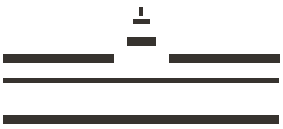


Westfälische
Wilhelms-Universität
Münster

Implementierung der LR-Zerlegung und des Mersenne-Twister mit der SkelCL-Bibliothek

Bachelorarbeit

Sebastian Mißbach

5. März 2013

Inhaltsverzeichnis

1 Einführung

Die derzeitigen Methoden zur Programmierung von modernen GPUs und CPUs, wie OpenCL [1] oder CUDA, sind aufwändig in der Entwicklung, haben eine hohe Fehleranfälligkeit und bei der Entwicklung von Programmen für Multi-GPU oder -CPU Systeme werden umfassende Kenntnisse der parallelen Programmierung benötigt. Für weniger erfahrene Programmierer ist dies ein Problem. Sie stehen vor der Aufgabe den Speicher manuell verwalten zu müssen und einer Schnittstelle auf niedrigem Abstraktionsniveau.

Algorithmische Skelette abstrahieren bekannte Muster der parallelen Berechnung, Kommunikation und Interaktion, dadurch ermöglichen Skelette die abstrakte Beschreibung von Algorithmen [2]. SkelCL, eine Skelettbibliothek entwickelt an der Universität Münster von der Arbeitsgruppe Parallele und Verteilte Systeme [3], erleichtert die parallele Programmierung durch das Bereitstellen von Skeletten und abstrakten Datentypen. Dadurch vermeidet SkelCL komplexe Programmierung auf niedrigem Abstraktionsniveau. SkelCL beruht auf dem OpenCL-Standard, dadurch ist es wie OpenCL Hardware und Hersteller unabhängig.

In dieser Bachelorarbeit werden Anwendungen mit der SkelCL-Bibliothek implementiert. Dadurch sollen neue Anwendungsbeispiele für die Nutzung von SkelCL geliefert werden. Die erstellten SkelCL-Anwendungen werden anschließend mit OpenCL- oder CUDA-Anwendungen verglichen. Um Anwendungsentwicklern unterschiedliche Einsatzgebiete von SkelCL aufzuzeigen, sind die OpenCL- oder CUDA-Anwendungen aus verschiedenen Anwendungsdomänen ausgewählt worden.

Anhand der Anwendungen wird die Benutzbarkeit und die Performance von SkelCL im Vergleich zu OpenCL evaluiert.

Diese Bachelorarbeit ist in sechs Kapitel eingeteilt. Im Anschluss an diese Einführung folgt die Einführung in die Grundlagen der Thematik. In dem dritten, vierten und fünften Kapitel werden die implementierten Anwendungen der LR-Zerlegung und des Mersenne-Twisters vorgestellt und evaluiert. Zuletzt wird mit einem Fazit diese Bachelorarbeit abgeschlossen.

2 Grundlagen

2.1 Open Computing Language

Die Open Computing Language (OpenCL [1]) wurde 2008 von der OpenCL Working Group definiert als offener Industrie-Standard für General Purpose Computation On Graphics Processing Unit (GPGPU). Im Gegensatz zu anderen GPGPU Ansätzen ist OpenCL ein plattformunabhängiges Programmiermodell für heterogene Systeme. Ein heterogenes System kann verschiedene Prozessortypen enthalten, in OpenCL sind dies zum Beispiel GPUs und CPUs. Die Programmierschnittstelle von OpenCL ist als Standard festgelegt und unterschiedliche Hersteller bieten Implementierungen an. Die Zielgruppe von OpenCL sind erfahrene Programmierer, die plattformunabhängige und effiziente Programme schreiben wollen.

Die Kernkomponenten von OpenCL werden durch drei Modelle beschreiben:

- Plattform-Modell
- Ausführungs-Modell
- Speicher-Modell

2.1.1 Plattform-Modell

Das *Plattform-Modell* beschreibt den abstrakten Aufbau eines OpenCL-Systems. Eine *Plattform* besteht aus einem Host, der mit einem oder mehreren OpenCL-*Devices* verbunden ist. Da OpenCL-Devices über unterschiedliche Architekturen verfügen können, wird durch eine abstrakte Aufteilung der Hardware eine Beschreibung eines OpenCL-Devices ermöglicht. Daher ist ein Device in OpenCL zunächst aufgeteilt in eine oder mehrere *Compute Units* (CUs). Diese sind ein weiteres Mal aufgeteilt in eine oder mehrere *Processing Elements* (PEs). Die OpenCL-Anwendung sendet ausgehend vom Host Befehle an die OpenCL-Devices, die ihrerseits die Berechnung innerhalb der PEs ausführen.

2.1.2 Ausführungs-Modell

Die Ausführung einer OpenCL-Anwendung kann in zwei Teile zerlegt werden:

Kernels
: Die Kernels sind Funktionen, die in einer oder mehreren Instanzen, parallel auf einem oder mehreren OpenCL-Devices ausgeführt werden.

Host Programm
: Das Host Programm wird durch den Host ausgeführt. Durch dieses Programm werden die benutzen OpenCL-Devices verwaltet, außerdem steuert und verwaltet das Host Programm die Ausführung der Kernels. Das Host Programm definiert wie viele Kernel-Instanzen gestartet werden sollen.

Die Kernel-Instanzen werden als Work-items bezeichnet, von diesen können mehrere zu einer Work-group zusammengefasst werden. Jedes Work-item besitzt eine eindeutige `globalID` und zusätzlich eine `localID`. Letztere dient der Identifikation des Work-items innerhalb der Work-group. OpenCL stellt sicher, dass jedes Work-item auf einem PE ausgeführt wird und alle Work-items derselben Work-group auf einer CU.

Dies gibt die Möglichkeit die Work-items innerhalb einer Work-group miteinander zu synchronisieren. OpenCL bietet keine Möglichkeit an, Work-items aus unterschiedlichen Work-groups zu synchronisieren.

2.1.3 Speicher-Modell

Jedem Work-item stehen unterschiedliche Speicherbereiche zur Verfügung:

Globaler Speicherbereich
: Erlaubt allen Work-items aus allen Work-groups Lese- und Schreibzugriff. Der globale Speicherbreich, stellt den größten Speicherbereich zur Verfügung.

Konstanter Speicherbereich
: Ein Teil des globalen Speichers, der nur gelesen werden kann und über die Ausführung konstant bleibt.

Lokaler Speicherbereich
: Sichtbar für alle Work-items einer Work-group, diese können lesend und schreibend auf diesen Speicherbereich zurückgreifen. Der Speicherbereich kann dazu genutzt werden, um Variablen innerhalb einer Work-group zu teilen, zum Beispiel zum Zwecke der Synchronisierung der Ausführung.

Privater Speicherbereich
: Sichtbar für ein einzelnes Work-item, das lesend und schreibend auf den Speicher zurückgreifen kann.

2.2 Skelettbibliotheken

Die derzeitigen Methoden der parallelen Programmierung, wie OpenCL oder CUDA, sind aufwändig in der Programmierung. Daher ist es ein Ansatz, den Programmierer zu entlasten. Durch eine Bibliothek werden der Kontrollfluß, die Verschachtelung und die Ressourcen Kontrolle des parallelen Programms übernommen. Diese Bibliothek wahrt ferner die Portabilität des parallelen Programms. Ausgedrückt wird das parallele Programm in der Folge durch eine Menge von Skeletten, daher rührt der Name der Skelettbibliothek.

Ein Skelett abstrahiert ein Muster der parallelen Berechnung, Kommunikation und Interaktion, dadurch wird eine abstrakte Beschreibung eines Algorithmus ermöglicht. Die Skelettbibliothek liefert Skelette über eine Programmierschnittstelle aus, die vom Benutzer verwendet werden kann, um sein Programm zu beschreiben.

2.3 SkelCL

SkelCL, eine Skelettbibliothek entwickelt an der Universität Münster von der Arbeitsgruppe Parallele und Verteilte Systeme [3], erleichtert die parallele Programmierung durch das Bereitstellen von Skeletten und abstrakten Datentypen. Dadurch vermeidet SkelCL komplexe Programmierung auf niedrigem Abstraktionsniveau. SkelCL beruht auf dem OpenCL-Standard, dadurch ist es wie OpenCL Hardware und Hersteller unabhängig.

Die vier Skelette *Map*, *Zip*, *Reduce* und *Scan* die SkelCL in der aktuellen Version zur Verfügung stellt, können in vielen Anwendungen vielseitig verwendet werden. Im Folgenden werden die vier Skelette beschrieben:

Map
: Liefert für einen unären Operator f und einen Vektor $(x_1, x_2, \ldots, x_n)$ den Vektor $(f(x_1), f(x_2), \ldots, f(x_n))$.

Zip
: Liefert für einen gegebenen assoziativen binären Operator $\oplus$ und der Vektoren $(x_1, x_2, \ldots, x_n)$, $(y_1, y_2, \ldots, y_n)$ den Vektor $(x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_n \oplus y_n)$.

