FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Algorithmen für dünn besetzte Matrizen auf der CBEA

Daniel Brinkers

Studienarbeit

Algorithmen für dünn besetzte Matrizen auf der CBEA

Daniel Brinkers

Studienarbeit

Aufgabensteller:	Prof. Dr. U. Rüde
Betreuer:	Dipl.-Inf. M. Stürmer
Bearbeitungszeitraum:	25.02.2008 – 26.11.2008

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 26. November 2008 .....

Inhaltsverzeichnis

1 Die Cell Broadband Engine

Der Cell Prozessor wurde von IBM zusammen mit Sony und Toshiba entwickelt. Das Entwicklungsziel war ein Prozessor mit hohen Rechendurchsatz und hoher Energie Effizienz. Das Ergebnis ist eine CPU mit neun Kernen. Ein PPE (PowerPC Processing Element) und acht SPEs (Synergistic Processing Elements). Das PPE besteht aus einen PowerPC Kern. Die Hauptaufgabe des PPE ist die Steuerung der SPEs und die Interaktion mit den Betriebssystem. Die SPEs hingegen übernehmen das Rechnen. Die Architektur dieser SPEs ist einfacher als die der meisten modernen Prozessoren. Die Befehle werden in zwei Piplines abgearbeitet, die nur "in order" Ausführung unterstützen. Ausserdem fehlen Caches, dafür besitzen sie je 256 kB lokalen Speicher. In diesen müssen Programme und Daten Platz finden. Der Zugriff auf diesen Speicher erfolgt mit vorhersagbarer Latenzen. Die Registerdatei der SPEs ist mit jeweils 128 128Bit Registern sehr gross. Der Befehlssatz der SPEs ist auf SIMD Instruktionen ausgelegt und ist vergleichbar mit den SSE Befehlssätzen der x86 Prozessoren. Die Kerne untereinander und der Speicher Controller sind über einen Bus verbunden. Über diesen können sich die SPEs Daten aus den Hauptspeicher in ihren lokalen Speicher kopieren. Das Verhältnis von Rechendurchsatz zu Speicherdurchsatz ist beim Cell besonders gross. Die theoretische Rechenleistung liegt bei 204,8 Gflop/s. Der Datendurchsatz zum Hauptspeicher bei 25,6 GB/s. Durch die architektonischen Unterschiede hat man bei der Programmierung andere Möglichkeiten und andere Probleme als bei herkömmlichen Prozessoren.

Um die Performance der Cell Prozessoren voll nutzen zu können, muss man die SIMD Befehle der SPEs nutzen. Bei der ersten Generation der Prozessoren verliert man ausserdem unverhältnismässig viel Leistung, wenn man mit Fließkommazahlen mit doppelter Genauigkeit rechnet. Deshalb wird in dieser Arbeit nur mit einfacher Genauigkeit gerechnet. Bei einfacher Genauigkeit arbeiten sie SIMD Befehle auf vier Elementen. Um so einen Vektor besser von den Daten unterscheiden zu koennen, wird er im folgenden SIMD-Vektor genannt.

Um Daten in den lokalen Speicher der SPEs und zurück zu kopieren verwendet man DMA Operationen. Diese können asynchron zu den restlichen Programm ablaufen. Damit die DMA Übertragungen optimal funktionieren, müssen die Quell- und Zieladressen auf 128 Byte ausgerichtet sein [CEL].

Der Cell Prozessor wurde auch in der Playstation 3 von Sony verbaut. Das ist eine Möglichkeit einen Cell Prozessor relativ günstig zu erwerben. Da Sony eine Portierung von Linux möglich gemacht hat, kann man auch eigene Software mit frei erhältlichen Programmen entwickeln. Allerdings ist bei der Playstation eine der SPEs abgeschaltet und eine weitere wird intern verwendet. Man kann also nur sechs der SPEs frei verwenden [PS3]. Auch die Entwicklung für diese Arbeit fand auf einer PS3 statt.

2 Matrix Vektor Multiplikation

Die Matrix Vektor Multiplikation ist eine Operation, die sehr oft in der linearen Algebra verwendet wird. Viele iterative Löser von linearen Gleichungssystem kann man mit einer SPMV (sparse matrix vector multiplication) ausführen. Dünn besetzte Matrizen sind Matrizen, die $O(n)$, oder $O(\log n)$ viele Nicht-Null-Einträge haben. Das heisst die meisten Einträge sind Null.

