYELL, Ronan ; O'BRIEN, Kenneth: SYCL C++ and OpenCL interoperability experimentation with triSYCL. In: *Proceedings of the 5th International Workshop on OpenCL*, 2017

[Far18] FARE, Callum: Enabling Profiling for SYCL Applications. In: *Proceedings of the 6th International Workshop on OpenCL*, 2018

[GIR17] GOLI, Mehdi ; IWANSKI, Luke ; RICHARDS, Andrew: Accelerated Machine Learning Using TensorFlow and SYCL on OpenCL Devices. In: *Proceedings of the 5th International Workshop on OpenCL*, 2017

[GLEC+17] GÓMEZ-LUNA, Juan ; EL HAJJ, Izzat ; CHANG, Li-Wen ; GARCÍA-FLORES, Víctor ; GARCIA DE GONZALO, Simon ; JABLIN, Thomas B. ; PEÑA, Antonio J. ; HWU, Wen-mei: Chai: Collaborative Heterogeneous Applications for Integrated-architectures. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software*, 2017, S. 43–54

[Grü19] GRÜNER, Sebastian: *Oracle gegen Google: Supreme Court soll Streit um Java-APIs prüfen*. Januar 2019. – https://www.golem.de/news/oracle-gegen-google-supreme-court-soll-streit-um-java-apis-pruefen-1901-138978.html, zuletzt abgerufen am 19. Februar 2019

[Har07] HARRIS, Mark: *Optimizing parallel reduction in CUDA*. 2007. – https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, zuletzt abgerufen am 27. Januar 2019

[har13] *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. April 2013. – https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, zuletzt abgerufen am 27. Januar 2019

[Har16] HARRIS, Mark: *CUDA 8 Features Revealed*. April 2016. – https://devblogs.nvidia.com/cuda-8-features-revealed/, zuletzt abgerufen am 26. Januar 2019

[hcr17] *HC API : Moving Beyond C++AMP for Accelerated GPU Computing*. Dezember 2017. – https://scchan.github.io/hcc/index.html, zuletzt abgerufen am 03. Februar 2019

[HWF17] HOU, Kaixi ; WANG, Hao ; FENG, Wu-chun: GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In: *Proceedings of the Computing Frontiers Conference 2017*, 2017, S. 107–116

[int19] *Intel Project for LLVM technology*. Januar 2019. – https://github.com/intel/llvm/tree/sycl, zuletzt abgerufen am 10. Februar 2019

[Jes17] JESENŠEK, Jure: *Prenos orodja SYCL-GTX na operacijski sistem Linux in koprocesor Xeon Phi*. Kongresni trg 12, 1000 Laibach, Slowenien, Univerza v Ljubljani, Diplomarbeit, 2017

[Kon17] KONSTANTINIDIS, Elias N.: *A GPU performance estimation model based on micro-benchmarks and black-box kernel profiling*. Πανεπιστημίου 30, 106 79 Αθήνα, Griechenland, Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών, Diss., Juli 2017

[KY18] KERYELL, Ronan ; YU, Lin-Ya: Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation. In: *Proceedings of the 6th International Workshop on OpenCL*, 2018

[Li16] LI, Hongyu: 在*HSA*系統上以間接*Finalizer*進行程式碼最佳化. No. 1, Sec. 4, Roosevelt Rd., Taipei 10617, Taiwan, National Taiwan University, Diplomarbeit, Januar 2016. http://dx.doi.org/10.6342/NTU201601752. – DOI 10.6342/NTU201601752

[LPD16] LOMÜLLER, Victor ; POTTER, Ralph ; DOLINSKY, Uwe: *C++ on Accelerators: Supporting Single-Source SYCL and HSA Programming Models Using Clang*. Vortrag, März 2016. – https://llvm.org/devmtg/2016-03/Presentations/Offload-EuroLLVM2016.pdf, zuletzt abgerufen am 09. Februar 2019

[LT16] LARSSON, Marcus ; TSOG, Nandinbaatar: *Time Predictability of GPU Kernel on an HSA Compliant Platform*. 721 23 Västerås, Schweden, Mälardalens högskola, Masterarbeit, Juni 2016

[Lui14] LUITJENS, Justin: *Faster Parallel Reductions on Kepler*. Februar 2014. – https://devblogs.nvidia.com/faster-parallel-reductions-kepler/, zuletzt abgerufen am 27. Januar 2019

[NHP07] NYLAND, Lars ; HARRIS, Mark ; PRINS, Jan: Fast N-Body Simulation with CUDA. In: NGUYEN, Hubert (Hrsg.): *GPU Gems 3*. Erste Auflage. Addison-Wesley, August 2007, Kapitel 31, S. 677–696

[nju17] NJUFFA: *theoretical/real shared/dram peak memory throughput*. Januar 2017. – https://devtalk.nvidia.com/default/topic/985255/cuda-programming-and-performance/theoretical-real-shared-dram-peak-memory-throughput/post/5048359/#5048359, zuletzt abgerufen am 26. Januar 2019

[NRB+18] NOBRE, Ricardo ; REIS, Luís ; BISPO, João ; CARVALHO, Tiago ; CARDOSO, João M.P̃. ; CHERUBIN, Stefano ; AGOSTA, Giovanni: Aspect-Driven Mixed-Precision Tuning Targeting GPUs. In: *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2018, S. 26–31

[pas18] *Tuning CUDA Applications for Pascal*. Oktober 2018. – https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html, zuletzt abgerufen am 20. Februar 2019

[Ram18] RAMARAO, Pramod: *CUDA 10 Features Revealed: Turing, CUDA Graphs, and More*. September 2018. – https://devblogs.nvidia.com/cuda-10-features-revealed/, zuletzt abgerufen am 03. Februar 2019

[SGZ+16] SUN, Yifan ; GONG, Xiang ; ZIABARI, Amir K. ; YU, Leiming ; LI, Xiangyu ; MUKHERJEE, Saoni ; MCCARDWELL, Carter ; VILLEGAS, Alejandro ; KAELI, David: Hetero-Mark, A Benchmark Suite for CPU-GPU Collaborative Computing. In: *2016 IEEE International Symposium on Workload Characterization*, 2016, S. 1–10

[SMB+18] SUN, Yifan ; MUKHERJEE, Saoni ; BARUAH, Trinayan ; DONG, Shi ; GUTIERREZ, Julian ; MOHAN, Prannoy ; KAELI, David: Evaluating Performance Tradeoffs on the Radeon Open Compute Platform. In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software*, 2018, S. 209–218

[SP18] ST CLERE SMITHE, Toby ; POTTER, Ralph: Building a brain with SYCL and modern C++. In: *Proceedings of the 6th International Workshop on OpenCL*, 2018

[Sun16] SUN, Peng: *High-Level Programming Model for Heterogeneous Embedded Systems Using Multicore Industry Standard APIs*. 4800 Calhoun Rd, Houston, TX 77004, USA, University of Houston, Diss., Juni 2016

[syc18] *SYCL™ Specification – SYCL integrates OpenCL devices with modern C++*. Juli 2018

[TOP18] TOP500.ORG: *November 2018 | TOP500 Supercomputer Sites*. November 2018. – `https://www.top500.org/lists/2018/11/`, zuletzt abgerufen am 18. Januar 2019

[Tri14] TRIGKAS, Angelos: *Investigation of the OpenCL SYCL Programming Model*. Old College, South Bridge, Edinburgh EH8 9YL, Vereinigtes Königreich, The University of Edinburgh, Masterarbeit, August 2014

[tri18] *triSYCL*. Dezember 2018. – `https://github.com/triSYCL/triSYCL`, zuletzt abgerufen am 10. Februar 2019

[Ver67] VERLET, Loup: Computer Ëxperimentsön Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. In: *Physical Review* 159 (1967), Juli, Nr. 1, S. 98–103

[Won18] WONG, Michael: *The Future Direction of C++ and the Four Horsemen of Heterogeneous Computing*. Vortrag, November 2018. – `https://www.youtube.com/watch?v=7Y3-pV_b-1U`, zuletzt abgerufen am 25. Januar 2019

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

A CUDA-Kernel

A.1 zcopy

```
__global__ void read_write(const float4* __restrict__ A,
                           float4* __restrict__ B,
                           std::size_t num_elems)
{
    auto stride = gridDim.x * blockDim.x;
    for(auto i = blockIdx.x * blockDim.x + threadIdx.x;
             i < elems;
             i += stride)
    {
        B[i] = A[i];
    }
}

__global__ void write(float4* __restrict__ B, std::size_t num_elems)
{
    auto stride = gridDim.x * blockDim.x;
    for(auto i = blockIdx.x * blockDim.x + threadIdx.x;
             i < elems;
             i += stride)
    {
        B[i] = make_float4{0.f, 0.f, 0.f, 0.f};
    }
}
```

Quelltext A.1: zcopy – CUDA-Implementierung

Anmerkung: Sofern der Programmierer garantieren kann, das zwei Zeiger auf unterschiedliche, nicht überlappende Datenbereiche verweisen, können diese mit dem Schlüsselwort `__restrict__` markiert werden. Der Compiler kann dadurch effizientere Speicherzugriffe generieren.

A.2 Reduction

```
__global__ void block_reduce(const int* __restrict__ data,
                             int* __restrict__ result,
                             std::size_t size)
{
    extern __shared__ int scratch[];

    auto i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i >= size) return;

    auto tsum = data[i];

    auto grid_size = gridDim.x * blockDim.x;
    i += grid_size;

    while((i + 3 * grid_size) < size) {
        tsum += data[i] + data[i + grid_size] +
                data[i + 2 * grid_size] + data[i + 3 * grid_size];
            i += 4 * grid_size;
    }
    while(i < size) { // noch fehlende Elemente
        tsum += data[i];
        i += grid_size;
    }

    scratch[threadIdx.x] = tsum;
    __syncthreads();

    #pragma unroll
    for(auto bs = blockDim.x, bsup = (blockDim.x + 1) / 2;
             bs > 1; bs /= 2, bsup = (bs + 1) / 2) {
        auto cond = threadIdx.x < bsup &&
            (threadIdx.x + bsup) < blockDim.x &&
            (blockIdx.x * blockDim.x + threadIdx.x + bsup)
            < size;

        if(cond)
            scratch[threadIdx.x] += scratch[threadIdx.x + bsup];
        __syncthreads();
    }

    if(threadIdx.x == 0) result[blockIdx.x] = scratch[0];
}
```

Quelltext A.2: Reduce – CUDA-Implementierung

A.3 N-Body

```
constexpr auto eps = 0.001f;
constexpr auto eps2 = eps * eps;

__device__ auto body_body_interaction(float4 bi, float4 bj, float3 ai)
-> float3
{
    // r_ij [3 FLOPS]
    auto r = float3{};
    r.x = bj.x - bi.x;
    r.y = bj.y - bi.y;
    r.z = bj.z - bi.z;

    // dist_sqr = skalarprodukt(r_ij, r_ij) + epsilon^2 [6 FLOPS]
    auto dist_sqr = fmaf(r.x, r.x,
                         fmaf(r.y, r.y,
                              fmaf(r.z, r.z, eps2)));

    // inv_dist_cube = 1 / dist_sqr^(3/2) [4 FLOPS]
    auto dist_sixth = dist_sqr * dist_sqr * dist_sqr;
    auto inv_dist_cube = rsqrtf(dist_sixth);

    // s = m_j * inv_dist_cube [1 FLOP]
    auto s = bj.w * inv_dist_cube;

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai.x = fmaf(r.x, s, ai.x);
    ai.y = fmaf(r.y, s, ai.y);
    ai.z = fmaf(r.z, s, ai.z);

    return ai;
}
```