Reduce
: Liefert für einen gegebenen assoziativen binären Operator $\oplus$ und einen Vektor $(x_1, x_2, \ldots, x_n)$ den Wert $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

Scan
: Liefert für einen gegebenen assoziativen binären Operator $\oplus$ und einen Vektor $(x_1, x_2, \ldots, x_n)$ den Vektor $(x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n)$.

Die Skelette akzeptieren die beiden abstrakten Datentypen *Vector* und *Matrix*, die von SkelCL zur Verfügung gestellt werden. Die Ausführung für eine Matrix geschiet analog zur vorgestellten Ausführung des Vektors, dazu wird eine $m \times n$-Matrix A als Zeilenvektor aufgefasst, das heißt:

$$A = (a_{00}, \ldots, a_{0,n-1}, a_{10}, \ldots, a_{1,n-1}, \ldots, a_{m-1,0}, \ldots, a_{m-1,n-1}).$$

Die zur Verfügung gestellten Datentypen von SkelCL befreien den Entwickler von der Verwaltung des Datentransfers zwischen dem Host-Speicher und den OpenCL-Devices. Außerdem kann SkelCL den Datentransfer optimieren, indem der Transfer so lange wie möglich oder ganz vermieden wird.

Ein SkelCL-Beispielprogramm kann in Listing 2.1 gesehen werden. Es handelt sich um die Berechnung des Skalarproduktes mithilfe der beiden Skelette `Zip` und `Reduce`, die auf die beiden Operanden vom Datentyp `Vector` angewandt werden. Anschließend wird das Ergebnis auf der Standardausgabe ausgegeben.

Jedes SkelCL-Programm folgt einem grundsätzlichen Aufbau. Dieser kann im Beispielprogramm verfolgt werden und besteht aus den folgenden Schritten:

1. Initialisierung von SkelCL (in Zeile 2).
2. Definieren der Eingabe (in den Zeilen 5 und 6).
3. Definieren des Skelett-Programms (in den Zeilen 12 bis 24).
4. Ausführen des Skelett-Programms (in Zeile 27).

SkelCL verringert den Aufwand bei der Entwicklung von Programmen für Multi-GPU und -CPU Systeme. Dies wird erreicht, indem die SkelCL Datentypen auf mehreren OpenCL-Devices verfügbar sind. Auf diese Weise wird dem Entwickler die Speicherverwaltung auf niedrigem Abstraktionsniveau abgenommen. Durch *Data Distributions* kann angegeben werden, wo sich die Daten einer Matrix oder eines Vektors befinden sollen. Im Folgenden sind die Distributionen, die SkelCL zur Verfügung stellt, beschrieben:

single
: In der single Distribution befinden sich die gesamten Daten auf einem einzelnen Device.

copy
: Die copy Distribution kopiert die gesamten Daten auf jedes Device.

block
: In der block Distribution speichert jedes Device einen zusammenhängenden Teil der Daten.

```
1  // 1. Initialize SkelCL
2  skelcl::init(skelcl::nDevices(1));
3
4  // 2. Define two input vectors.
5  Vector<int> A(1024);
6  Vector<int> B(1024);
7
8  // Initialize input vectors.
9  init(A.begin(), A.end());
10 init(B.begin(), B.end());
11
12 // 3. Define the skeleton program.
13 Zip<int(int,int)> MUL(
14   "int func(int x, int y) { return x * y; }"
15 );
16
17 Reduce<int(int)> SUM(
18   "int func(int x, int y) { return x + y; }", "0"
19 );
20
21 // 4. Input vectors.
22 Vector<int> C = SUM( MUL( A, B ) );
23
24 // Print the result.
25 std::cout << C.front() << std::endl;
```

Listing 2.1: Berechnung des Skalarprodukt in SkelCL

3 Standard LR-Zerlegung

Ein Standardalgorithmus zur Lösung linearer Gleichungssysteme aus dem Fachbereich der linearen Algebra und der Numerik ist die LR-Zerlegung. Dabei sei A eine invertierbare $n \times n$-Matrix, dann existiert eine Permutationsmatrix P (die Matrix hat in jeder Zeile genau einen 1 Wert sonst 0 Werte), eine linke untere Dreiecksmatrix L mit Diagonalelementen 1 und eine obere Dreiecksmatrix R mit $PA = LR$ [10].

In diesem Kapitel wird die LR-Zerlegung mit der SkelCL-Bibliothek implementiert. Zunächst wird der Algorithmus der LR-Zerlegung vorgestellt, anschließend wird auf die Implementierung der SkelCL-Anwendung eingegangen. Zuletzt soll die SkelCL-Anwendung mit einer OpenCL-Anwendung verglichen werden. Das Accelerated Parallel Processing (APP) Software Development Kit (SDK) von AMD [4], stellt die OpenCL-Anwendungen bereit.

3.1 Algorithmus

Für die LR-Zerlegung wird der Algorithmus aus Algorithmus 1 verwendet.

Input : $n \times n$-Matrix
for $i = 0, \ldots, n-2$ **do**
  for $j = i+1, \ldots, n-1$ **do**
    $r = a_{ji}$ / a_{ii};
    $a_{ji} = r$;
    for $k = i+1, \ldots, n-1$ **do**
      $a_{jk} = a_{jk} - a_{ik} * r$;
    end
  end
end

Algorithmus 1: LU Decomposition

Der Algorithmus führt sukzessive Manipulationen innerhalb der Eingabematrix A im Speicher (in-place Verfahren) aus. Für die linke untere Dreiecksmatrix ($i = j$) wird $r = a_{ji}/a_{ii}$ abgespeichert, der Diagonalwert (für $j = i$) wird jedoch nicht gespeichert (in Zeile 3 und 4). Um die rechte obere Dreiecksmatrix zu

berechnen, wird anschließend von der j-ten Zeile das r-fache von der i-ten Zeile subtrahiert (in Zeile 5 bis 7). Auf diese Weise entsteht im linken unteren Bereich der Eingabematrix A die linke untere Dreiecksmatrix L – ohne die Diagonalwerte – und im rechten oberen Bereich die rechte obere Dreiecksmatrix R.

Der Algorithmus kann durch zwei Skelette beschrieben werden. Dazu wird die innere Schleife (in den Zeilen 2 bis 8) in zwei Skelette zerlegt. Das erste Skelett kann die Zeilen 3 und 4 abbilden, also übernimmt es die Berechnung der linken untere Dreiecksmatrix. Das zweite Skelett bildet die Schleife zur Berechnung der rechten oberen Dreiecksmatrix ab (Zeilen 5 bis 7). Die berechneten Werte der linken unteren Dreiecksmatrix werden zur Berechnung benutzt.

3.2 Implementierung

Die SkelCL-Implementierung der LR-Zerlegung besteht im Wesentlichen aus den folgenden Schritten:

- Initialisierung von SkelCL.
- Definieren der Eingabe.
- Definieren des Skelett-Programms.
- Die Daten auf das Device kopieren.
- Ausführen des Skelett-Programms durch die Eingabe der Parameter.
- Die Daten auf den Host kopieren.

Die SkelCL-Implementierung ist in Listing 3.1 dargestellt.

Über den Aufruf der Funktion `skelcl::init` (Zeile 2) wird SkelCL initialisiert. Der Aufruf entspricht der OpenCL-Initialisierung, die zumeist aus der Auswahl der Devices, dem Erstellen des Kontextes und dem Erstellen der Command-queue besteht. In diesem Fall wird vereinfacht ein beliebiges Device (`nDevices(1)`) ausgewählt. Die Implementierung ermöglicht aber auch eine Auswahl eines Devices nach einer vorgegeben Plattform- und Devicenummer.

Die SkelCL-Implementierung versucht durch die Benutzung von OpenCL-Vektortypen die Ausführung zu beschleunigen. Dadurch ist es in der SkelCL-Implementierung möglich, 4 Werte mit einem OpenCL-Vektor der Größe 4 zu verarbeiten. Dafür muss die gewählte Dimension der Matrix jedoch ein Vielfaches der Größe, der gewählten OpenCL-Vektorgröße (`VECTOR_SIZE` = 4) sein. Aus diesem Grund wird (in den Zeilen 4 bis 9) die Dimension der Matrix angepasst auf ein Vielfaches von `VECTOR_SIZE`. Im Anschluss wird die Eingabematrix erstellt und durch die Funktion `generateRandom` mit Zufallszahlen gefüllt (Zeilen 11 und 12).

Der nächste Schritt besteht aus der Definition des Skelettprogramms (Zeilen 18 bis 22). Die Beschreibung der einzelnen Skelette ist im Folgenden angegeben:

DECOMPOSE_L Diese Map-Skelettfunktion zerlegt eine Eingabematrix in eine linke untere Dreiecksmatrix. Dieses Map-Skelett entspricht den Zeilen 3 und 4 aus Algorithmus 1. Die berechneten Werte werden in die definierte Matrix `matrixL` (Zeile 26) abgelegt.