2.1 Bisherige Formate

Der entscheidend Unterschied zwischen verschiedenen Matrix Vektor Multiplikation Implementierungen ist das Speicherformat der Matrix. Die verschiedenen Formate haben unterschiedliche Vor- und Nachteile und damit Einsatzgebiete. Zunächst sollen oft verwendete Formate betrachtet werden.

2.1.1 COO

Die einfachste Möglichkeit eine dünn besetzte Matrix zu speichern ist das COO (coordinate) Format. Die Matrix wird in drei Arrays gespeichert. Elemente mit gleichen Index gehören zusammen. Die Grösse ist jeweils die Anzahl der Nicht-Null-Elemente. In einem Array stehen die Werte der Einträge. Die Spalten- und Zeilenindizes belegen jeweils ein weiteres Array. Der Vorteil dieses For-

mates ist, dass es sehr einfach zu handhaben ist. Der grösste Nachteil ist, dass sehr viel Speicherplatz verschwendet wird.

Ausserdem ist das Format sehr schlecht für die Berechnung auf den SPEs geeignet. Werte, die im Array hintereinander stehen und deshalb auf der SPE in einen SIMD-Vektor geladen werden, können sehr verstreute Spaltenindizes haben. Die für die Multiplikation benötigten Werte aus den Eingabe- und den Ergebnisevektor liegen also verstreut und selten in einem SIMD-Vektor. Deshalb kann man die SIMD Operationen nicht direkt ausführen, sondern muss erst die Daten richtig ausrichten. Man kann entweder die Werte der Matrix aufteilen und die Multiplikationen einzeln ausführen, oder man muss sich die benötigten Werte aus den Vektoren zusammensuchen. Beides ist zu aufwendig und nutzt die Vorteile von SIMD-Operationen nicht aus.

2.1.2 CRS

Ein sehr verbreitetes Speicherformat ist das CRS (compressed row storage) Format. Auch hier Verwendet man drei Arrays zum Speichern der Matrix. Die Arrays für die Werte und die Spaltenindizes sind genauso belegt wie bei den COO Format. Allerdings müssen die Werte in diesen Arrays nach Zeile sortiert sein. Das dritte Array ist um eins grösser als die Matrix hoch ist. In diesen Array wird für jede Zeile der Index, den der erste Werte dieser Zeile in den anderen beiden Arrays hat, gespeichert. In den letzten Wert des Arrays speichert man den ersten ungültigen Index. Abbildung 1 zeigt ein Beispiel für eine im CRS Format gespeicherte Matrix.

1	2		
	3	4	5
		6	
			7

```
Werte       = {1,2,3,4,5,6,7}
Spalte      = {0,1,1,2,3,2,3}
Zeilenbegin = {0,  2,    5,6,7}
```

Abbildung 1: CRS einer 4x4 Matrix

Durch die Kompression der Zeilenindizes spart man Speicherplatz. Allerdings sind auch hier die Daten schlecht für SIMD Operationen geeignet. Weil man das gleiche Problem beim Ausrichten der zugehörigen Vektoren hat.

2.1.3 BCOO

Eine Abwandlung des COO Formates ist das BCOO (blocked coordinate) Format. Hier wird die Matrix zunächst in kleine Blöcke aufgeteilt zum Beispiel 2x2, 1x4, 4x4. Wenn in einem Block Nicht-Null-Einträge vorhanden sind, wird dieser Block mit expliziten Nullen aufgefüllt. Die Blöcke liegen hintereinander im Werte Array. Die Koordinaten der Werte müssen nur pro Block gespeichert werden. Abbildung 2 zeigt wie eine Matrix im BCOO Format gespeichert wird.

1	2	0	0
0	3	4	5
		6	0
		0	7

```
Werte  = {1,2,0,3, 0,0,4,5, 6,0,0,7}
Spalte = {      0,       1,       1}
Zeilen = {      0,       0,       1}
```

Abbildung 2: BCOO einer 4x4 Matrix

Dieses Format ist viel besser geeignet um mit SIMD Operationen verwenden zu werden. Wenn die Blockgrösse ein Vielfaches der SIMD Vektorgrösse ist, kann man die Blöcke in SIMD-Vekoren speichern, so dass alle Elemente genutzt werden. Dadurch sind die Blöcke im Speicher richtig ausgerichtet. Die Daten in einem SIMD-Vektor haben nun untereinander einen Bezug. Man kann den

zweiten Operanten leichter aufbauen, weil die Datenoffsets, zwischen den Elementen fest sind. Dadurch kann man die SIMD Operationen gut nutzen.