Quelltext A.3: N-Body: body_body_interaction - CUDA-Implementierung

```
__device__ auto force_calculation(float4 body_pos,
                                  const float4* positions,
                                  unsigned tiles)
-> float3
{
    extern __shared__ float4 sh_position[];

    auto acc = float3{};

    for(auto tile = 0u; tile < tiles; ++tile)
    {
        auto idx = tile * blockDim.x + threadIdx.x;

        sh_position[threadIdx.x] = positions[idx];
        __syncthreads();

        // entspricht tile_calculation()
        #pragma unroll
        for(auto i = 0u; i < blockDim.x; ++i)
            acc = body_body_interaction(body_pos, sh_position[i],
                                        acc);
        __syncthreads();
    }
    return acc;
}
```

Quelltext A.4: N-Body: force_calculation - CUDA-Implementierung

B HIP-Kernel

B.1 zcopy

```
__global__ void read_write(const float4* __restrict__ A,
                           float4* __restrict__ B,
                           std::size_t num_elems)
{
    auto stride = hipGridDim_x * hipBlockDim_x;
    for(auto i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
            i < elems;
            i += stride)
    {
        B[i] = A[i];
    }
}

__global__ void write(float4* __restrict__ B, std::size_t num_elems);
{
    auto stride = hipGridDim_x * hipBlockDim_x;
    for(auto i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
            i < elems;
            i += stride)
    {
        B[i] = make_float4{0.f, 0.f, 0.f, 0.f};
    }
}
```

Quelltext B.1: zcopy – HIP-Implementierung

B.2 Reduction

```
__global__ void block_reduce(const int* __restrict__ data,
                             int* __restrict__ result,
                             std::size_t size)
{
    extern __shared__ int scratch[];

    auto i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if(i >= size) return;

    auto tsum = data[i];

    auto grid_size = hipGridDim_x * hipBlockDim_x;
    i += grid_size;

    while((i + 3 * grid_size) < size) {
        tsum += data[i] + data[i + grid_size] +
                data[i + 2 * grid_size] + data[i + 3 * grid_size];
            i += 4 * grid_size;
    }
    while(i < size) { // noch fehlende Elemente
        tsum += data[i];
        i += grid_size;
    }

    scratch[hipThreadIdx_x] = tsum;
    __syncthreads();

    #pragma unroll
    for(auto bs = hipBlockDim_x, bsup = (hipBlockDim_x + 1) / 2;
             bs > 1; bs /= 2, bsup = (bs + 1) / 2) {
        auto cond = hipThreadIdx_x < bsup &&
            (hipThreadIdx_x + bsup) < hipBlockDim_x &&
            (hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x + bsup)
            < size;

        if(cond)
            scratch[hipThreadIdx_x] += scratch[hipThreadIdx_x + bsup];
        __syncthreads();
    }

    if(hipThreadIdx_x == 0) result[hipBlockIdx_x] = scratch[0];
}
```

Quelltext B.2: Reduce – HIP-Implementierung

B.3 N-Body

```
constexpr auto eps = 0.001f;
constexpr auto eps2 = eps * eps;

__device__ auto body_body_interaction(float4 bi, float4 bj, float3 ai)
-> float3
{
    // r_ij [3 FLOPS]
    auto r = float3{};
    r.x = bj.x - bi.x;
    r.y = bj.y - bi.y;
    r.z = bj.z - bi.z;

    // dist_sqr = skalarprodukt(r_ij, r_ij) + epsilon^2 [6 FLOPS]
    auto dist_sqr = fmaf(r.x, r.x,
                         fmaf(r.y, r.y,
                              fmaf(r.z, r.z, eps2)));

    // inv_dist_cube = 1 / dist_sqr^(3/2) [4 FLOPS]
    auto dist_sixth = dist_sqr * dist_sqr * dist_sqr;
    auto inv_dist_cube = rsqrtf(dist_sixth);

    // s = m_j * inv_dist_cube [1 FLOP]
    auto s = bj.w * inv_dist_cube;

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai.x = fmaf(r.x, s, ai.x);
    ai.y = fmaf(r.y, s, ai.y);
    ai.z = fmaf(r.z, s, ai.z);

    return ai;
}
```

Quelltext B.3: N-Body: body_body_interaction - HIP-Implementierung

```
__device__ auto force_calculation(float4 body_pos,
                                  const float4* positions,
                                  unsigned tiles)
-> float3
{
    extern __shared__ float4 sh_position[];

    auto acc = float3{};

    for(auto tile = 0u; tile < tiles; ++tile)
    {
        auto idx = tile * hipBlockDim_x + hipThreadIdx_x;

        sh_position[hipThreadIdx_x] = positions[idx];
        __syncthreads();

        // entspricht tile_calculation()
        #pragma unroll
        for(auto i = 0u; i < hipBlockDim_x; ++i)
            acc = body_body_interaction(body_pos, sh_position[i],
                                        acc);
        __syncthreads();
    }
    return acc;
}
```

Quelltext B.4: N-Body: force_calculation - HIP-Implementierung

C HC-Kernel

C.1 zcopy

```
[[hc]]
template <typename DataT>
void read_write(hc::tiled_index<1> idx, hc::array_view<DataT, 1> A,
                hc::array_view<DataT, 1> B, std::size_t elems,
                unsigned int tiles, unsigned int tile_size)
{
    auto stride = tiles * tile_size;
    for(auto i = idx.tile[0] * tile_size + idx.local[0];
             i < elems;
             i += stride)
    {
        B[i] = A[i];
    }
}

[[hc]]
template <typename DataT>
void write(hc::tiled_index<1> idx, hc::array_view<DataT, 1> B,
           std::size_t elems, unsigned int tiles,
           unsigned int tile_size)
{
    auto stride = tiles * tile_size;
    for(auto i = idx.tile[0] * tile_size + idx.local[0];
             i < elems;
             i += stride)
    {
        B[i] = DataT{};
    }
}
```

Quelltext C.1: zcopy - HC-Implementierung

C.2 Reduction

```
void block_reduce(hc::tiled_index<1> idx,
    hc::array_view<int, 1> data, hc::array_view<int, 1> result,
    std::size_t size, int blocks, int block_size) [[hc]]
{
    auto scratch = static_cast<int*>(
        hc::get_dynamic_group_segment_base_pointer());

    auto i = static_cast<unsigned>(idx.tile[0] * block_size
                                   + idx.local[0]);
    if(i >= size) return;

    auto tsum = data[i];

    auto grid_size = blocks * block_size;
    i += grid_size;

    while((i + 3 * grid_size) < size) {
        tsum += data[i] + data[i + grid_size] +
                data[i + 2 * grid_size] + data[i + 3 * grid_size];
        i += 4 * grid_size;
    }
    while(i < size) { // Noch fehlende Elemente
        tsum += data[i];
        i += grid_size;
    }

    scratch[idx.local[0]] = tsum;
    idx.barrier.wait_with_tile_static_memory_fence();

    #pragma unroll
    for(auto bs = block_size, bsup = (block_size + 1) / 2;
             bs > 1; bs /= 2, bsup = (bs + 1) / 2) {
        auto cond = idx.local[0] < bsup &&
            (idx.local[0] + bsup) < block_size &&
            (idx.tile[0] * block_size + idx.local[0] + bsup) < size;

        if(cond)
            scratch[idx.local[0]] += scratch[idx.local[0] + bsup];
        idx.barrier.wait_with_tile_static_memory_fence();
    }

    if(idx.local[0] == 0) result[idx.tile[0]] = scratch[0];
}
```

Quelltext C.2: Reduction - HC-Implementierung

C.3 N-Body

```
using float3 = hc::short_vector::float_3;
using float4 = hc::short_vector::float_4;

[[hc]]
auto body_body_interaction(float4 bi, float4 bj, float3 ai) -> float3
{
    constexpr auto eps = 0.001f;
    constexpr auto eps2 = eps * eps;

    // r_ij [3 FLOPS]
    auto r = bj.get_xyz() - bi.get_xyz();

    // dist_sqr = skalarprodukt(r, r) + epsilon^2 [6 FLOPS]
    auto dist_sqr = fmaf(r.x, r.x,
                         fmaf(r.y, r.y,
                              fmaf(r.z, r.z, eps2)));

    // inv_dist_cube = 1 / dist_sqr^(3/2) [4 FLOPS]
    auto dist_sixth = dist_sqr * dist_sqr * dist_sqr;
    auto inv_dist_cube = rsqrtf(dist_sixth);

    // s = m_j * inv_dist_cube [1 FLOP]

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai.x = fmaf(r.x, s, ai.x);
    ai.y = fmaf(r.y, s, ai.y);
    ai.z = fmaf(r.z, s, ai.z);

    return ai;
}
```

Quelltext C.3: N-Body: body_body_interaction - HC-Implementierung

```
using float3 = hc::short_vector::float_3;
using float4 = hc::short_vector::float_4;

[[hc]]
auto force_calculation(hc::tiled_index<1> idx, float4 body_pos,
                       hc::array_view<const float4, 1> positions,
                       float4* sh_position, unsigned tiles)
-> float3
{
    auto acc = float3{};

    for(auto tile = 0u; tile < tiles; ++tile)
    {
        auto id = tile * idx.tile_dim[0] + idx.local[0];

        sh_position[idx.local[0]] = positions[id];
        idx.barrier.wait_with_tile_static_memory_fence();

        // entspricht tile_calculation()
        #pragma unroll
        for(auto i = 0u; i < idx.tile_dim[0]; ++i)
            acc = body_body_interaction(body_pos, sh_position[i],
                                        acc);
        idx.barrier.wait_with_tile_static_memory_fence();
    }
    return acc;
}
```

Quelltext C.4: N-Body: force_calculation - HC-Implementierung

D SYCL-Kernel

D.1 zcopy

```
struct reader_writer
{
    cl::sycl::accessor<cl::sycl::float4, 1,
                       cl::sycl::access::mode::read,
                       cl::sycl::access::target::global_buffer> A;
    cl::sycl::accessor<cl::sycl::float4, 1,
                       cl::sycl::access::mode::discard_write,
                       cl::sycl::access::target::global_buffer> B;

    auto operator()(cl::sycl::nd_item<1> my_item) -> void
    {
        auto stride = my_item.get_group_range(0) *
                      my_item.get_local_range(0);
        for(auto i = my_item.get_global_id(0); i < elems; i += stride)
        {
            B[i] = A[i];
        }
    }
};

struct writer
{
    cl::sycl::accessor<cl::sycl::float4, 1,
                       cl::sycl::access::mode::discard_write,
                       cl::sycl::access::target::global_buffer> B;

    auto operator()(cl::sycl::nd_item<1> my_item) -> void
    {
        auto stride = my_item.get_group_range(0) *
                      my_item.get_local_range(0);
        for(auto i = my_item.get_global_id(0); i < elems; i += stride)
        {
            B[i] = cl::sycl::float4{};
        }
    }
};
```