DECOMPOSE_R Diese Map-Skelettfunktion zerlegt die Eingabematrix in die rechte obere Dreiecksmatrix, dabei werden zusätzlich die Werte der linken unteren Dreiecksmatrix zur Lösung benötigt. Das Map-Skelett entspricht den Zeilen 5 bis 7 aus Algorithmus 1. Die Funktion verarbeitet jeweils ein OpenCL-Vektorelement, das aus `VECTOR_SIZE` Elementen der Eingabematrix besteht. Die berechneten Ergebnisse werden in der Eingabematrix abgelegt.

COMBINE Mit dieser Funktion werden zwei Eingabematrizen in eine Matrix kombiniert. Damit kann die linke untere Dreieckmatrix in die rechte obere Dreiecksmatrix kombiniert werden.

Die SkelCL-Implementierung nutzt eine feste vorgebene Work-group Größe. Vorgegeben wird diese durch SkelCL. Für `DECOMPOSE_R` werden so $16 \times 16 \times 1$ Work-groups, für `DECOMPOSE_L` UND `COMBINE` $256 \times 1 \times 1$ Work-groups gebildet. Außerdem kann durch die SkelCL-Funktion `setWorkGroupSize(size)` die Work-Group Größe angepasst werden.

Im Anschluss an die Definition des Skelettprogramms folgt die Ausführung der LR-Zerlegung (Zeilen 27 bis 31), indem die Parameter an die beiden Skelette übergeben werden.

Die beiden Map-Skelette `DECOMPOSE_L` und `DECOMPOSE_R` werden auf eine Menge von Indizes angewandt. Die Map-Skelettfunktion verarbeitet die Matrixelemente mit den vorgegebenen Indizes. SkelCL stellt dafür die beiden Typen `IndexVector` und `IndexMatrix` zur Verfügung. Der Datentyp `IndexVector` (benutzt in Zeile 28), ist ein SkelCL-Vektor gefüllt mit Indizes (`Index`). Die `IndexMatrix` (benutzt in Zeile 29 und 32) ist eine SkelCL-Matrix gefüllt mit Index-Punkten (`IndexPoint`).

In der dargestellten Implementierung ist kein direkter Datentransfer zu sehen. SkelCL übernimmt das Kopieren der Eingabematrix vom Host zum Device und das Kopieren der Ergebnismatrix vom Device zum Host. SkelCL stellt sicher, dass die Daten der Eingabematrix bis zur ersten Ausführung von `DECOMPOSE_L` (Zeile 28) auf das Device kopiert wurden. Beim ersten Zugriff auf `matrix` nach der Ausführung (in Zeile 34) werden die Daten der gelösten Eingabematrix wieder auf den Host kopiert.

Die Implementierung verfügt über eine einfache Verifikation. Diese besteht daraus, die Matrix mit der LR-Zerlegung sequentiell auf dem OpenCL-Host zu lösen. Anschließend wird die gelöste Referenzmatrix mit der vom Device gelösten Matrix verglichen. Die Abweichung von der Referenzmatrix kann auf der Kommandozeile angezeigt werden.

```
1  // 1. Initialize SkelCL
2  skelcl::init(skelcl::nDevices(1));
3
4  // The matrix dimension must be a mutiple of VECTOR_SIZE.
5  unsigned int effDim;
6  if(dimension % VECTOR_SIZE != 0) {
7    effDim = dimension - (dimension % VECTOR_SIZE) + VECTOR_SIZE;
8  else {
9    effDim = dimension;
10 }
11
12 // 2. Define the input matrix.
13 Matrix<double> matrix({effDim, effDim});
14 generateRandom<double>(matrix.begin(), matrix.end(), 1.0, 3.0);
15
16 // 3. Define the skeleton program.
17 // Decomposes input matrix into a lower triangular matrix.
18 Map<void(Index)> DECOMPOSE_L(/* ... */);
19 // Decomposes input matrix into an upper triangular matrix.
20 Map<void(IndexPoint)> DECOMPOSE_R(/* ... */);
21 // Combines lower triangular matrix and upper triangular matrix
22 Map<void(IndexPoint)> COMBINE(/* ... */);
23
24 // 4. Decompose the input matrix.
25 const unsigned int dim = matrix.rowCount();
26 Matrix<double> matrixL({dim, dim});
27 for(unsigned int i = 0; i < dim - 1; ++i) {
28   DECOMPOSE_L( IndexVector(dim - i), matrix, matrixL, i );
29   IndexMatrix index({dim - i, dim - i});
30   DECOMPOSE_R( index, matrix, matrixL, i );
31 }
32 COMBINE( IndexMatrix({dim, dim}), out(matrix), matrixL );
33
34 printMatrix(matrix);
```

Listing 3.1: Vereinfachte Darstellung der LR-Zerlegung in SkelCL

3.3 Evaluierung

Die erstellte SkelCL-Anwendung aus Abschnitt 3.2 ist eine einfache Implementierung einer LR-Zerlegung. Der Programmcode ist im Vergleich mit einer OpenCL-Anwendung aus dem APP SDK weniger komplex und umfangreich (etwa ein

Drittel weniger Programmzeilen). Des Weiteren übernahm SkelCL die Speicherverwaltung, die OpenCL-Initialisierung und den Datentransfer der SkelCL-Datentypen Matrix und Vektor.

Beide Anwendungen wählten unterschiedliche Work-group Größen. Die Eingabematrix wurde sukzessive durch die LR-Zerlegung im Speicher (in-place) zerlegt. Daher wurde in der OpenCL-Anwendung die Work-group Größe an das die lösende Restmatrix (die noch nicht gelöste Teilmatrix) angepasst. Da jedes Work-item 4 Zahlen verarbeitete, wurden nicht quadratische Work-group Größen gewählt. Dazu ein vereinfachtes Beispiel: Lag eine 32×32-Eingabematrix vor, so wurde eine 8×32 Work-group in der ersten Iteration der LR-Zerlegung gewählt. In weiteren Iterationen der LR-Zerlegung wurden dann kleinere Work-group Größen (7×28, 6×24, usw.) gewählt.

In SkelCL ist diese Wahl der Work-group Größe durch `setWorkGroupSize` nicht möglich. Daher wurde für das Map-Skelett, das die Restmatrix löst (`DECOMPOSE_R` vgl. Abschnitt 3.2), eine feste Work-group Größe (zum Beispiel: 16×16) gewählt. Dies konnte dazu führen, dass wenn das zu lösende Restproblem sehr klein wurde (zum Beispiel: Dimension der Restmatrix kleiner als 16), zu viele Work-items ausgewält wurden. Wenn die Dimension der Restmatrix kein Vielfaches der Work-group war, führte dies ebenfalls zu einer Auswahl von zu vielen Work-items. Vor den Messungen wurden jeweils auf dem Device unterschiedliche Work-group Größen (8×8, 16×16 und 32×32) getestet. Die Work-group Größe, die die beste Performance lieferte wurde für den Vergleich mit der OpenCL-Anwendung ausgewählt.

3.3.1 Messungen

Die SkelCL-Anwedung aus Abschnitt 3.2 wurde mit einer OpenCL-Anwendung aus dem APP SDK verglichen. Dazu wurden eine Reihe von Messungen durchgeführt. Die SkelCL- und die OpenCL-Anwendung zerlegten dazu eine 2000×2000- und eine 4000×4000-Matrix, gefüllt mit Zufallszahlen (double precision floating point numbers). Beide Anwendungen mussten dazu zunächst die Matrix vom Host auf das Device kopieren. Anschließend wurde die LR-Zerlegung angewandt. Zuletzt musste die Matrix vom Device auf den Host kopiert werden.

Gemessen wurde die Zeit um die Matrix zwischen Host und Device zu Übertragen, und der benötigten Zeit um die LR-Zerlegung auf dem Device auszuführen. Ausgeführt wurden 100 Iterationen, über diese wurde die durchschnittliche Ausführungszeit ermittelt.

Für die Messungen wurden die folgenden Devices ausgewählt (Für die CPU in Klammern die Taktfrequenz und der verfügbare Hauptspeicher):

- AMD A8-3850 CPU (2,9 GHz, 16 GB)

- Nvidia GeForce GTX 480 (1,5 GB Hauptspeicher)

In dem System der GPU befand sich eine Intel Core 2 CPU (2,4 GHz, 2 GB). Die CPU zerlegte die 2000×2000-Matrix und die GPU die 4000×4000-Matrix.

Tabelle 3.1 zeigt, dass die OpenCL-Anwendung deutlich schneller war als die SkelCL-Anwendung auf der CPU. OpenCL war etwa 31 % Prozent schneller als die SkelCL-Anwendung. Die 16×16 Work-group Größe lieferte für die SkelCL-Anwendung die beste Performance.

Device	Anwendung	LR-Zerlegung	Total
AMD A8-3850	SkelCL (16×16)	$6,144$	$6,554$
	OpenCL	$4,683$	$5,010$

Tabelle 3.1: Ausführungszeiten der SkelCL- und OpenCL-Anwendung für die CPU in Sekunden (Durchschnitt von 100 Iterationen. In Klammern gewählte Work-group Größe).