2.1.4 BCRS

Bei BCRS (blocked compress row storage) werden Blöcke wie bei BCOO gebildet und gespeichert. Die Informationen über die Zeilenindizes werden aber wie beim CRS Format gespeichert.

Der Speicherverlust durch die explizite Speicherung der Nullen in einem belegten Block wird teilweise durch die geringere Anzahl von Spaltenindizes und Zeilenvektoren verringert. Es gibt sogar Matrizen, die mit diesem Speicherformat weniger Speicher verbrauchen als mit den CRS Format. Da ein 2x2 Block genau in einen SIMD Vektor passt wird dieses Format erfolgreich mit den Cell Prozessor verwendet [BCR]. Bei grösseren Blöcken wird der Nachteil durch die explizit gespeicherten Nullen zu gross.

2.1.5 Weitere Formate

Es gibt noch viele weitere Möglichkeiten, wie man dünn besetzte Matrizen speichern kann. Einige davon sind leichte Modifikationen von Formaten, die hier vorgestellt wurden und haben ähnliche Eigenschaften. Andere Formate funktionieren nur mit Matrizen mit bestimmten Eigenschaften, wie zum Beispiel Symmetrie, dicht besetzten Diagonalen oder Bandmatrizen. Solche Sonderfälle sollen hier nicht beachtet werden. Es gibt noch einige Formate, die eine bessere Kompression als CRS erreichen, aber die Extraktion der Datenstruktur ist da noch komplizierter und die Daten sind ebenfalls nicht günstig für die SIMD Operationen gespeichert. [SPM]

2.2 Probleme der bisherigen Ansätze

Die Cell Broadband Engine scheint ungünstige Eigenschaften für die Berechnung von Matrix Multiplikationen mit dünn besetzten Matrizen zu haben. Um mit den SIMD Operationen arbeiten zu können, muss man Formate wie BCOO oder BCRS einsetzen. Damit kann man Algorithmen schreiben, die die volle Speicherbandbreite ausnutzen. Allerdings sorgt auch das Format für eine höhere Auslastung dieser durch die explizit übertragenen Nullen.

2.3 Komprimiertes BCOO

Matrizen im CRS Format sind gut komprimiert, lassen sich von den SPEs aber nicht schnell genug verarbeiten. Dadurch ist hier die Geschwindigkeit durch den Rechendurchsatz beschränkt. Matrizen im BCRS oder BCOO Format lassen sich hingegen sehr schnell auf den SPEs verarbeiten, allerdings verbrauchen sie mehr Speicher und dadurch mehr Speicherbandbreite. Die Geschwindigkeit ist hier durch den Datendurchsatz beschränkt.

Es soll versucht werden ein Format zu finden, dass beide Vorteile vereint. Die Anforderungen an das Format sind:

- Die Daten sollten sich leicht zu den Vektordaten ausrichten lassen, damit man die SIMD Instruktionen der SPU verwenden kann.
- Es wird angenommen, dass die Matrix nur für die Multiplikation verwendet wird. Die Daten und die Struktur der Matrix müssen also nicht veränderbar sein und eine allgemeinen Zugriffsmöglichkeiten für andere Operationen ist nicht nötig.
- Es wird angenommen, dass die Matrix sehr oft multipliziert wird, wie es zum Beispiel bei iterativen Lösungsverfahren vorkommt. Das heisst die Konvertierung von einem verbreiteten Format in das proprietäre Format muss nicht performat sein, da sie nur einmal für sehr viele Multiplikationen durchgeführt werden muss.
- Die Matrizen sind so gross sie nicht komplett in den Speicher der SPEs passen
- Wichtig ist auch, dass man die Daten leicht auf die verschiedenen SPEs verteilen kann. Man muss die Datenstruktur also leicht partitionieren können und man muss auch wissen, welche Teile der Vektoren zu diesen Matrixpartitionen gehören.