Quelltext D.1: zcopy – SYCL-Implementierung

D.2 Reduction

```
namespace sycl = cl::sycl;
struct block_reduce {
    sycl::accessor<int, 1, sycl::access::mode::read,
                   sycl::access::target::global_buffer> data;
    sycl::accessor<int, 1, sycl::access::mode::write,
                   sycl::access::target::global_buffer> result;
    sycl::accessor<int, 1, sycl::access::mode::read_write,
                   sycl::access::target::local> scratch;
    std::size_t size;
    auto operator()(sycl::nd_item<1> my_item) -> void {
        auto i = my_item.get_global_id(0);
        if(i >= size) return;
        auto tsum = data[i];
        auto grid_size = my_item.get_group_range(0) *
                         my_item.get_local_range(0);
        i += grid_size;
        while((i + 3 * grid_size) < size) {
            tsum += data[i] + data[i + grid_size] +
                data[i + 2 * grid_size] + data[i + 3 * grid_size];
            i += 4 * grid_size;
        }
        while(i < size) { // noch verbleibende Elemente
            tsum += data[i]; i += grid_size;
        }
        scratch[my_item.get_local_id(0)] = tsum;
        my_item.barrier(sycl::access::fence_space::local_space);
        auto block_size = my_item.get_local_range(0);
        auto global_id = my_item.get_global_id(0);
        auto local_id = my_item.get_local_id(0);

        #pragma unroll
        for(auto bs = block_size, bsup = (block_size + 1) / 2;
                 bs > 1; bs /= 2, bsup = (bs + 1) / 2) {
            auto cond = local_id < bsup &&
                        (local_id + bsup) < block_size &&
                        (global_id + bsup) < size;
            if(cond) scratch[local_id] += scratch[local_id + bsup];
            my_item.barrier(sycl::access::fence_space::local_space);
        }
        if(my_item.get_local_id(0) == 0)
            result[my_item.get_group_linear_id()] = scratch[0];
    }
};
```

Quelltext D.2: N-Body: Reduction - SYCL-Implementierung

D.3 N-Body

```
using float3 = cl::sycl::float3;
using float4 = cl::sycl::float4;

auto body_body_interaction(float4 bi, float4 bj, float3 ai) -> float3
{
    // r_ij [3 FLOPS]
    auto r = bj.xyz() - bi.xyz();

    // dist_sqr = dot(r_ij, r_ij) + EPS^2 [6 FLOPS]
    const auto dist_sqr = cl::sycl::fma(r.x(), r.x(),
                          cl::sycl::fma(r.y(), r.y(),
                              cl::sycl::fma(r.z(), r.z(), eps2)));

    // inv_dist_cube = 1/dist_sqr^(3/2) [4 FLOPS]
    auto dist_sixth = dist_sqr * dist_sqr * dist_sqr;
    auto inv_dist_cube = Q_rsqrt(dist_sixth);

    // s = m_j * inv_dist_cube [1 FLOP]
    const auto s = float{bj.w()} * inv_dist_cube;
    const auto s3 = float3{s, s, s};

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai = cl::sycl::fma(r, s3, ai);

    return ai;
}
```

Quelltext D.3: N-Body: body_body_interaction - SYCL-Implementierung

```
namespace sycl = cl::sycl;
using float3 = sycl::float3;
using float4 = sycl::float4;

auto force_calculation(sycl::nd_item<1> my_item, float4 body_pos,
                       sycl::accessor<float4, 1,
                         sycl::access::mode::read,
                         sycl::access::target::global_buffer>
                         positions,
                       sycl::accessor<float4, 1,
                         sycl::access::mode::read_write,
                         sycl::access::target::local> sh_position,
                       unsigned tiles) -> float3
{
    auto acc = float3{0.f, 0.f, 0.f};

    for(auto tile = 0u; tile < tiles; ++tile)
    {
        auto idx = tile * my_item.get_local_range(0) +
                          my_item.get_local_id(0);

        sh_position[my_item.get_local_id()] = positions[idx];
        my_item.barrier(sycl::access::fence_space::local_space);

        // entspricht tile_calculation()
        #pragma unroll
        for(auto i = 0u; i < my_item.get_local_range(0); ++i)
        {
            acc = body_body_interaction(body_pos, sh_position[i],
                                        acc);
        }
        my_item.barrier(sycl::access::fence_space::local_space);
    }

    return acc;
}
```

Quelltext D.4: N-Body: force_calculation - SYCL-Implementierung

E Benchmark-Ergebnisse

E.1 HIP-zcopy (AMD)

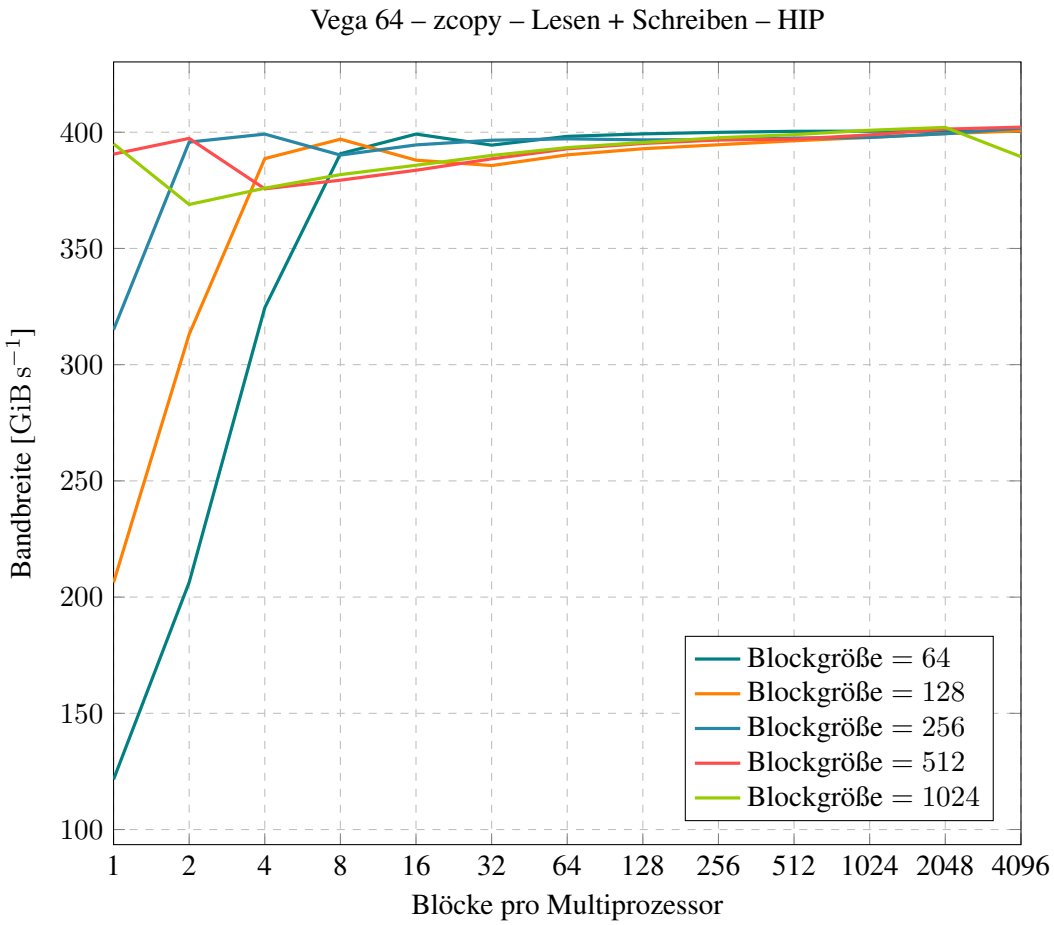


Abbildung E.1: zcopy: Bandbreite der Vega 64 ($n = 268435456$, Lesen und Schreiben, HIP)

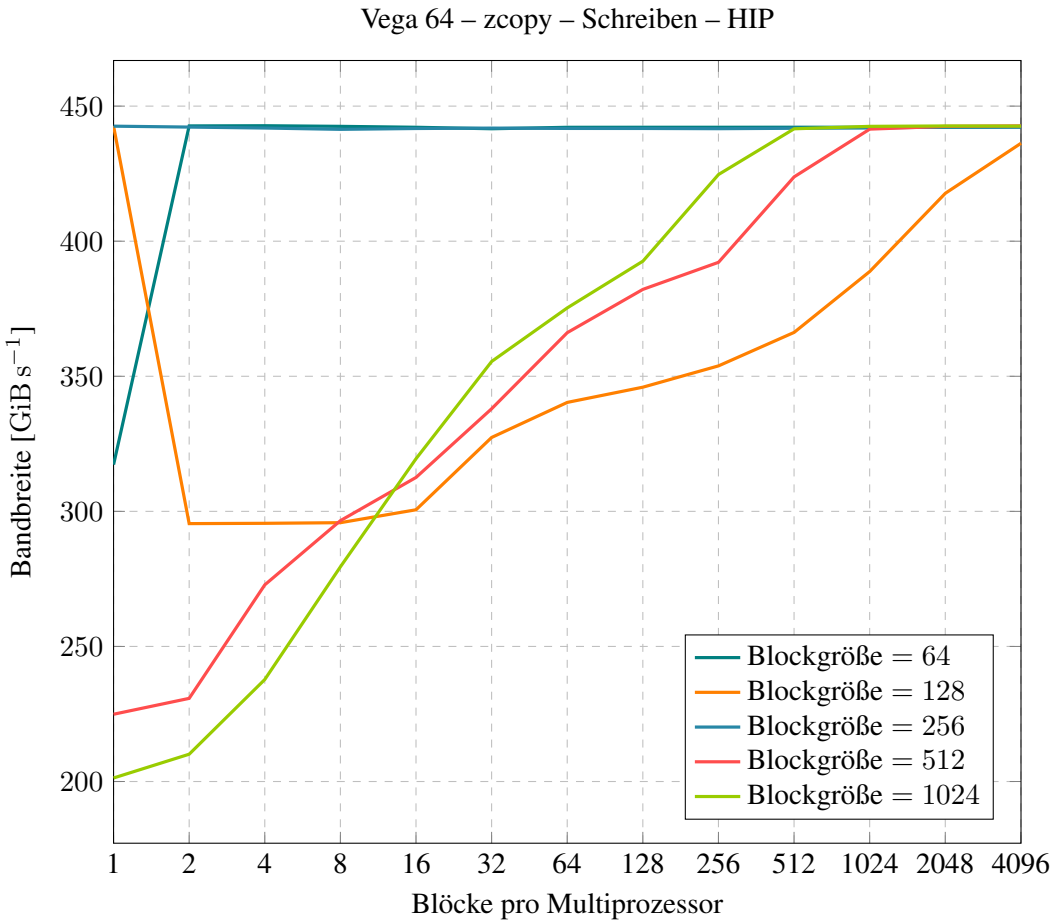


Abbildung E.2: zcopy: Bandbreite der Vega 64 ($n = 268435456$, Schreiben, HIP)

E.2 HIP-zcopy (NVIDIA)