Weniger deutlicher ist der Unterschied auf der GPU, gezeigt wird dies in Tabelle 3.2. Die OpenCL-Anwendung war für die 8×8 Work-group Größe der SkelCL-Anwendung um 12 % schneller. Für die gesammt Laufzeit der SkelCL-Anwendung war sogar kein Unterschied messbar (letzte Spalte in Tabelle 3.2).

Device	Anwendung	LR-Zerlegung	Totalzeit
GTX 480	SkelCL (8×8)	$5,507$	$7,389$
	OpenCL	$4,914$	$7,361$

Tabelle 3.2: Ausführungszeiten der SkelCL- und OpenCL-Anwendung für die GPU in Sekunden (Durchschnitt von 100 Iterationen. In Klammern gewählte Work-group Größe).

Die Entscheidung für die Art der Verarbeitung und der Wahl der Work-group Größen war in der Implementierung der LR-Zerlegung nicht effizient. Vorallem die Anzahl der Work-items pro Work-group, die nicht mitrechnen, sollte reduziert werden. SkelCL hätte es auch ermöglicht auf die OpenCL-Methoden der Work-group Größen Auswahl zurückzugreifen. Darauf wurde jedoch zugunsten einer weniger komplexen Anwendung verzichtet. Die Anwendung könnte somit durch Veränderungen in der Implementierung der LR-Zerlegung verbessert werden.

3.4 Schnittstellenbeschreibung

Die SkelCL-Anwendung der LR-Zerlegung verfügt über eine Kommandozeilenschnittstelle, die wie folgt aus dem Programmverzeichnis aufgerufen werden kann:

```
1 ./lud [Optionen]
```

In Tabelle 3.3 findet sich eine Beschreibung der möglichen Optionen.

Tabelle 3.3: Kommandozeilen Optionen

Kurzform	Langform	Beschreibung
-h	--help	Zeigt eine Hilfe an und beendet das Programm.
-v	--verify	Verifiziert die Ergebnismatrix.
-o	--outputFile arg	Gibt den Pfad zur Ausgabedatei an. Die berechnete Matrix wird in diese Datei geschrieben.
-n	--dimension arg	Gibt die Dimension der Matrix an. Wird kein Vielfaches von 4 angegeben, wird auf das nächste Vielfache von 4 skaliert.
-p	--platformId arg	Auswahl der Plattform Id von 0 bis $N-1$, wobei N die Anzahl der verfügbaren OpenCL-Plattformen ist.
-d	--deviceId arg	Auswahl des Devices Id von 0 bis $N-1$, wobei N die Anzahl der verfügbaren OpenCL-Devices ist.

4 Blockweise LR-Zerlegung

Die vorgestellte LR-Zerlegung in Kapitel 3 eignet sich gut zum Lösen von Matrizen mit einer Dimension kleiner als 1000. Aufgrund des Rechenaufwandes der LR-Zerlegung wird das Lösen von großen Matrizen jedoch zu einem Problem.

Eines dieser Probleme für große Matrizen ist, die nicht ausreichende Nutzung des lokalen Speicherbereichs. In der SkelCL-Anwendung wird auf die Benutzung des lokalen Speichers gänzlich verzichtet.

Ein weiteres Problem bei der Lösung aus Kapitel 3, ist die Wahl der Work-group Größen. Work-group Größen werden zwar fest gewählt, jedoch kann es vorkommen, dass mehr Work-items ausgewählt werden, als zur Berechnung notwendig sind. Dies kann zu Ressourcen Verschwendungen führen, die die Performance signifikant beeinflussen können.

All diese Probleme sollen durch ein alternatives Verfahren adressiert werden. Diese LR-Zerlegung soll den lokalen Speicher des Gerätes besser ausnutzen und die Auswahl der Work-group Größen verbessern. Erreicht werden soll dies durch eine Zerlegung der Matrix in Teilmatrizen fester quadratischer Größe, im Folgenden als Block bezeichnet. Daher auch der Name der blockweisen LR-Zerlegung.

In diesem Kapitel wird nach dem Vorbild einer OpenCL-Anwendung aus der Rodinia Benchmark Suite ([5]) eine blockweise LR-Zerlegung implementiert. Vor der Implementierung soll dazu zunächst der Algorithmus vorgestellt werden. Anschließend soll die Implementierung evaluiert werden.

4.1 Algorithmus

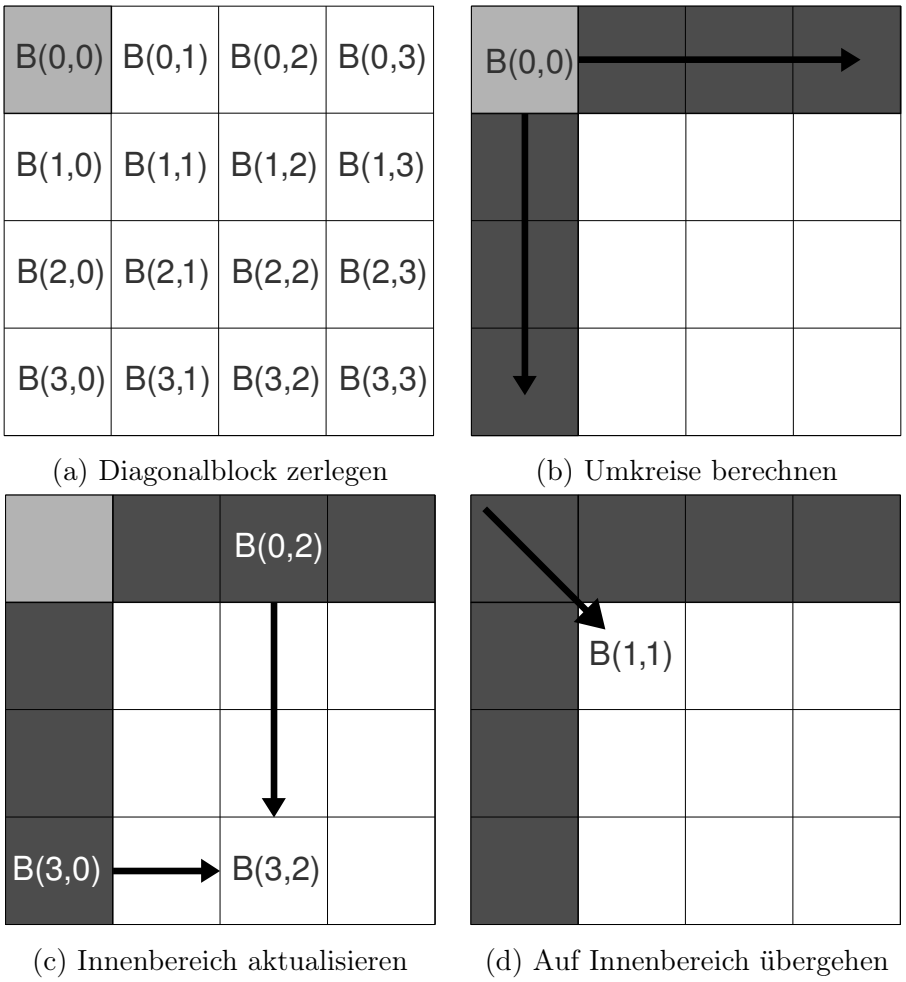
Die blockweise LR-Zerlegung teilt die Matrix in gleich große Blöcke mit vorgegebener Blockgröße auf. Für die einzelnen Blöcke liegen jedoch Abhängigkeiten vor. Denn die LR-Zerlegung führt die Zerlegung ausgehend vom linken oberen Matrixelement (a_{00}) zum rechten unteren Matrixelement ($a_{n-1,n-1}$) einer Eingabematrix mit Dimension n durch. Diese Abhängigkeit innerhalb der Matrix muss auch innerhalb der blockweisen LR-Zerlegung beibehalten werden.

Daher wird nicht jeder Block unabhängig zerlegt, sondern um Abhängigkeiten zu berücksichtigen, werden die folgenden drei Schritte nacheinander ausgeführt:

(a) Zerlegen des ersten Diagonalblocks.

(b) Berechnen der Umkreisblöcke mithilfe des Diagonalblocks.

(c) Berechnen des Innenbereichs mithilfe der Umkreisblöcke.



(a) Diagonalblock zerlegen (b) Umkreise berechnen

(c) Innenbereich aktualisieren (d) Auf Innenbereich übergehen

Abbildung 4.1: Verfahren der blockweisen LR-Zerlegung (gelöste Bereiche grau unterlegt).

Schritt (a) besteht darin die LR-Zerlegung auf den ersten Diagonalblock ($B(0,0)$) anzuwenden (vlg. Abbildung 4.1a). Anschließend kann in Schritt (b) (vlg. Abbildung 4.1b) mithilfe des Diagonalblocks $B(0,0)$, der Außenbereich der Matrix berechnet werden. Zur Lösung eines einzelnen Blocks aus dem Außenbereich wird lediglich der Diagonalblock benötigt.