- Das wichtigste Ziel ist die Kompression der Daten.
- Ein Dekomprimieren muss ausserdem schnell möglich sein.

2.3.1 Format

2.3.1.1 Partitionierung

Zunächst wird die Matrix in Partitionen aufgeteilt. Eine Partition besteht aus 1024x1024 Werten. Zu jeder dieser Partitionen werden die Koordinaten innerhalb der grossen Matrix gespeichert und der Inhalt als Teilmatrix. Innerhalb dieser Teilmatrix gibt es 256x256 4x4 Blöcke.

2.3.1.2 Koordinatenspeicherung

Die Daten in den Partitionen werden ähnlich wie im BCOO gespeichert. Zunächst werden 4x4 Blöcke mit Nicht-Null-Einträgen gesucht. Für diese Blöcke werden Zeilen- und Spaltenindizes gespeichert.

2.3.1.3 Bitmap

Anders als beim BCOO werden keine Nullen im Wertearry gespeichert. Die Nicht-Null-Einträge eines Blockes werden hintereinander gespeichert. Es gibt ein weiteres Array, mit einen Eintrag für jeden Block. Ein Eintrag besteht aus einem 16 Bit Bitmap. Jedes Bit steht für einen Wert im Block. Eine Eins bedeutet, dass der Wert ungleich Null ist, eine Null, dass der Wert gleich Null ist.

2.3.1.4 Vertikale Ausrichtung

Die Werte eines Blockes sind unter sich erst nach Spaltenindex und dann nach Zeilenindex gespeichert Die Abbildung 3 zeigt, wie eine Teilmatrix abgespeichert wird.

1	2			3		4	
	5						
		6			7		
			8				
							9

```
Werte  ={   1,  2,5,    6,    8,    3,    7,    4,                                9}
Bitmap ={1000, 1100, 0010, 0001, 1000, 0010, 1000, 0000, 0000, 0000, 0000, 0001}
Spalte ={0,                      1,                      1}
Zeile  ={0,                      0,                      1}
```

Abbildung 3: Komprimiertes BCOO einer 4x4 Matrix

2.3.2 Eigenschaften

Das Speichern ist so möglich. Es muss noch überprüft werden, ob die Anforderungen erfüllt werden.

2.3.2.1 Speicherplatz

Beim Speicherplatzverbrauch kann nicht ein Format für allen Matrizen optimal sein. Es ist sehr schwer Annahmen über die Verteilung von Nicht-Null-Einträge und Nullen, die für alle in der Praxis verwendeten Matrizen zutreffen, zu machen. Deshalb wird hier nur für einige Massnahmen beschrieben, wie sie sich auf die Grösse auswirken.

Da die Partitionen so gross sind und die Werte die pro Partition gespeichert werden müssen, kann man davon ausgehen, dass diese Daten für die Gesamtbilanz kaum ins Gewicht fallen. Es ist sogar so, dass man durch die kleineren Datentypen für die Zeilen und Spaltenindizes Speicherplatz spart.

Verglichen mit einem 4x4 BCOO verliert man durch die Bitmaps zunächst zwei Bytes pro Block, aber für jede Null in diesen Block gewinnt man vier Bytes. Also spart man schon ab einer einzigen Null in einem Block. Der Vergleich mit Verfahren, die keine Blöcke verwenden ist schwieriger. Auch hier verliert man zunächst die zwei Bytes durch das Bitmap für den ersten Wert in einem Block. Allerdings spart man sich hier für jeden weiteren Wert in diesen Block das Speichern weiterer Koordinaten. Man spart also Speicherplatz bei relativ dicht besetzten teilen der Matrix. Solche Stellen gibt es in vielen Matrizen. Zum Beispiel in Bändern oder Diagonalen.

Dass man den Zeilenvektor nicht komprimiert, kann sich nicht nur negativ, sondern auch positiv auf den Speicherverbrauch auswirken. Zum einen können die Indizes, die man bei der komprimierten Speicherung speichern muss, so gross werden, dass man dafür 16 Bit Werte verwenden muss, während bei der direkten Speicherung ein Byte reicht. Ausserdem kann es durch die Partitionierung passieren, dass man leere Zeilen in den Teilmatrizen hat. Bei der komprimierten Zeilen Speicherung muss man auch für diese Zeilen einen Wert speichern.