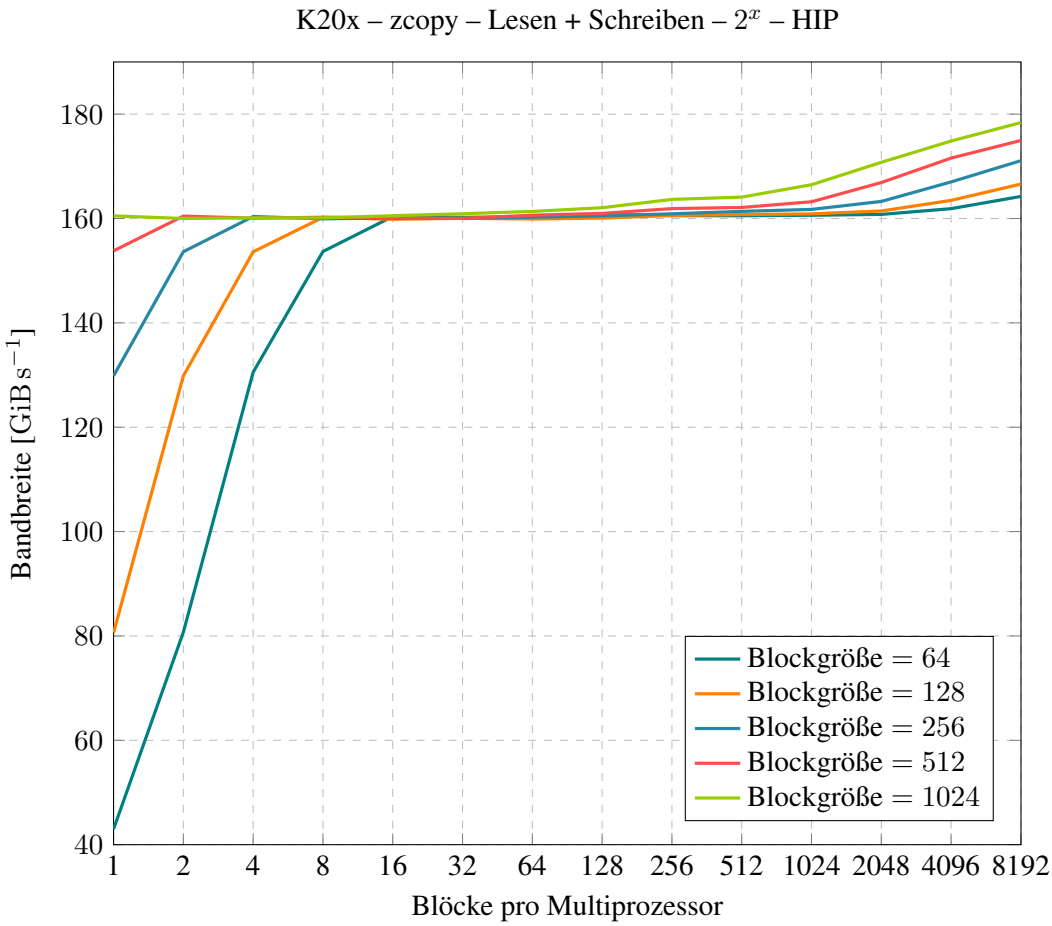


Abbildung E.3: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Lesen und Schreiben, HIP)

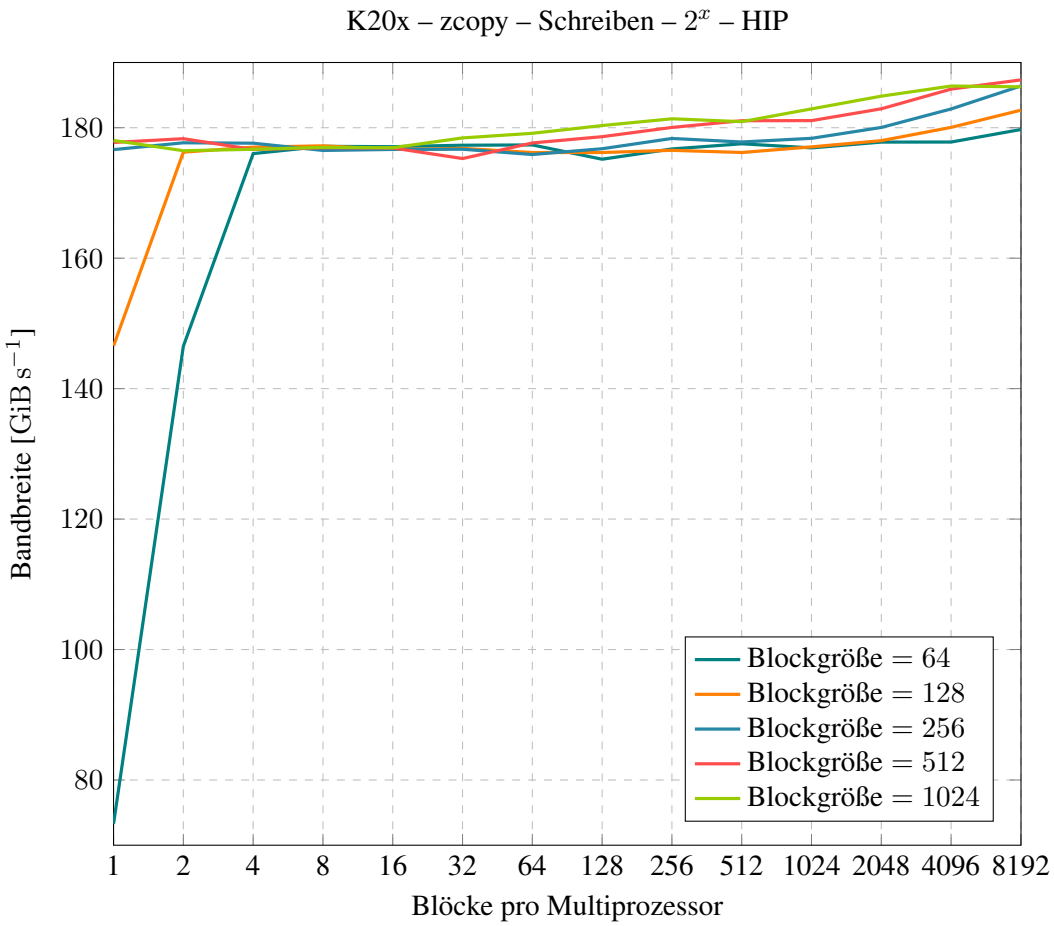


Abbildung E.4: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Schreiben, HIP)

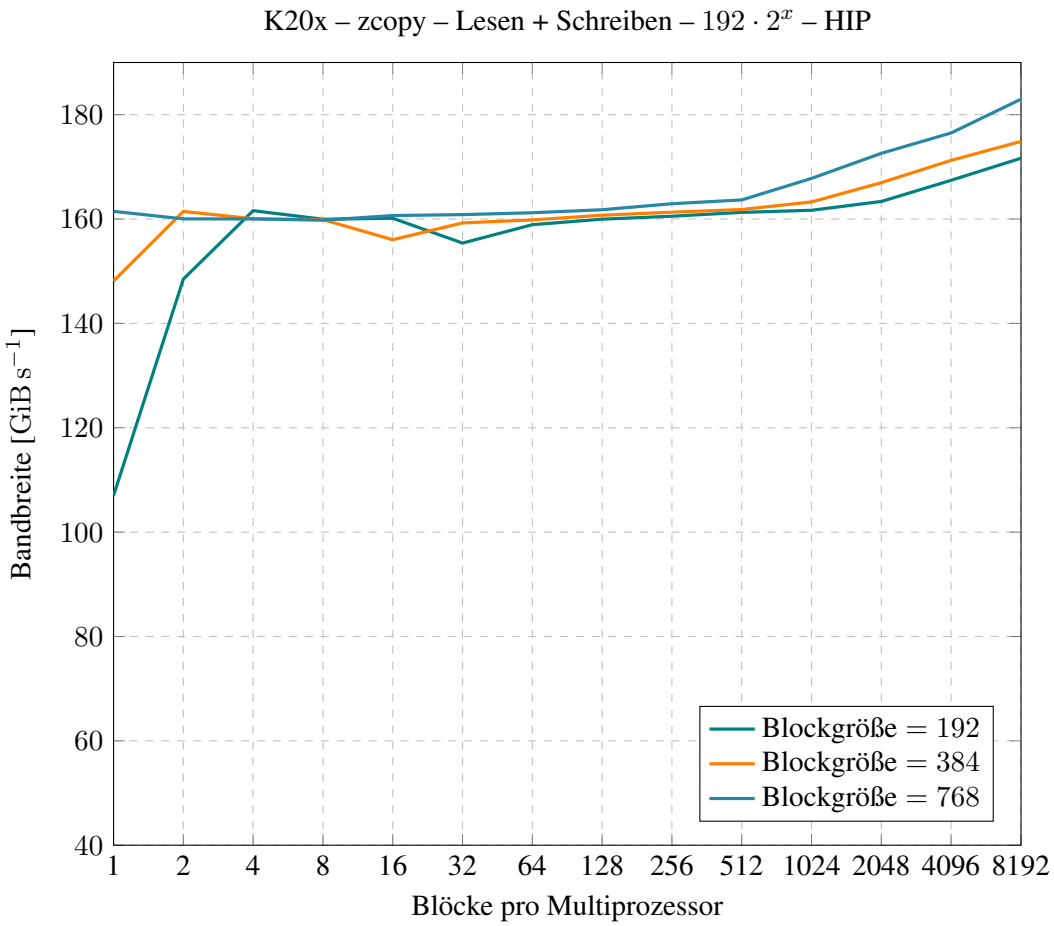


Abbildung E.5: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Lesen und Schreiben, HIP)

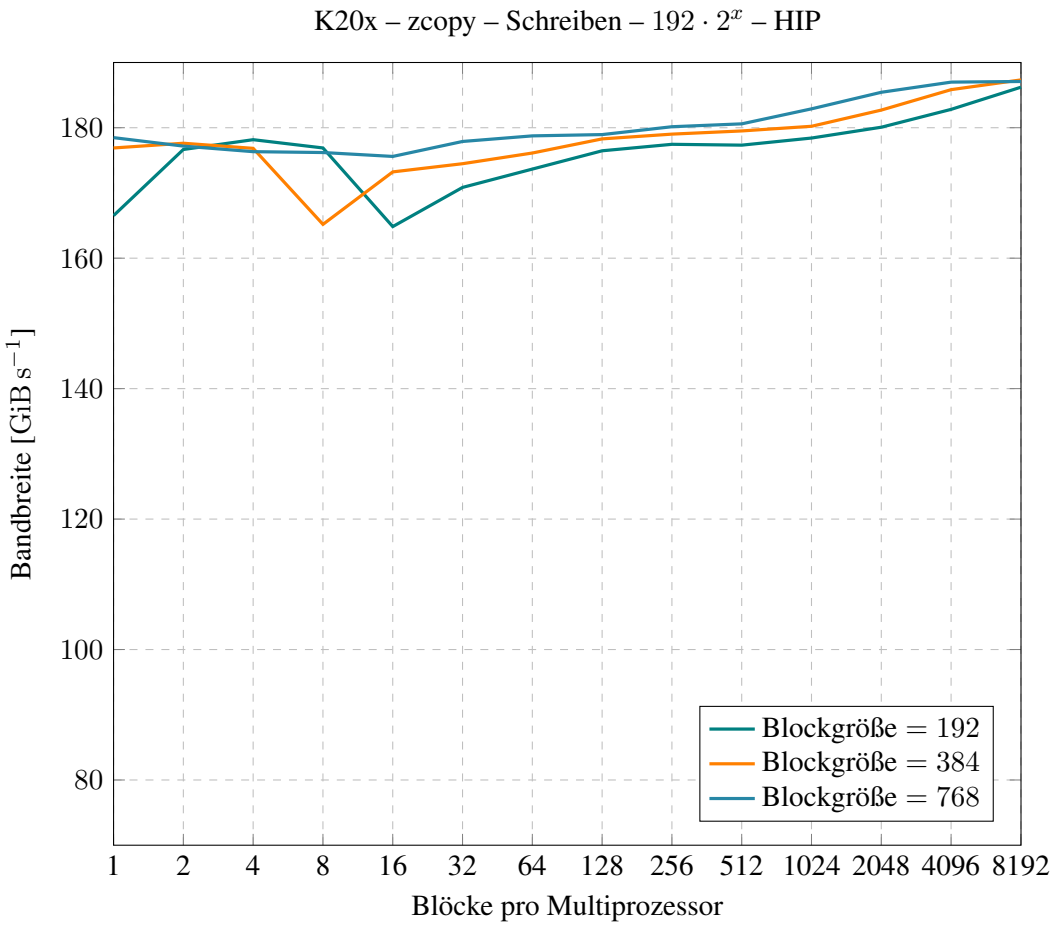


Abbildung E.6: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Schreiben, HIP)