Nach diesen beiden Schritten ist der Außenbereich der Matrix gelöst worden. Jedoch ist die Matrix nun inkonsistent, da keine Operationen auf dem Innenbereich stattgefunden haben. Daher wird in Schritt (c) der Innenbereich der Matrix

aktualisiert. Diese Aktualisierung ist möglich, indem von einem Block aus dem Innenbereich, das Ergebnis der Multiplikation der Umkreisblöcke subtrahiert wird. In Abbildung 4.1c kann der Block $B(3,2)$ über $B(3,2) = B(3,2) - B(3,0) * B(0,2)$ berechnet werden.

In der Folge wird der Innenbereich als erneute Eingabematrix betrachtet. Auf der die drei Schritte (a), (b) und (c) angewendet werden können (vlg. Abbildung 4.1d). Auf diese Weise wird die Matrix im Speicher sukzessive in die Matrizen L – gespeichert ohne die Diagonalelemente – und R zerlegt.

4.2 Implementierung

Wie die Standard LR-Zerlegung, verfügt auch die Implementierung der blockweisen LR-Zerlegung über die Schritte der Initialisierung, Definierung der Eingabe, Definierung und anschließende Ausführung des Skelettprogramms. Diese Schritte sollen im Folgenden erläutert werden.

Die Schritte der Initialisierung umfassen das Auswählen eines Devices. Dabei kann ein Device nach vorgegebener Plattform und Devicenummer oder ein Beliebiges zur Verfügung stehendes Devices ausgewählt werden. Im Anschluss wird die Eingabematrix definiert und mit Zufallszahlen befüllt. Die Matrix wird zuvor auf Vielfache der Blockgröße (`BLOCK_SIZE`) skaliert, da andere Matrizengrößen vom Algorithmus nicht verarbeitet werden können.

In der Definition des Skelettprogramms wurden die drei Schritte (a), (b) und (c) aus Abschnitt 4.1 durch drei Skelettfunktionen implementiert. Dabei konnten die OpenCL-Kernel der OpenCL-Anwendung aus der Rodinia Benchmark Suite benutzt werden. Im Folgenden werden die drei Skelettfunktionen beschrieben:

DECOMPOSE_DIAGONAL

Diese Map-Skelettfunktion führt die LR-Zerlegung auf einen Block (im lokalen Speicher) der Eingabematrix aus. Der Block wird von genau einer Work-group (mit `BLOCK_SIZE` Work-items) im lokalen Speicher zerlegt. Dazu wird eine Menge von Indizes übergeben, um die Aufgabe auf mehrere Work-items zu verteilen. Außerdem als zusätzliche Parameter die Eingabematrix aus dem globalen Speicher, lokaler Speicher für einen einzelnen Block und ein *Offset*-Wert zum Verschieben der Indizes. Zunächst wird ein einzelner Block mithilfe aller Work-items in den lokalen Speicher kopiert. Anschließend wird die LR-Zerlegung auf dem Block im lokalen Speicher durchgeführt. Um dann den Block wieder zurück in den globalen Speicher zu schreiben. Durch den *Offset*-Wert kann der Diagonalblock innerhalb der Matrix verschoben werden.

COMPUTE_PERIMETER

Um die Umkreisblöcke der Eingabematrix mithilfe des Diagonalblockes zu berechnen, wurde diese Map-Skelettfunktion implementiert. Dazu wird ein Vektor mit Indizes, die globale Matrix, lokaler Speicher für einen Diagonalblock und zwei Umkreisblöcke (ein Block aus dem linken oberen einer aus dem rechten unteren Bereich) übergeben. Da für die Lösung eines Umkreisblockes, jeweils die Diagonalmatrix benötigt wird, werden pro Work-group (mit 2 * `BLOCK_SIZE` Work-items) zwei Blöcke verarbeitet. Zunächst werden der Diagonalblock und zwei Umkreisblöcke in den lokalen Speicher kopiert. Anschließend wird die Berechnung der Blöcke im lokalen Speicher durchgeführt und die Umkreisblöcke werden in den globalen Speicher zurück kopiert.

UPDATE_INTERIOR

Die Aktualisierung der Blöcke innerhalb der Eingabematrix mithilfe der Umkreisblöcke kann mit dieser Map-Skelettfunktion durchgeführt werden. In dieser Skelettfunktion wird pro Work-group ein Block verarbeitet (mit `BLOCK_SIZE` * `BLOCK_SIZE` Work-items). In Analogie zu den vorherigen Skelettfunktionen werden die Blöcke in den lokalen Speicher geladen, verarbeitet und wieder zurück kopiert.

Durch die Benutzung des lokalen Speichers soll die Ausführung der Skelettfunktionen beschleunigt werden.

In Listing 4.1 kann eine vereinfachte Darstellung der Definition des Skelettprogramms verfolgt werden (in Zeile 3, 7 und 11). Ebenfalls dargestellt ist das Setzen der Work-group Größe, durch die Funktion `setWorkGroupSize` auf dem Skelett (in Zeile 4, 8 und 12).

```
1  // 3. Define the skeleton program.
2  // Decomposes a single block of the input matrix.
3  Map<void(Index)> DECOMPOSE_DIAGONAL(/* [...] */);
4  DECOMPOSE_DIAGONAL.setWorkGroupSize(BLOCK_SIZE);
5  // Computes all perimeter blocks.
6  Map<void(Index)> COMPUTE_PERIMETER(/* [...] */);
7  COMPUTE_PERIMETER.setWorkGroupSize(2 * BLOCK_SIZE);
8  // Updates all interior blocks.
9  Map<void(IndexPoint)> UPDATE_INTERIOR(/* [...] */);
10 UPDATE_INTERIOR.setWorkGroupSize(BLOCK_SIZE * BLOCK_SIZE);
```

Listing 4.1: Definieren des Skelettprogramms

Listing 4.2 zeigt, wie lokaler Speicher als zusätzlicher Parameter definiert wird. Durch `Local(size)` wird in der Skelettausführung der entsprechende lokale Speicher mit `size` Bytes zur Verfügung gestellt (in Zeile 6, 13 bis 15, 19 und 21). Dargestellt ist außerdem wie durch die `IndexMatrix` oder den `IndexVector`, das Map-Skelett auf eine Menge von Indizes angewendet werden kann (Zeile 4, 11, 17).

```
// 4. Decompose input matrix.
const size_t dim = matrix.columnCount();
for(unsigned int i = 0; i < dim; i += BLOCK_SIZE) {
  DECOMPOSE_DIAGONAL( IndexVector(BLOCK_SIZE),
      out(matrix),
      Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
      i );
  if(i + BLOCK_SIZE < dim) {
    size_t index = BLOCK_SIZE * ((dim - i) / BLOCK_SIZE - 1);
    COMPUTE_PERIMETER( IndexVector(2 * index),
        matrix,
        Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
        Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
        Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
        i );
    UPDATE_INTERIOR( IndexMatrix({index, index}),
        matrix,
        Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
        Local(sizeof(float) * BLOCK_SIZE * BLOCK_SIZE),
        i );
  }
}
```

Listing 4.2: Ausführung des Skelettprogramms

4.3 Evaluierung

In diesem Abschnitt wird die SkelCL-Anwendung der blockweisen LR-Zerlegung aus Abschnitt 4.2 mit einer OpenCL- und CUDA-Anwendung aus der Rodinia Benchmark Suite [5] verglichen.

Im Gegensatz zur OpenCL- und CUDA-Anwendung konnte in der SkelCL-Anwendung auf die manuelle Speicherverwaltung verzichtet werden. Dies liegt an den zur Verfügung gestellten SkelCL-Datentypen (Vektor und Matrix). Dadurch kann SkelCL auch den Datentransfer mit dem Device übernehmen. Des Weiteren wahrt SkelCL die Portabilität der Anwendung.

4.3.1 Messungen

Um die Performance der SkelCL-Anwendung mit einer OpenCL- und einer CUDA-Anwendung zu vergleichen, wurden Messungen durchgeführt.

Die drei Anwendungen zerlegten eine 8192×8192-Matrix, gefüllt mit Zufallszahlen (single precision floating point numbers). In der OpenCL- und CUDA-Anwendung wurde die Matrix über eine Textdatei eingelesen. Die SkelCL-An-

wendung benötigte keine Textdatei, sie erstellte die Matrix und füllte diese mit Zufallszahlen (vgl. Abschnitt 4.2). Die Matrix wurde dann auf das Device kopiert, gelöst mit der blockweisen LR-Zerlegung und zurück auf den Host kopiert.

Gemessen wurde die Ausführungszeit der blockweisen LR-Zerlegung, einschließlich der Übertragungszeiten für die Matrix (Matrix auf das Device kopieren und zurück zum Host kopieren), über insgesammt 100 Iterationen. Für die durchgeführten Iterationen wurde der Durchschnitt für die gemessene Ausführungszeit ermittelt.