Die Messungen mit reellen Matrizen haben gezeigt, dass das komprimierte BCOO in viele Fällen tatsächlich sehr sparsam ist. Die Ergebnisse werden in Kapitel 3.2 vorgestellt.

2.3.2.2 Parallelesierbarkeit

Die Teilmatrizen können parallel voneinander bearbeitet werden, wenn man das Schreiben auf den Ergebnisvektor synchronisiert.

2.3.2.3 Dekompression

Um zu beurteilen, ob die Dekompression einfach ist, wird diese genauer betrachtet. Als Eingabe hat man zwei Streams mit den Koordinaten, einen für die Bitmaps und einen mit den Werten. Die Koordinaten und die Bitmaps kann man einfach sequenziell lesen. Die Rekonstruktion der Nicht-Null-Einträge und der Nullen ist komplizierter. Der Zugriff auf die Daten erfolgt mit einem 128 Bit grossen Quadword. Um eine Position eines Elements im Stream zu bestimmen werden zwei Indices verwendet. Einer bestimmt den SIMD-Vektor, in welchen sich das Element befindet, und der andere das Element innerhalb dieses SIMD-Vektors. So ein Indexpaar zeigt auf den Anfang der noch nicht extrahierten Werte des Streams. Die Werte sollen mit Hilfe einer shuffle Operation von den Stream in einen SIMD-Vektor kopiert werden. Eine shuffle Operation hat drei Operanten, zwei Quellenvektoren und ein Patternvektor aus 16 Bytes, der für jedes Byte des Zielvektores angibt, welches der 32 Quellbytes kopiert werden soll. Die Quellvektoren sind die ersten beiden SIMD-Vektoren des Wertestreams. In den Pattern sind für Nullen spezielle Einträge, für die nicht-Nullen müssen die Bytes aus den Zielvektoren angegeben werden. Diese werden immer aufsteigend verwendet. Das Pattern ist also von den Positionen der Einsen im Bitmap und den Index des ersten Wertes abhängig. Für die Bitmaps gibt es 16 Möglichkeiten, für den Index vier. Insgesamt gibt es also 64 verschiedene Pattern. Diese kann man sich vor berechnen. Die Abbildung 4 zeigt diese Operation schematisch, wobei die shuffle Operation auf 4 Byte Wörtern und nicht auf einzelnen Bytes arbeitet.

Um die Position des ersten nicht bearbeiteten Wertes zu aktualisieren, muss man die Einsen in den Bitmap zählen. Diese Zahl addiert man auf den zweiten Index, wenn dieser grösser als vier wird, verkleinert man ihn um vier und zählt den anderen Index um eins hoch. Wie diese Operation vereinfacht werden kann wird in Kapitel 2.4.1 beschrieben.

Das ganze muss vier mal für jeden Block durchgeführt werden. Das erscheint für das Laden eines SIMD-Vektors viel Aufwand zu sein. Aber wenn man es mit den Laden eines einzelnen Wertes von einer nicht 128 Bit ausgerichteten Adresse vergleicht, ist der Aufwand ähnlich. Auch hier braucht man einen shuffle Operation. Das Pattern wird hier aus der Adresse berechnet.

2.3.2.4 SIMD Eigenschaften

Nach der Dekompression hat man die Matrixdaten in SIMD Registern geladen. Die Teile der zugehörigen Eingabe- und Ergebnisvektoren sind passend ausgerichtet. Man kann damit also gut rechnen.