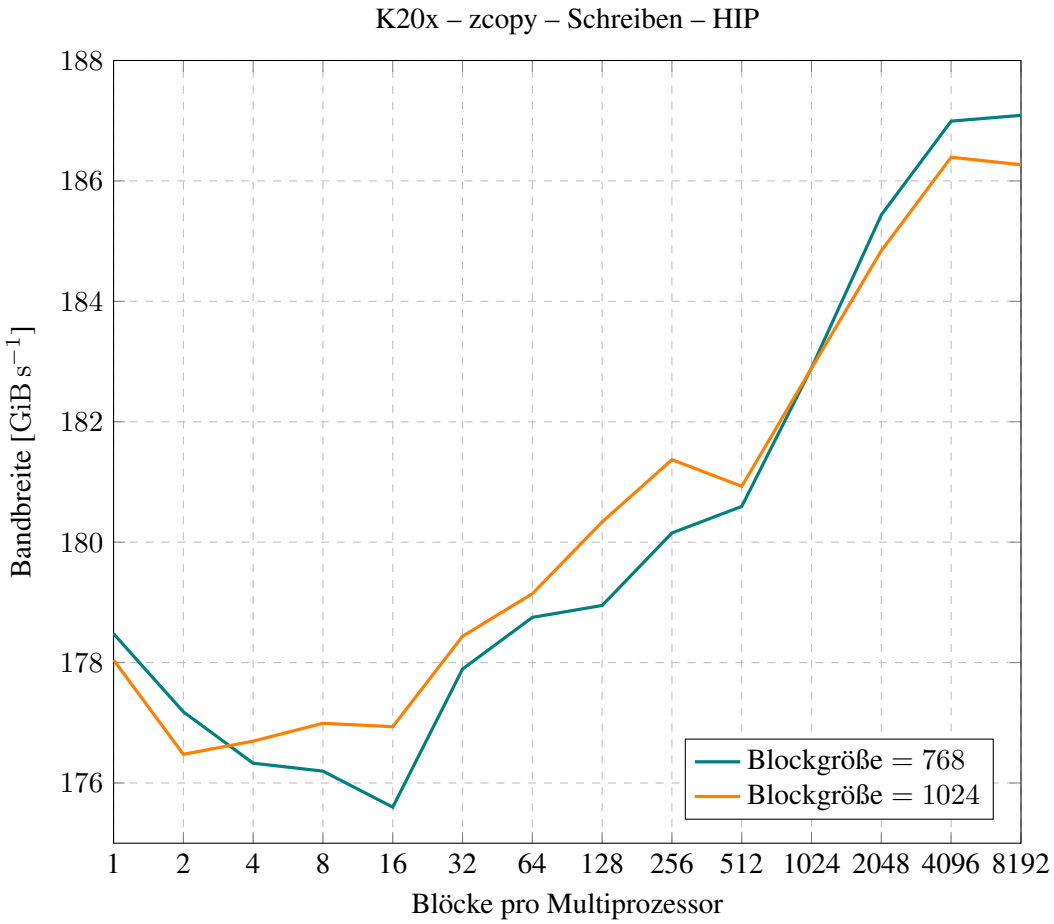


Abbildung E.7: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$) und 192er-Vielfache ($n = 88080384$, Schreiben, HIP)

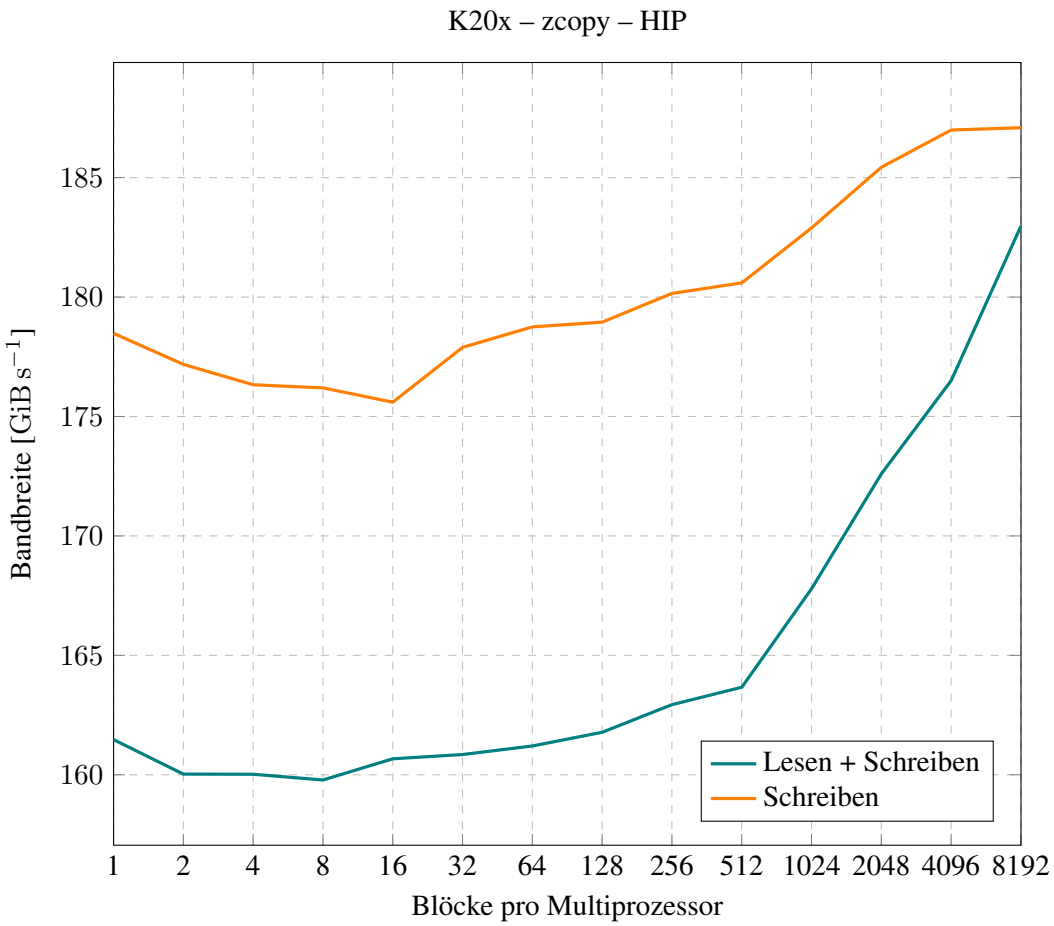


Abbildung E.8: zcopy: K20x-Bandbreite für 768er-Blöcke ($n = 88080384$, HIP)

E.3 SYCL-zcopy

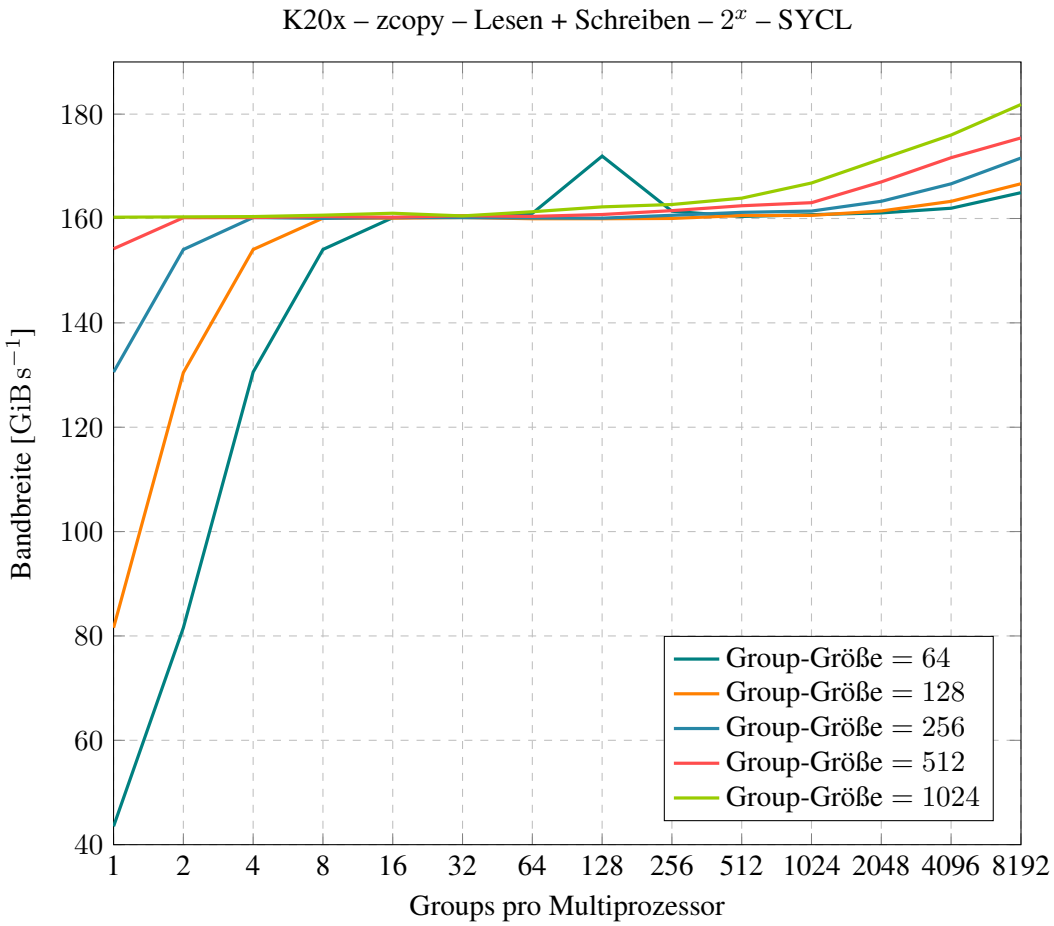


Abbildung E.9: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Lesen und Schreiben, SYCL)