Es wurden zwei Devices für die Messungen verwendet. Das erste Device ist eine Nvidia GPU (GeForce GTX 480) mit 1,5 GB Hauptspeicher. Das System der GPU enthielt eine Intel Core 2 CPU mit 2,4 GHz, der 2 GB Hauptspeicher zur Verfügung standen. Das zweite verwendete Device ist eine Intel CPU (Core i7 860) mit 2,8 GHz, der 6 GB Hauptspeicher zur Verfügung standen.

In Tabelle 4.1 sind die Ausführungszeiten der LR-Zerlegung und die gesammt Ausführungszeit der Anwendungen für die GPU und die CPU angegeben. Bei der Ausführung der LR-Zerlegung auf der CPU war die OpenCL-Anwendung nur um 1 % schneller als die SkelCL-Anwendung.

Die SkelCL-, OpenCL- und CUDA-Anwendung auf der GPU unterschieden sich bei der Ausführungszeit für die LR-Zerlegung nur geringfügig. Die OpenCL-Anwendung war nur um etwa 1 % schneller als die SkelCL-Anwendung. Die CUDA-Anwendung war in der durchgeführten Messung um etwa 3 % schneller als die OpenCL-Lösung.

Device	Anwendung	LR-Zerlegung	Total
Intel Core i7 860	SkelCL	112,185	124,194
	OpenCL	111,073	127,947
GeForce GTX 480	SkelCL	2,495	22,836
	OpenCL	2,464	27,108
	CUDA	2,375	27,663

Tabelle 4.1: Die Ausführungszeiten der SkelCL-, OpenCL- und CUDA-Anwendung für die durchgeführte LR-Zerlegung auf dem Device und die gesammte Anwendung in Sekunden (Durchschnitt von 100 Iterationen).

Aufgrund der unterschiedlichen Initialisierung der Eingabematrix lieferte die SkelCL-Anwendung in der gesammt Ausführungszeit die beste Performance (vgl.

Tabelle 4.1, letzte Spalte). Das verwendete Verfahren der OpenCL- und CUDA-Anwendung zum Lesen der Matrix aus einer Textdatei ist langsamer, als das Generieren durch einen Zufallszahlgenerator.

4.4 Schnittstellenbeschreibung

In diesem Abschnitt wird die Kommandozeilenschnittstelle der blockweisen LR-Zerlegung vorgestellt. Der Aufruf der Schnittstelle ist aus dem Programmverzeichnis wie folgt möglich:

```
1 ./blud [Optionen]
```

In Tabelle 4.2 findet sich eine Beschreibung der möglichen Optionen.

Tabelle 4.2: Kommandozeilen Optionen

Kurzform	Langform	Beschreibung
-h	--help	Zeigt eine Hilfe an und beendet das Programm.
-o	--outputFile arg	Gibt den Pfad zur Ausgabedatei an. Die berechnete Matrix wird in diese Datei geschrieben.
-n	--dimension arg	Gibt die Dimension der Matrix an. Wird kein ein Vielfaches von 16 angegeben, wird auf das nächste Vielfache von 16 skaliert.
-p	--platformId arg	Auswahl der Plattform Id von 0 bis $N-1$, wobei N die Anzahl der verfügbaren OpenCL-Plattformen ist.
-d	--deviceId arg	Auswahl des Devices Id von 0 bis $N-1$, wobei N die Anzahl der verfügbaren OpenCL-Devices ist.

5 SIMD-oriented Fast Mersenne-Twister

Für die Generierung von Zufallszahlen werden Algorithmen gebraucht, die Zahlenketten erzeugen, die für den Betrachter zufällig erscheinen. Ein derartiger Algorithmus wird als Pseudozufallszahlengenerator bezeichnet. Eine bestimmte Gruppe sind die Linear Feedback Shift Register (LFSR) Generatoren.

Ein LFSR-Generator erzeugt eine Zahlenfolge $X_0, X_1, X_2, \cdots \in \mathbb{F}_2^\omega$ durch die folgende Rekursion

$$X_{i+N} := g(X_i, X_{i+1}, \cdots, X_{i+N-1}),$$

dabei ist $X_i \in \mathbb{F}_2^\omega$ und $g : (\mathbb{F}_2^\omega)^N \to \mathbb{F}_2^\omega$ eine lineare Funktion und diese Zahlenfolge kann als pseudozufalls ω-bit Integer Folge verwendet werden[6]. Der Körper $\mathbb{F}_2$ ist die Menge $\{0, 1\}$ mit den Operationen + (gegeben durch bitweise XOR) und $*$ (gegeben durch bitweise UND).

Der SIMD-oriented (Single Instruction, Multiple Data) Fast Mersenne-Twister (SFMT) ist ein LFSR-Generator und eine Variante des Zufallszahlgenerators Mersenne-Twister. Im Gegensatz zum Mersenne-Twister wurde der SFMT entwickelt, um technische Neuerungen wie SIMD-Operationen, schnelle Fließkommazahl Operationen und multi-stage pipelining von modernen CPUs auszunutzen [6].

In diesem Kapitel wird ein SFMT-Zufallszahlgenerator mit der SkelCL-Bibliothek implementiert. Zunächst wird der Algorithmus des SFMT vorgestellt. Im Anschluss folgt die Darstellung der SkelCL-Anwendung. Zuletzt wird die SkelCL-Anwendung mit einer OpenCL-Anwendung verglichen.

5.1 Algorithmus

Der SFMT ist ein LFSR mit der Funktion

$$g(w_0, \cdots, w_N - 1) = w_0 A + w_M B + w_{N-2} C + w_{N-1} D,$$

dabei sind $w_0, w_M, \cdots$ 128-Bit-Integer Werte ($\omega = 128$), $1 < M < N$ und A, B, C und D schwachbesetzte 128x128 Matrizen über $\mathbb{F}_2^\omega$ [6]. Der Algorithmus wurde

ausführlich von Mutsuo Saito [7] beschrieben, die linearen Transformationen A, B, C und D wurden wie folgt beschrieben:

- $wA := (w \overset{128}{<<} 8) + w$.
  Diese Transformation verschiebt w als 128-Bit-Integer um 8 Bits nach links. Das Ergebnis wird anschließend XOR-Verknüpft mit w. Diese Transformation wird in Abbildung 5.1 auf w_0 angewandt.

- $wB := (w \overset{32}{>>} 11)$ & MASK.
  In der zweiten Transformation wird w als 32-Bit Quadruple aufgefasst. Jede einzelne Komponente wird zunächst um 11 Bits nach rechts verschoben. Anschließend wird mit einer vorgegeben 128-Bit Konstante MASK bitweise UND-Verknüpft. Diese lineare Transformation wird in Abbildung 5.1 auf w_{122} angewandt.

- $wC := (w \overset{128}{>>} 8)$.
  In dieser Transformation wird w wieder als 128-Bit Zahl aufgefasst und um 8 Bits nach recht verschoben. Angewandt auf w_{154} in Abbildung 5.1.

- $wD := (w \overset{32}{<<} 18)$.
  Die letzte Transformation besteht daraus w als 32-Bit Quadruple aufzufassen, und jede Komponente um 18 Bits nach links zu verschieben. In Abbildung 5.1 angewandt auf w_{155}.

Der SFMT-Algorithmus benötigt eine Menge von N Startzuständen. Genauer zu sehen ist dieses Verfahren in einer weiteren grafischen Beschreibung des SFMT-Algorithmus in Abbildung 5.1 für $N = 156$.

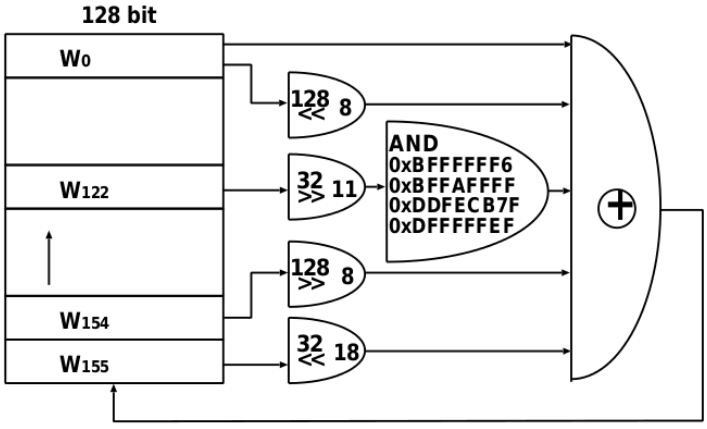


Abbildung 5.1: Beschreibung des SFMT ($N = 156$)[6].

5.2 Implementierung

Die SkelCL-Implementierung dargestellt in Listing 5.2 initialisiert zunächst OpenCL durch die SkelCL-Funktion `skelcl::init` (in Zeile 2). Anschließend wird ein SkelCL-Vektor erstellt, der mit Startwerten gefüllt wird (Zeile 5).