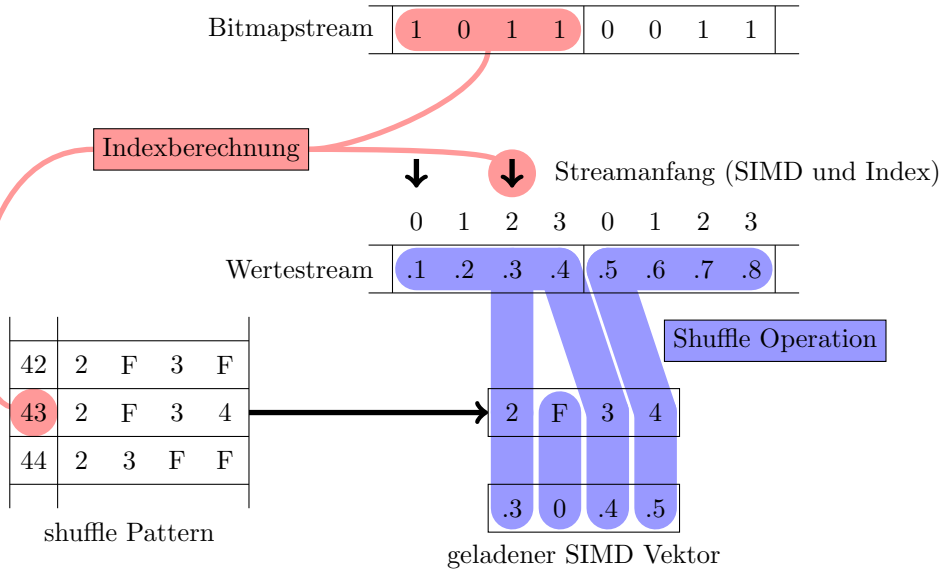

Abbildung 4: Laden eines SIMD Vektors aus den Wertestream

Die explizite Speicherung der Zeilenindizes erspart eine Schleife, in der man bei komprimierten Zeilenindizes, nach den richtigen Zeilenindex suchen muss.

Die vertikale Ausrichtung der Blöcke sorgt für eine einfachere Multiplikation, weil man nur den Eingabevektors aufteilen muss. Bei der horizontalen Ausrichtung kann man zwar die Multiplikation sofort durchführen, muss dann aber die horizontale Additionen mit den Ergebnissen durchführen. Die Abbildung 5 zeigt Pseudocode beider Varianten. Die Abbildung 6 zeigt die Register während des horizontalen Addierens.

2.4 Anpassungen für die Implementierung

Für die Implementierung wurden einige Änderungen am Format vorgenommen um die Berechnung zu beschleunigen.

2.4.1 Zusätzliche Informationen

Bei der Dekomprimierung, die im Kapitel 2.3.2.3 beschrieben wurde, muss man zählen, wieviele Werte man schon entpackt hat um zu wissen wo im Speicher die nächsten Werte liegen. Dazu muss man die gesetzten Bits in den 4 Bit Bitmaps zählen. Die Operationen, die dafür benötigt werden sollen eingespart werden. Deshalb werden neben den Bitmap für einen SIMD-Vektor noch weitere Daten gespeichert. Zwei Bits speichern den Offset der ersten Nicht-Null-Eintrag von einer SIMD-Vektor Grenze. Damit muss dieser Wert nicht berechnet werden, allerdings wurde der Wert auch verwendet um die Position im Stream zu aktualisieren. Deshalb wird ein weiters Bit verwendet um zu signalisieren, ob die Position im Stream nach den Laden um eins inkrementiert werden muss.

2.4.2 Padding

Die Daten werden in Partitionen auf die SPEs kopiert. In der Implementierung geschieht das mit festen Grössen. Dabei ist es wichtig, dass eine Partition der Werteübertragung, mit einen Wert endet, welcher der letzte Wert von einer achter Gruppe von Blöcken ist. Das hat mit Implementierungsdetails zu tun, die in Kapitel 2.5.2 beschrieben werden. Um das zu erreichen, werden zum einen Blöcke ohne Einträge eingefügt und zum anderen werden Blöcke verdoppelt. Wobei jede der Kopien einen Teil der Werte enthält. Abbildung 7 zeigt, wie das Bitmaparray aufgefüllt wird, wenn die dazugehörigen Werte über einer Partitionsgrenze verteilt liegen.

```
x0 = shuffle x;          b0 = a0 * x;
x1 = shuffle x;          b1 = a1 * x;
x2 = shuffle x;          b2 = a2 * x;
x3 = shuffle x;          b3 = a3 * x;
b = b + a0 * x0;         t0 = shuffle b0, b1;
b = b + a1 * x1;         t1 = shuffle b0, b1;
b = b + a2 * x2;         t2 = shuffle b2, b3;
b = b + a3 * x3;         t3 = shuffle b2, b3;
                         t0 = t0 + t1;
                         t2 = t2 + t3;
                         t1 = shuffle t0, t2;
                         t3 = shuffle t0, t2;
                         t0 = t1 + t3;
                         b = b + t0;
```