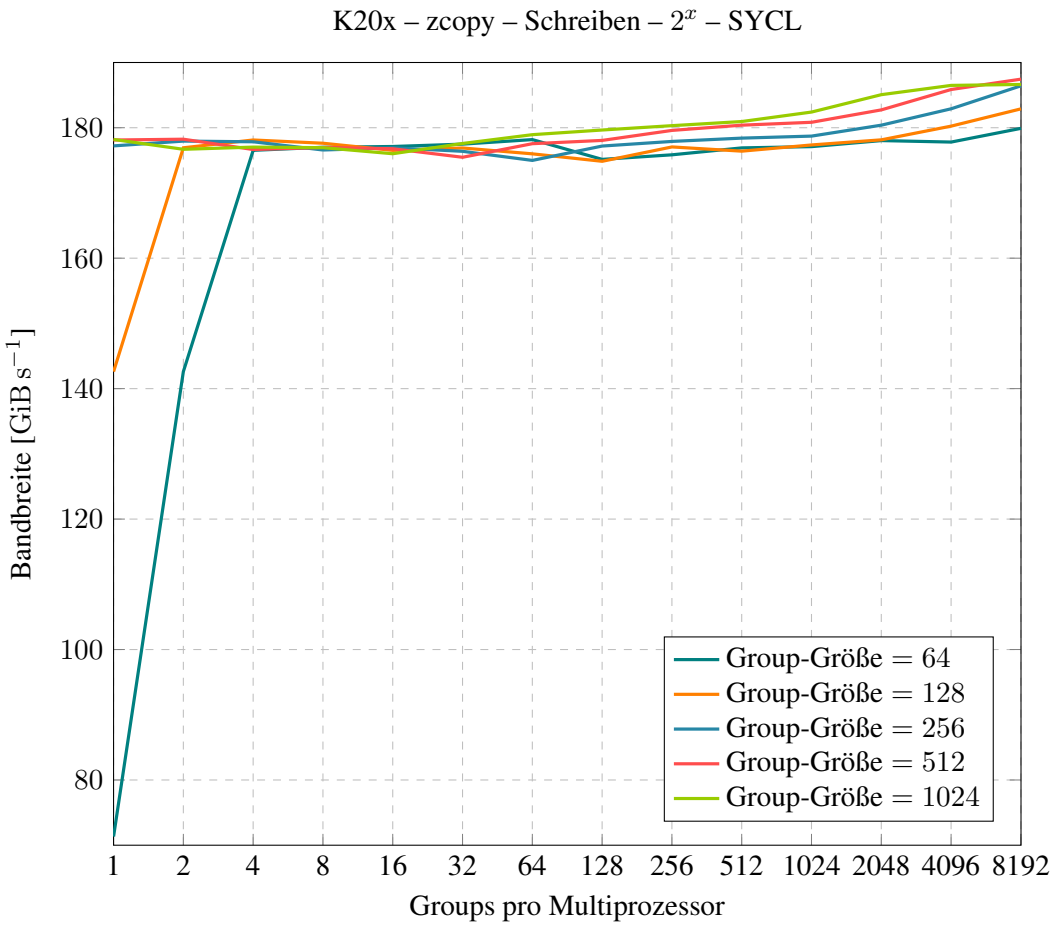


Abbildung E.10: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$, Schreiben, SYCL)

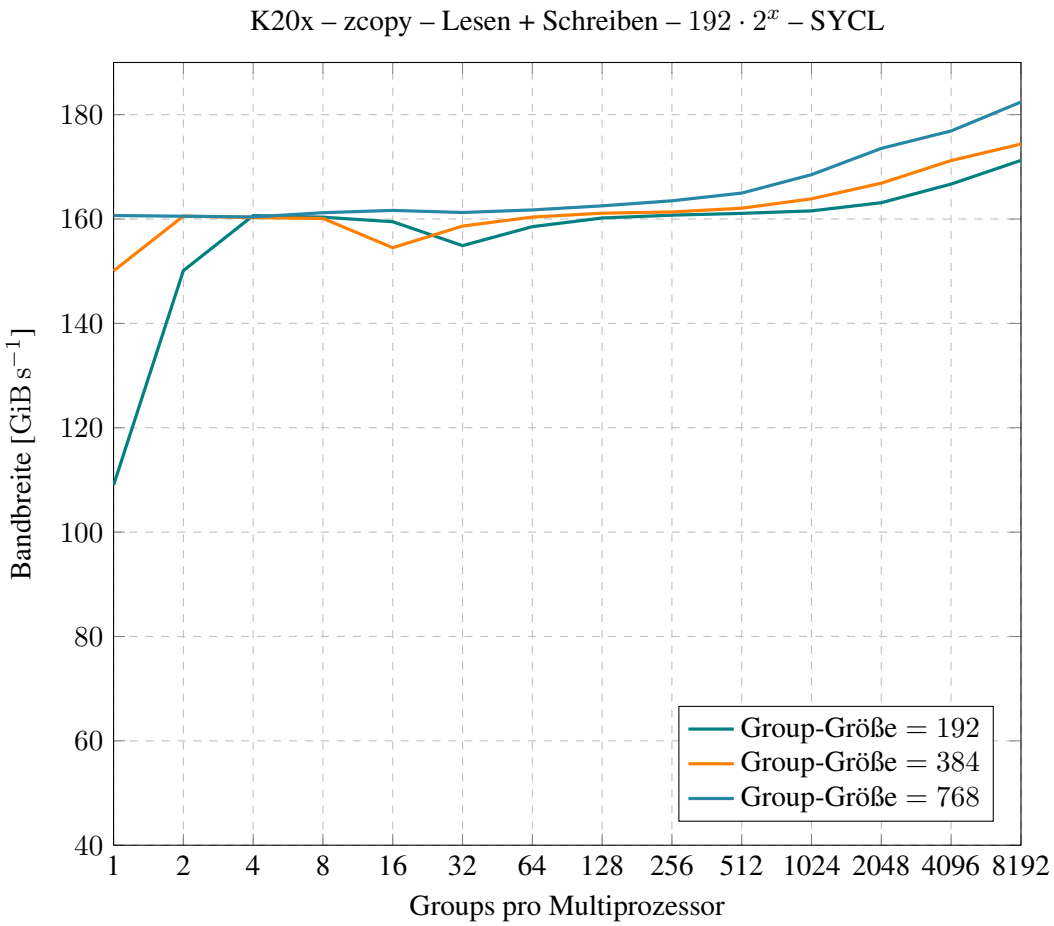


Abbildung E.11: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Lesen und Schreiben, SYCL)

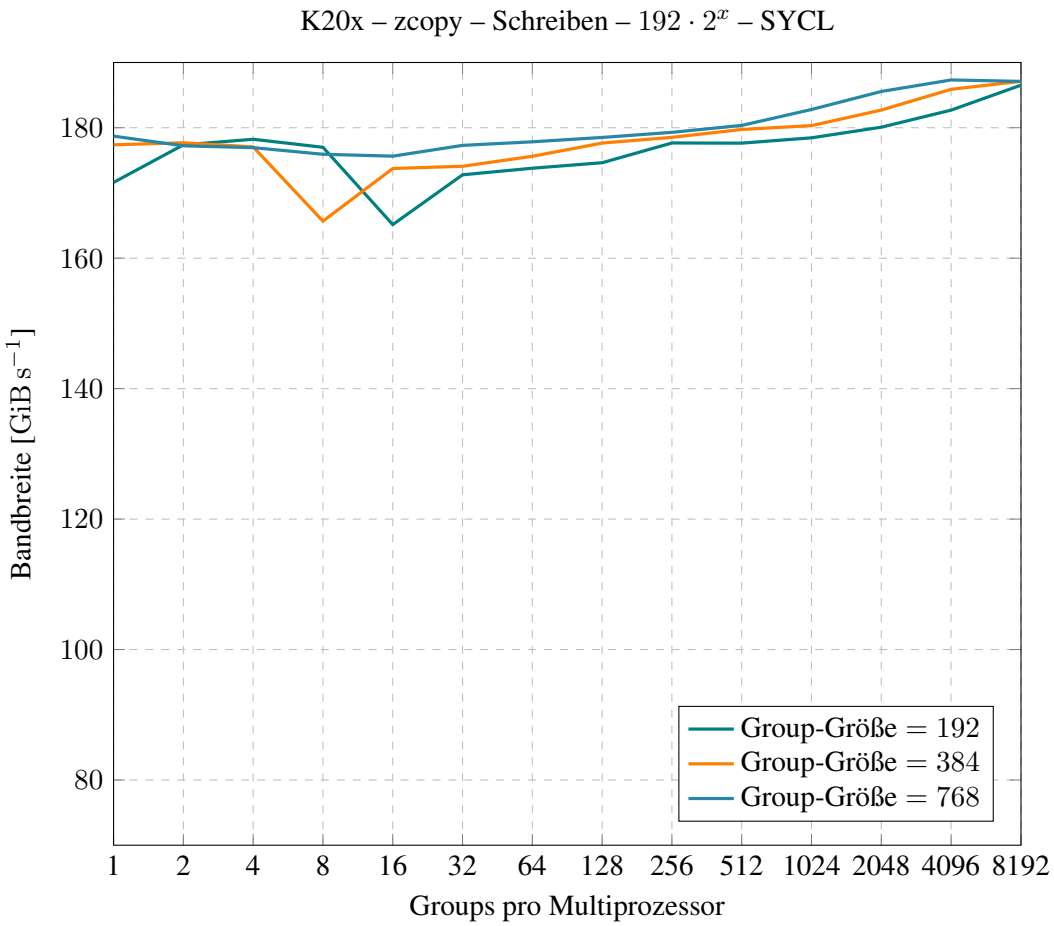


Abbildung E.12: zcopy: K20x-Bandbreite für 192er-Vielfache ($n = 88080384$, Schreiben, SYCL)

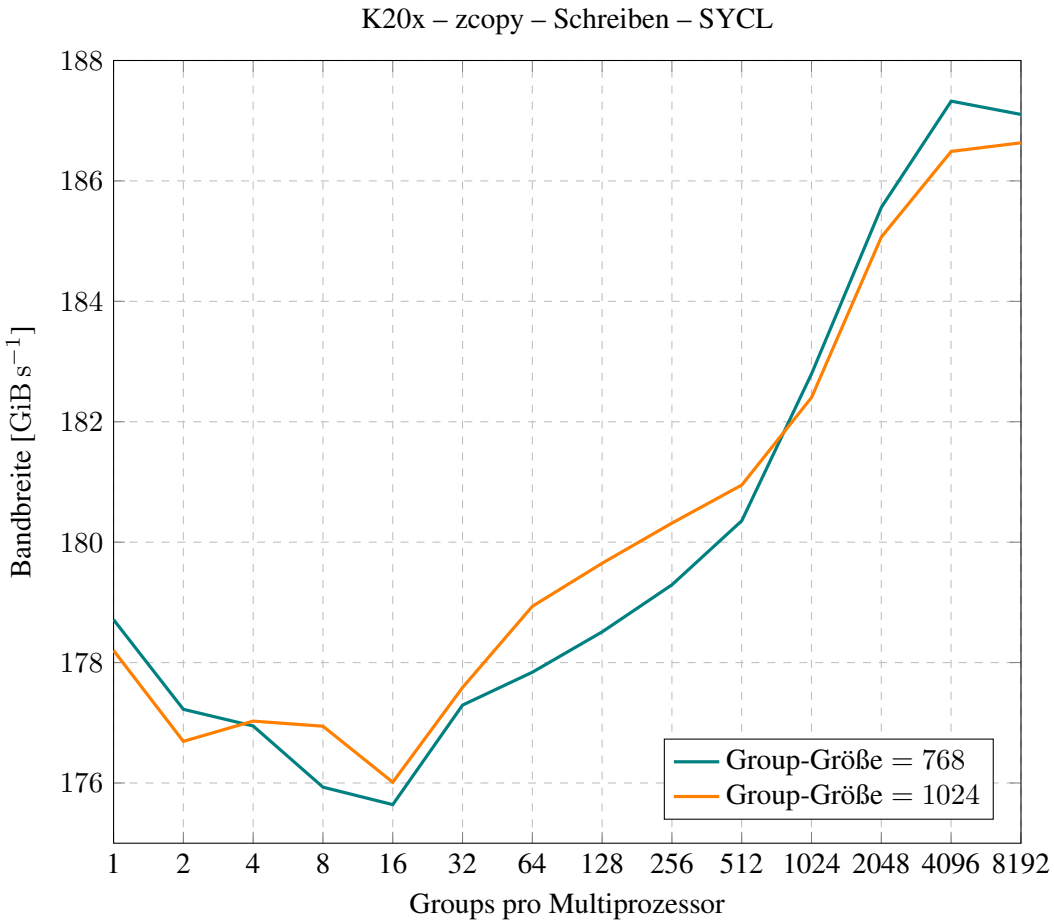


Abbildung E.13: zcopy: K20x-Bandbreite für Zweierpotenzen ($n = 117440512$) und 192er-Vielfache ($n = 88080384$, Schreiben, SYCL)