Es folgt im Anschluss die Definierung des Skelettprogramms (Zeile 9). Die Implementierung bildet durch ein Work-item genau einen SFMT-Generator (aus Abschnitt 5.1) ab. Jedem SFMT-Generator wird genau ein Startwert übergeben. Dafür wird ein Map-Skelett verwendet, das auf alle Elemente (Startwerte des SFMT) des SkelCL-Vektors die Map-Skelettfunktion (SFMT-Generator) anwendet.

Für die Map-Skelettfunktion `TWISTER` liegen zwei Implementierungen vor, die beide genutzt werden können. Die erste Skelettfunktion hat den SFMT-Generator aus Abschnitt 5.1 implementiert. In der zweiten Skelettfunktion wurde der OpenCL-Kernel der AMD-Implementierung aus dem APP SDK übernommen. Der OpenCL-Kernel konnte ohne wesentliche Veränderungen (Änderungen betreffen die Schnittstelle) übernommen werden.

Damit liegen für eine Skelettfunktion keine signifikaten Änderungen im OpenCL-C Programmcode vor. Die SkelCL-Implementierung unterscheidet sich jedoch weiterhin von der OpenCL-Implementierung bei Wahl der Work-groups. Während die OpenCL-Anwendung $N \times N \times 1$ Work-group Größen wählt, werden in der SkelCL-Implementierung durch die Verwendung des Map-Skelettes $(N * N) \times 1 \times 1$ Work-group Größen gewählt. Die Anzahl der Work-items pro Work-group bleibt gleich.

Die beiden Map-Skelett-Implementierungen können unterschiedlich viele Zufallszahlen pro Generator erstellen. In der ersten Implementierung (SFMT nach Abschnitt 5.1) können maximal 156 Zahlen pro Generator produziert werden. In der AMD-Implementierung können maximal 8 Zahlen pro SFMT-Generator produziert werden, dafür wurden Optimierungen vorgenommen – wie der Verzicht auf Funktionsaufrufe und Array-Indexrechnungen.

Für beide Implementierungen wurden die Schritte der Initialisierung und einer anschließenden Transformation analog zur AMD-Implementierung gewählt. Der SFMT-Algorithmus aus Abschnitt 5.1 benötigte eine Menge von N Startzuständen. Um auf dem Host die Menge an zu erstellenden Startzuständen zu verringern, wurde jedem Work-item nur ein einzelner Startzustand (Seed) übergeben. Die restlichen $N-1$ Zustände werden durch die Methode dargestellt in Listing 5.1 erstellt (für Zustand 2 in Zeile 3).

```
1 uint4 state1 = seed;
2
3 uint4 state2 = MASK * (state1 ^ (state1 >> SHIFT)) + 1;
```

Listing 5.1: Erzeugung weiterer Zustände aus dem Startzustand `state1`.

Jeder SFMT-Generator erwartet 128-Bit-Integer, diese werden dargestellt durch jeweils vier 32-Bit-Integer. Die nach der Zufallszahlgenerierung in vier 32-Bit-Fließkommazahlen transformiert werden. Dazu wird im Anschluss die Box-Muller Transformation auf die Zufallszahlen angewandt [9]. Dabei handelt es sich um einen Algorithmus, der aus gleichverteilten Zufallszahlen standardnormalverteilte Zufallszahlen erzeugt.

Die Skelettfunktion führt das Verfahren aus (in Zeile 15) und liefert anschließend eine Rückgabe in Form des letzten generierten Zustandes. Der letzte Zustand könnte in einer weiteren Iteration als erneuter Seed verwendet werden. Alle generierten Zahlen werden in die Matrix `numbers` (in Zeile 12) geschrieben, diese wurde dem Skelett als zusätzlicher Parameter übergeben.

Ebenfalls Teil der Implementierung ist eine Verifikation. Diese Verifikation wurde in der AMD-Implementierung verwendet, um die Qualität der Zufallszahlen zu überprüfen. Das Verifikationsverfahren bildet die Summe von allen generierten Zahlen. Wenn eine große Anzahl Zahlen produziert wurde, sollte die Summe im Idealfall 0 sein. Die SkelCL-Implementierung kann die Abweichung von 0 als Fehlerwert auf der Kommandozeile anzeigen.

```
1  // 1. Initialize SkelCL.
2  skelcl::init(skelcl::nDevices(1));
3
4  // 2. Input seed vector.
5  Vector<skelcl::uint4> seeds(seedCount);
6  fillSeedVector(seeds.begin(), seeds.end());
7
8  // 3. Define the SFMT skeleton program.
9  Map<skelcl::uint4(skelcl::uint4)> TWISTER(/* [...] */);
10
11 // Output matrix, contains numbers generated.
12 Matrix<float> numbers({seedCount, 4 * multiplier});
13
14 // 4. Perform number generation.
15 seeds = TWISTER( seeds, skelcl::out(numbers), multiplier );
16
17 // 5. Retrieve the result matrix.
18 printMatrix(numbers);
```

Listing 5.2: Vereinfachte SFMT-Implementierung in SkelCL

5.3 Evaluierung

In der SkelCL-Anwendung aus Abschnitt 5.2 konnten, durch die Verwendung des OpenCL-Kernels der OpenCL-Anwendung aus dem APP SDK, die Unterschiede in der Implementierung gering gehalten werden. Unterschiede liegen in der Art der Parameterübergabe, einer zusätzlichen Rückgabe und dem Setzen des ersten Zustandes vor.

Es wurde außerdem eine neue SFMT-Implementierung erstellt. Die Erstellung der SFMT-Implementierung war ohne großen Aufwand möglich. Der Programmcode der entstandenen Anwendung ist dabei weniger komplex als der OpenCL-Anwendung. Dies geht daraus hervor, dass SkelCL die Speicherverwaltung und das Kopieren der Daten übernahm. Des Weiteren wurde die OpenCL-Initialisierung durch die SkelCL-Funktion `skelcl::init` übernommen. Dies verringerte den Aufwand für die Initialisierung von OpenCL und die Anwendung wurde vereinfacht.

5.3.1 Messungen

Für den Vergleich, der SkelCL-Anwendung des SFMT aus Abschnitt 5.2 mit einer OpenCL-Anwendung eines SFMT aus dem APP SDK, wurden Messungen durchgeführt.

Beide Anwendungen erstellten dazu 1536^2 Zufallszahlgeneratoren, die jeweils 8 128-Bit-Integer produzierten. Diese generierten 128-Bit-Integer wurden in 4 32-Bit-Fließkommazahlen mit einfacher Genauigkeit (single precision floating point number) umgewandelt. Anschließend wurden diese mit der Box-Muller Transformation in normalverteilte Zufallszahlen transformiert. Insgesammt wurden damit etwa $7,5 * 10^7$ (genauer: $1536^2 * 8 * 4 = 75497472$) 32-Bit-Zufallszahlen erstellt.

Gemessen wurde dann die durchschnittliche Ausführungszeit (über 400 Iterationen), um die Zufallszahlen zu generieren und vom Device auf den Host zu kopieren. Die SkelCL-Anwendung verwendete den angepassten OpenCL-Kernel der OpenCL-Anwendung (vgl. Abschnitt 5.2).

Die SkelCL- und OpenCL-Anwendungen profitieren von der Unterstützung von SIMD-Operations (hier die Möglichkeit die 4 32-Bit Integer der 128-Bit Integer Zahl gleichzeitig zu verarbeiten). Damit die Messungen nicht von der Unterstüzung von SIMD-Operations von einer CPU abhängen, wurden 4 unterschiedliche CPUs verwendet. Im Folgenden sind die verwendeten CPUs angeben (in Klammern die Taktrequenz und der verfügbare Hauptspeicher der CPU):

- Intel Core i7 860 (2,8 GHz, 6 GB)
- AMD A8-3850 (2,9 GHz, 16 GB)

- Intel Core i3-2330M (2, 2 GHz, 4 GB)
- Intel XEON E5520 (2, 27 GHz, 12 GB)

Für die Intel XEON wurden die Anwendung mit der Intel OpenCL-Plattform, als auch mit der AMD OpenCL-Plattform ausgeführt. Auf der Intel Core i7 und der AMD A8-3850 CPU wurde nur die AMD-Plattform eingesetzt. Die Intel Core i3 wurde mit der Intel OpenCL-Plattform eingesetzt. Zusätzlich zu den verwendeten CPUs wurde eine Geforce GTX 480 GPU (1,5 GB Hauptspeicher) eingesetzt.

Tabelle 5.1, zeigt die Ausführungszeit des SFMT für die Core i7, Core i3 und AMD A8-3850 (die ausgewählte OpenCL-Plattform steht in Klammern). Die Unterschiede zwischen der OpenCL-Anwendung und der SkelCL-Anwendung wirken sich unterschiedlich auf die Devices aus. Beobachtet wurde, dass die OpenCL-Anwendung auf der Core i7 um 16 % und auf der AMD A8 um 11 % schneller war. Für die Core i3 konnte kein Unterschied gemessen werden.