Abbildung 5: Multiplikation mit Spalten bzw. Zeilen ausgerichteten Vektoren

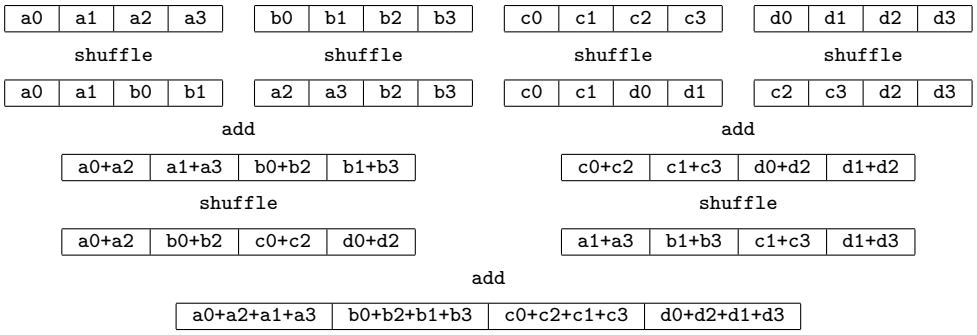

Abbildung 6: Horizontales Addieren

2.5 Implementierung

2.5.1 Entrollen

Die Datentransfers der SPU so ausgerichtet, dass der Datenstream immer bei durch acht teilbaren Block Indizes leer läuft, damit man die Schleifen besser entrollen kann. Dieses Entrollen hat nicht nur Performance Gründe. Da die Strukturdaten sechs Bytes (zwei Indizes plus vier für die Bitmaps) pro Block gross sind, wiederholt sich der Offset auf die 128 Bit ausgerichteten SIMD-Vektoren alle acht Blöcke. Um mit hart codierten Offsets arbeiten zu können muss man in einem Schleifendurchgang acht Blöcke abarbeiten. Dieses Verhalten nutzt auch die Datenstruktur, die für jeweils acht Blöcke eine Struktur enthält und deren Grösse dadurch mit 48 Byte ein Vielfaches von 16 Byte ist. Damit laesst sich die Struktur in genau drei SIMD-Vektoren speichern.

2.5.2 Speicherausrichtung

Damit die Datenübertragung parallel zur Rechenarbeit ablaufen kann, wurden double Buffering Techniken verwendet. Da der Cellprozessor nur 128 Byte ausgerichtete Daten vernünftig übertragen kann, müssen die Daten im Hauptspeicher und deren Kopie im lokalen Speicher der SPEs 128 Byte ausgerichtet werden.

Da die Datentransfers ausserhalb dieser Schleife organisiert werden, und man innerhalb der Schleife gerne linear auf den Speicher arbeitet ohne auf das leer Laufen der Buffer zu achten, müssen die Blöcke für den Datentransfer, so liegen, dass sie mit durch acht teilbaren Blocken beginnen und passend enden. Das erreicht man durch die in Kapitel 2.4.2 beschriebenen Modifikationen am Speicherformat. Damit sie Speicherausrichting zwischen Datenübertragung und Berechnungsschleife klappt, müssen diese Daten also sogar auf die Buffergrösse ausgerichtet sein.