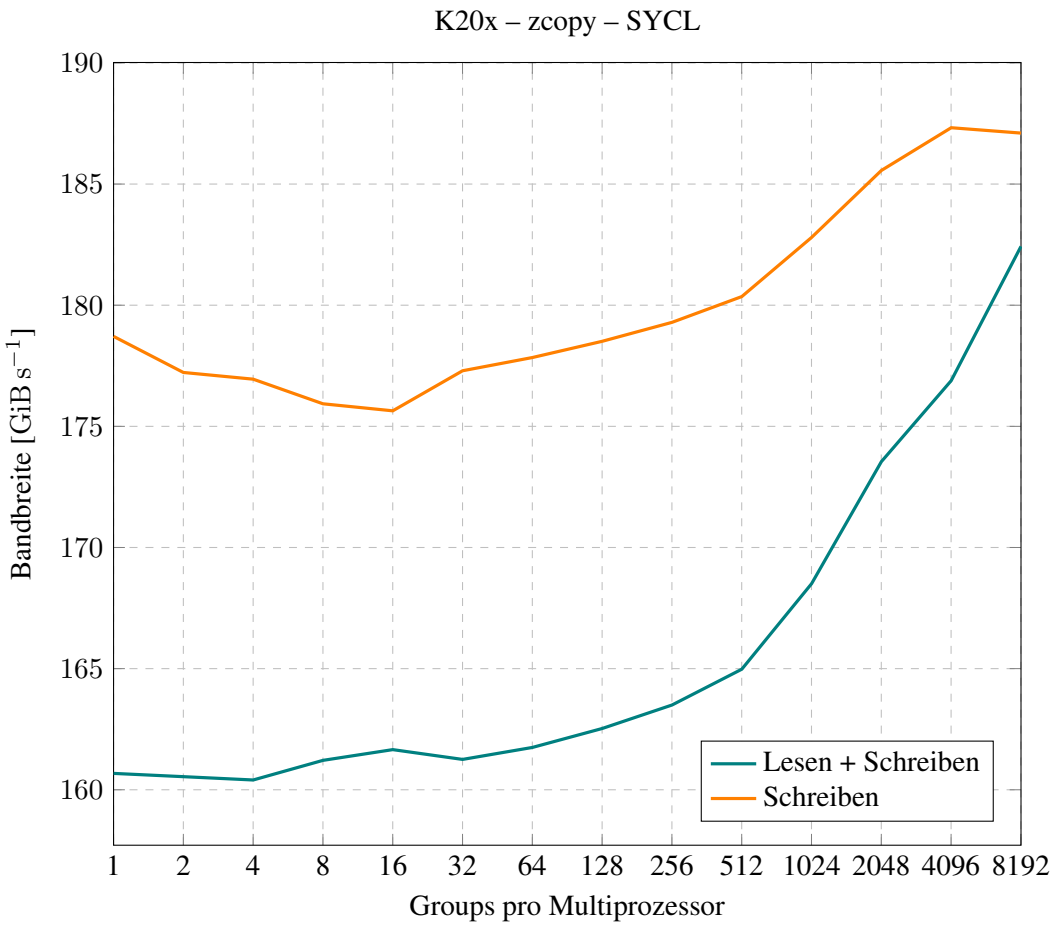


Abbildung E.14: zcopy: K20x-Bandbreite für 768er-Groups ($n = 88080384$, SYCL)

E.4 HIP-Reduction (AMD)

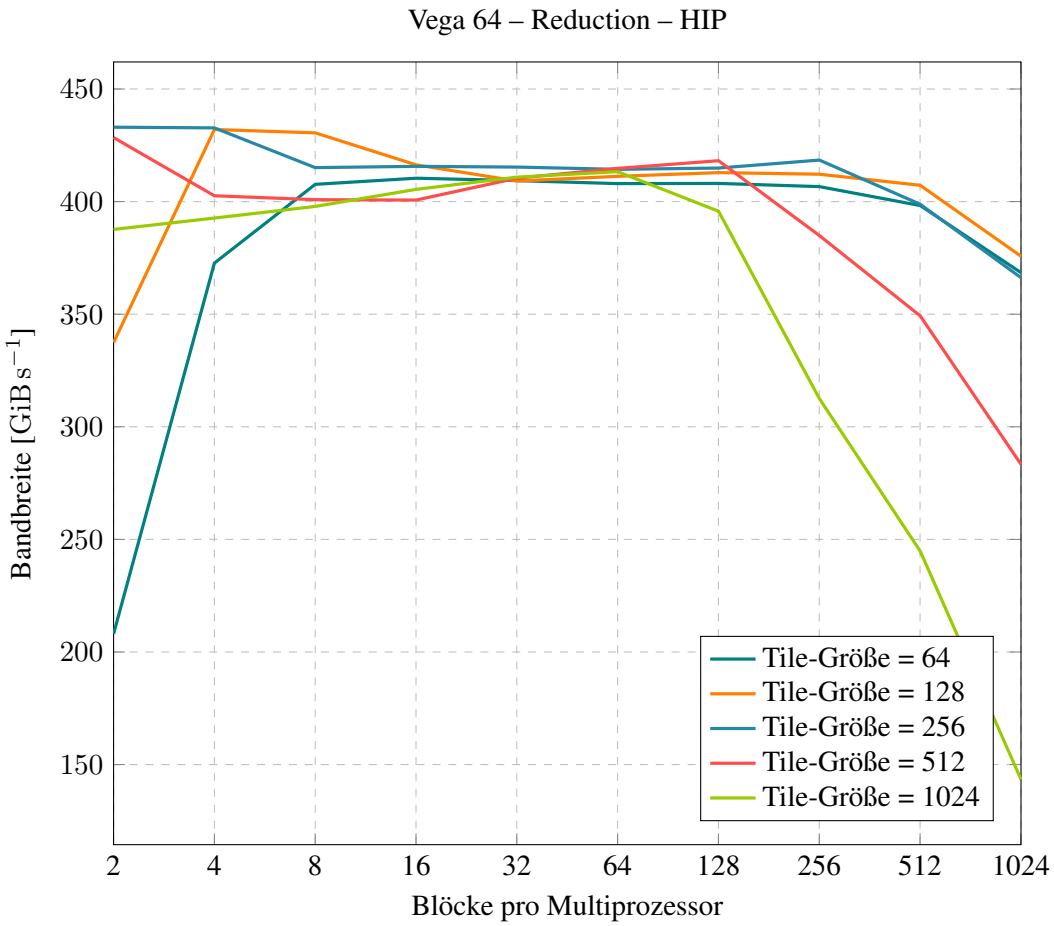


Abbildung E.15: Reduction: Bandbreite der Vega 64 (HIP)

E.5 HIP-Reduction (NVIDIA)

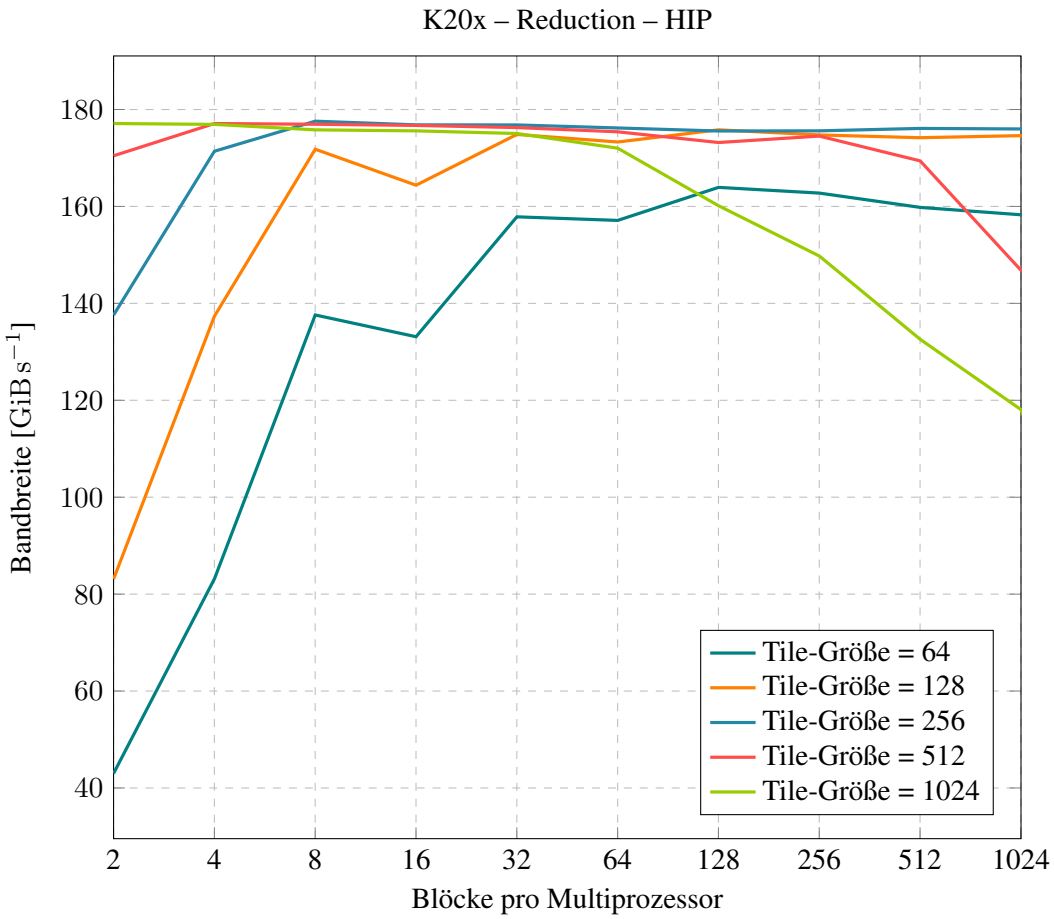


Abbildung E.16: Reduction: Bandbreite K20x (HIP)