Device	Anwendung	SFMT
Core i7 (AMD)	SkelCL	0, 325
	OpenCL	0, 278
A8-3850 (AMD)	SkelCL	0, 592
	OpenCL	0, 527
Core i3 (Intel)	SkelCL	0, 790
	OpenCL	0, 793

Tabelle 5.1: Die Ausführungszeiten der SkelCL- und der OpenCL-Anwendung für den SFMT (Durchschnitt von 400 Iterationen in Sekunden). Die gewählte OpenCL-Plattform steht in Klammern.

In Tabelle 5.2 sind die AMD und die Intel OpenCL-Plattform für die Intel XEON CPU verwendet worden. Die AMD OpenCL-Plattform liefert für beide Anwendungen die bessere Performance. Die OpenCL-Anwendung ist hier lediglich 2 % schneller. Für die Intel-Plattform ist die OpenCL-Anwendung mit 16 % kürzerer Ausführungszeit ermittelt worden.

Für die CPU lag der Performance Verlust somit bei maximal 16 %. Für zwei der Devices lag der Unterschied sogar bei unter 2 %.

Device	Anwendung	SFMT
XEON (Intel)	SkelCL	0,577
	OpenCL	0,494
XEON (AMD)	SkelCL	0,413
	OpenCL	0,402

Tabelle 5.2: Die Ausführungszeiten des SFMT in SkelCL und OpenCL für die Intel XEON CPU (Durchschnitt von 400 Iterationen in Sekunden). Die OpenCL-Plattform steht in Klammern.

In den Messungen für die GPU wurde die Anzahl der Zufallszahlen die pro Zufallszahlgenerator erstellt wurden auf 2 verringert. Da das maximale Speicherobjekt hier nur 128 MB groß sein durfte. In Tabelle 5.3 sind die Ausführungszeiten für die GPU angegeben. Die OpenCL-Anwendung war um den Faktor 3 schneller. Aufgrund der geringen Ausführungszeit zwischen 50 und 150 ms, könnten jedoch auch die geringen Unterschiede zwischen den Anwendung diesen deutlichen Unterschied ausgelöst haben.

Device	Anwendung	SFMT
GeForce GTX 480	SkelCL	0,151
	OpenCL	0,052

Tabelle 5.3: Die Ausführungszeiten des SFMT in SkelCL und OpenCL für die GPU (Durchschnitt von 400 Iterationen in Sekunden).

5.4 Schnittstellenbeschreibung

In diesem Abschnitt soll die Kommandozeilenschnittstelle der SFMT SkelCL-Anwendung beschrieben werden. Gestartet werden kann die Schnittstelle im Programmverzeichnis über den folgenden Aufruf:

```
1 ./mersenne_twister [Optionen]
```

In Tabelle 5.4 findet sich eine Beschreibung der möglichen Optionen.

Tabelle 5.4: Kommandozeilen Optionen

Kurzform	Langform	Beschreibung
-h	--help	Zeigt eine Hilfe an und beendet das Programm.
-q	--quiet	Stellt die Textausgabe ab.
-s	--display	Anzeige der 32-Bit-Zufallszahlen.
-v	--verify	Startet die Durchführung einer einfachen Verifikation.
-o	--outputFile arg	Gibt den Pfad zur Ausgabedatei an. Sämtliche generierte Zufallszahlen werden in diese Datei geschrieben.
-n	--generators arg	Wählt die Anzahl der Zufallszahlgeneratoren.
-m	--rands arg	Gibt die Anzahl der zu generierenden 128-Bit-Zufallszahlen pro Generator an.
-i	--iterations arg	Anzahl der Iterationen, die durchgeführt werden sollen. Führt die Zufallszahlzahlgeneration mehrfach nacheinander aus. Die erzeugten Zahlen werden verworfen.
-k	--impl arg	Wählt die Kernel Implementierung. Durch das Argument `amd` kann die APP SDK Implementierung verwendet werden. Durch die Angabe von `new` wird die neue SFMT-Implementierung verwendet. Standard ist die APP SDK SFMT-Implementierung.
-p	--platformId arg	Auswahl der Plattform Id von 0 bis $N - 1$, wobei N die Anzahl der verfügbaren OpenCL-Plattformen ist.
-d	--deviceId arg	Auswahl des Devices Id von 0 bis $N - 1$, wobei N die Anzahl der verfügbaren OpenCL-Devices ist.

6 Fazit

In dieser Bachelorarbeit wurden mithilfe der SkelCL-Bibliothek drei Anwendungen implementiert. Die erstellten Anwendungen sind die LR-Zerlegung – in einer standardweisen und einer blockweisen Variante – und der Mersenne-Twister Zufallszahlgenerator in der Variante des SFMT.

Die erstellten SkelCL-Anwendungen sind ähnlich flexibel und portabel wie vergleichbare OpenCL- oder CUDA-Anwendungen. Der entstandene Host-Code ist dabei weniger komplex und weniger umfangreich. Alle gewünschten OpenCL-C-Funktionen konnten in der SkelCL-Anwendung benutzt werden – wie Vektordatentypen und Fließkommazahl-Funktionen. Außerdem konnte durch SkelCL die aufwändige OpenCL-Initialisierung vereinfacht werden.

SkelCL stellte vielseitig einsetzbare Skelette zur Verfügung, die es ermöglichten einen Algorithmus abstrakt zu beschreiben. Auf eine hohe Granularität der Skelette bei der Beschreibung des Algorithmus wurde aber verzichtet. In den Anwendungen wurden die Skelette ähnlich zu OpenCL-Kernel-Funktionen eingesetzt.

Ferner führte SkelCL zwei neue Datentypen Vektor und Matrix ein. Diese ließen sich sinnvoll in die Anwendung integrieren und machten es sogar möglich auf die manuelle Speicherverwaltung zu verzichten. Denn SkelCL übernahm das Kopieren der Daten des Vektors oder der Matrix zwischen dem OpenCL-Host und dem OpenCL-Device. Dadurch entfielen fehleranfällige und komplexe Routinen für den Transport der Daten.

Die SkelCL-Datentypen und die Skelette standen in SkelCL über eine leicht verständliche C++-Schnittstelle bereit.

Die Entscheidung für die Art der Verarbeitung in der standardweisen LR-Zerlegung hatte negativen Einfluss auf die Performance. Gemessen wurde daher ein deutlicher Unterschied zur OpenCL-Anwendung. Für die blockweise LR-Zerlegung zeigten sich in den Messungen nur minimale Performance Verschlechterungen gegenüber der OpenCL- oder der CUDA-Anwendungen.

Bei der SFMT SkelCL-Anwendung wurden zum Teil verschlechterte Ausführungszeiten gegenüber der OpenCL-Anwendung gemessen. Aufgrund der geringen Ausführungszeiten der Anwendung könnten die Unterschiede in den Implementierungen dafür verantwortlich sein. Änderungen in der SFMT-Implementie-

rung und in der Implementierung der standardweisen LR-Zerlegung könnten die Performance verbessern.

Die Erfahrungen mit der SkelCL-Bibliothek, die in dieser Bachelorarbeit gesammelt wurden, können in der Folge helfen bessere und performantere SkelCL-Anwendung zu erstellen.

Quellen

[1] Khronos OpenCL Working Group: *The OpenCL Specification, Version: 1.2* http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, 14.11.2012.

[2] Horacio Gonzalez-Velez, Mario Leyton (2010): *A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers*, Software – Practice & Expirience, Volume 40 Issue 12, Pages 1135 – 1160.

[3] Michel Steuwer, Philipp Kegel, Sergei Gorlatch (to appear): *Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library*, 2012 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW), Shanghai, China.

[4] AMD APP SDK: *Accelerated Parallel Processing (APP) SDK* http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/, 19.02.2013.

[5] University of Virginia: *Rodinia:Accelerating Compute-Intensive Applications with Accelerators* https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page, 28.01.2013.

[6] Mutsuo Saito and Makoto Matsumoto (2008): *SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*, Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, pp. 607 – 622, URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/sfmt.pdf, 28.02.2013.

[7] Mutsuo Saito (2007): *An Application of Finite Field: Design and Implementation of 128-bit Instruction-Based Fast Pseudorandom Number Generator*, URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/M062821.pdf, 28.02.2013.

[8] Sergei Gorlatch: *Folien zur Vorlesung Multi-core und GPU: Parallele Programmierung* http://pvs.uni-muenster.de/pvs/lehre/SS11/mgpp/vorlesung.html, 19.02.2013.

[9] MathWorld: *Box-Muller Transformation*
http://mathworld.wolfram.com/Box-MullerTransformation.html,
26.02.2013

[10] Mario Ohlberger: *Einführung in die Numerische Mathematik*
http://wwwmath.uni-muenster.de/num/Vorlesungen/Numerik_WS07/
Skript/skriptum.pdf, 26.02.2013

Versicherung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Ort, Datum Sebastian Mißbach