E.6 SYCL-Reduction

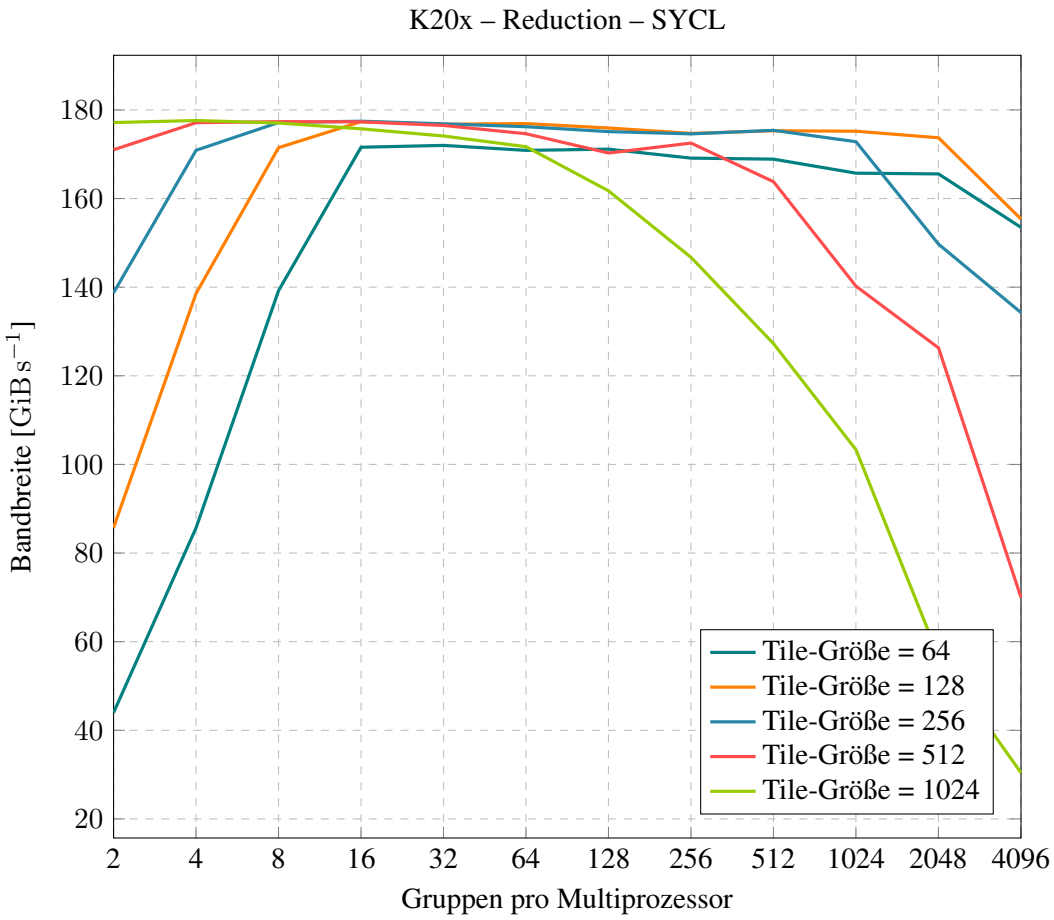


Abbildung E.17: Reduction: Bandbreite K20x (SYCL)

E.7 HIP-N-Body (AMD)

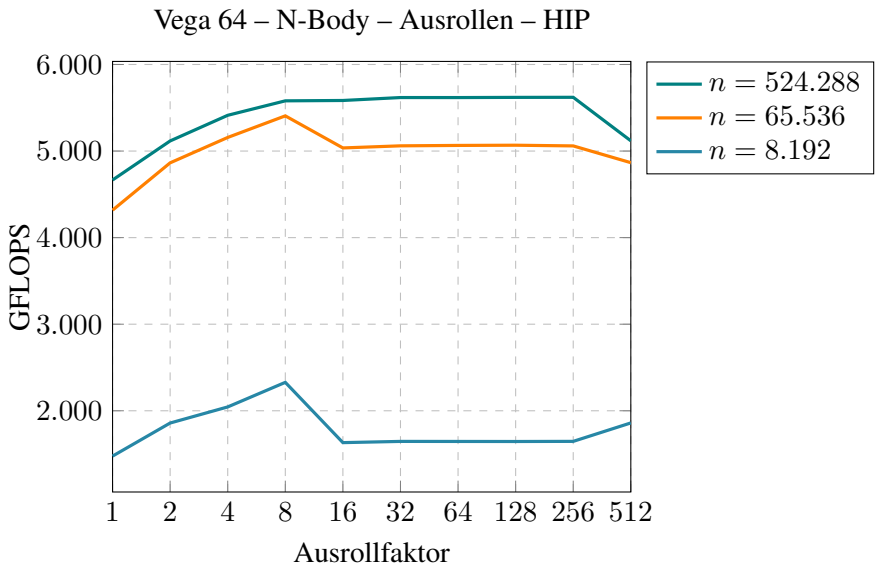


Abbildung E.18: Performanzgewinn der Vega 64 durch das Ausrollen der Schleife (HIP)

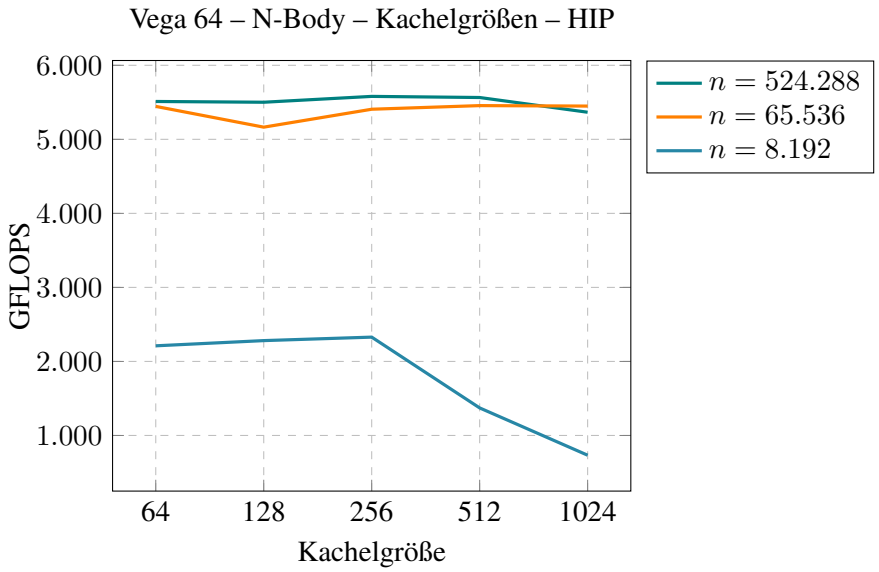


Abbildung E.19: N-Body: Performanz der Vega 64 bei verschiedenen Kachelgrößen (HIP)

E.8 HIP-N-Body (NVIDIA)

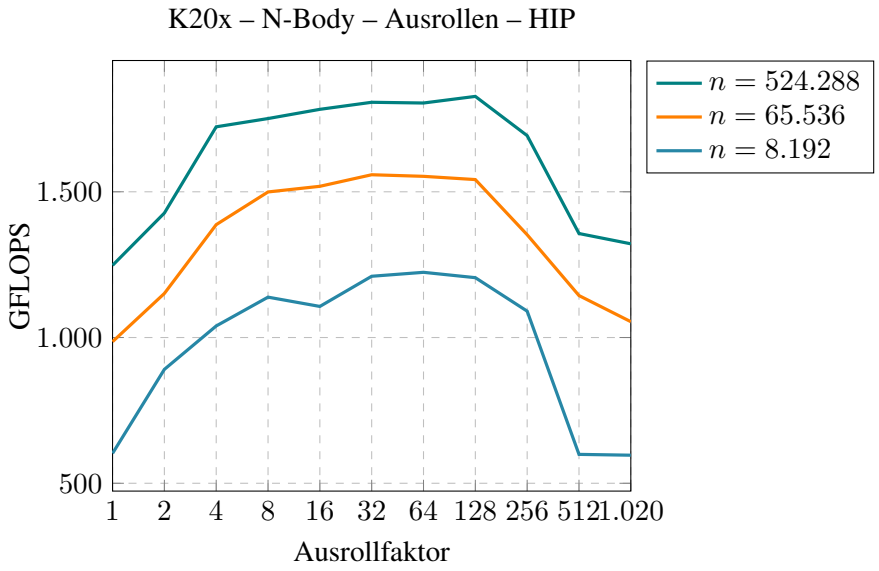


Abbildung E.20: N-Body: K20x-Performanzgewinn durch das Ausrollen der Schleife (HIP)

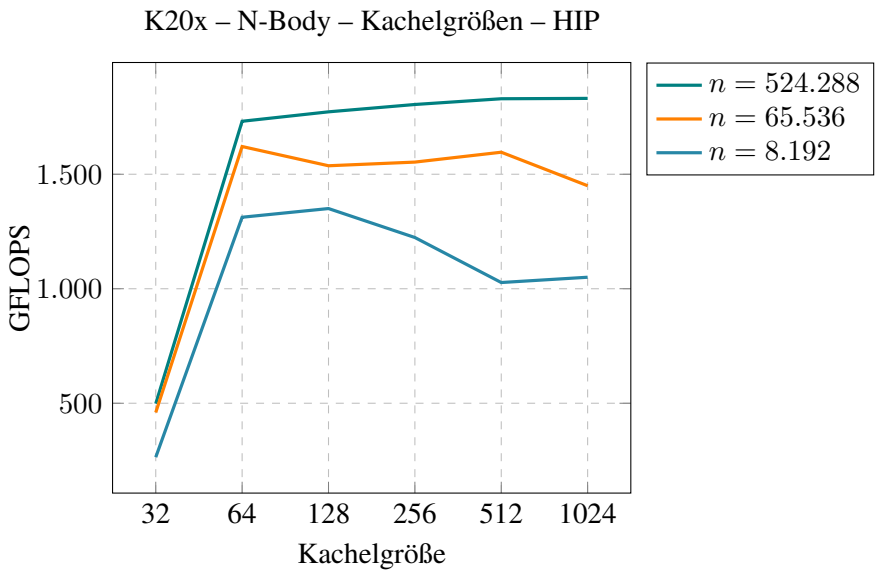


Abbildung E.21: N-Body: K20x-Performanz bei verschiedenen Kachelgrößen (HIP)

E.9 SYCL-N-Body

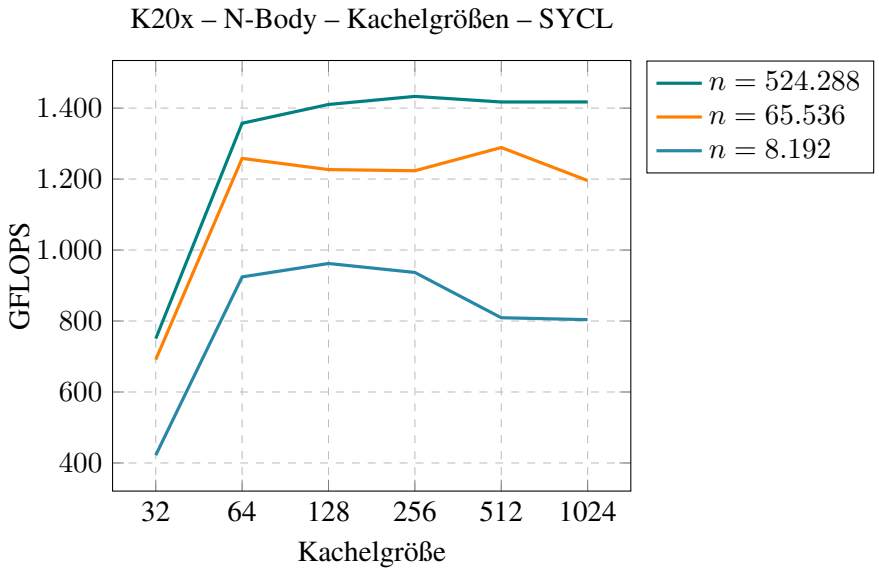


Abbildung E.22: N-Body: K20x-Performanz bei verschiedenen Kachelgrößen (SYCL)