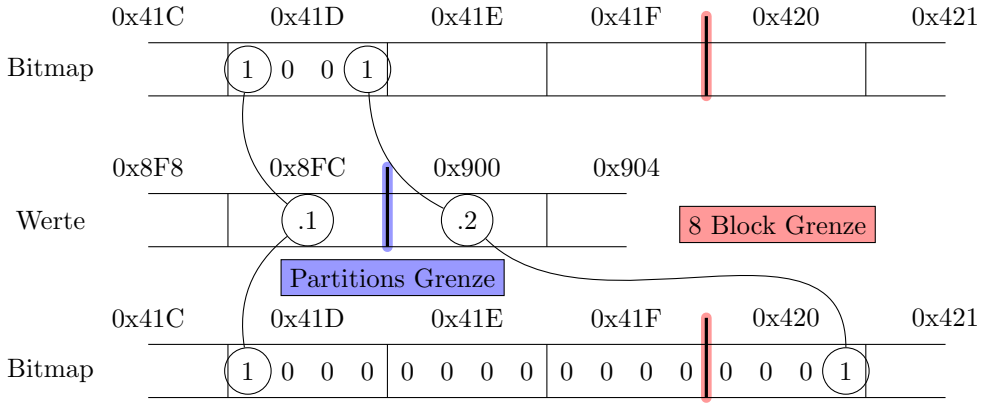

Abbildung 7: Padding des Bitmaparrays

2.5.3 Aufteilung einer fma

Für die Berechnung hat sich als längste Abhängigkeit die Multiplikationen untereinander erwiesen. Zunächst erscheint es als ob man einen Block am schnellsten mit vier fmas (eine Multiplikation und eine Addition in einer Operation) berechnen kann. Es hat sich aber herausgestellt, dass es günstiger ist mit einer Multiplikation (fm), drei fmas und einer Addition (fa) zu berechnen. Da sich dadurch die Berechnung zweier Blöcke überschneiden können. Abbildung 8 zeigt Diagramme der beiden Ansätze für zwei Blöcke. Das Laden der Matrixwerte und des Eingabevektors sind nicht eingezeichnet. Das Laden (ld) und Speichern (st) bezieht sich auf den Ergebnisvektor. Dieser kann nur gespeichert werden, wenn der vorherige gespeichert wurde, weil es der gleiche Wert sein kann. Das kann der Cell Prozessor aber schon im nächsten Takt.

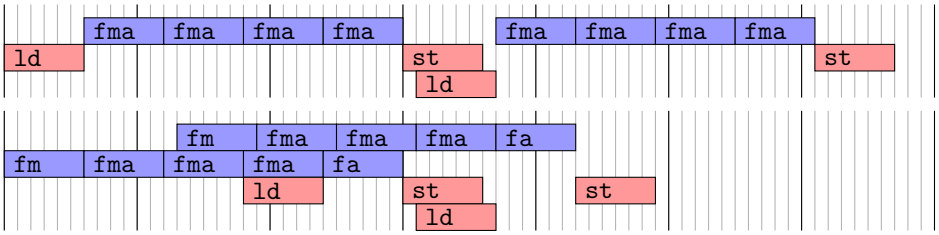

Abbildung 8: Pipelines der Multiplikation zweier Blöcke, nur mit fma und mit aufgeteiltem fma

Eine weitere Aufteilung um noch mehr Parallelität zu ermöglichen ist nicht nötig, da für die Dekomprimierung genug zu tun ist.

2.5.4 Parallelisierung

Beim Load Balancing wird ausgenutzt, dass die Multiplikation oft ausgeführt wird. Am Anfang rechnet jede SPE gleich ziele Zeilen und bei jeder Multiplikation wird gemessen wie lange jede einzelne SPE gerechnet hat. Die SPEs, die wenig zu rechnet hatten bekommen bei der nächsten Iteration mehr zu tun.

3 Ergebnisse

3.1 Benchmark

Um Ergebnisse vergleichen zu können, wurden bei den Benchmarks Matrizen verwendet, für die schon Ergebnisse mit den Cell bekannt sind. Die Matrizen stammen von unterschiedlichen Anwendungen und haben unterschiedliche Eigenschaften in Bezug auf die Verteilung und Dichte der

Nicht-Null-Einträge. Sie sollen einen repräsentative Auswahl aus relevanten Problemen sein. [BEN] In der Tabelle 1 sind Grösse, Anzahl der Nicht-Null-Einträge und die Anwendung aus der die Matrizen stammen aufgelistet.

#	Name	n	m	nnz[1]	Anwendung
1	3dtuber	45k	45k	1,6M	Fluidmechanik
2	av41092	41k	41k	1,7M	2D PDGL
3	crystk02	14k	14k	491k	FEM
4	finan512	75k	75k	336k	Ökonomie
5	gupta1r	32k	32k	1,1M	Optimierungsproblem
6	memplus	18k	18k	126k	Schaltungssimulation
7	pwtr	37k	37k	181k	NASA Windtunnel
8	rim	23k	23k	1,0M	Fluidmechanik
9	shyy161	76k	76k	330k	Fluidmechanik
10	vibrobox	12k	12k	178k	Akustik

Tabelle 1: Dimensionen der Matrizen

3.2 Speicherverbrauch

Ein Ziel war es, dass die Matrizen weniger Speicherplatz verbrauchen. In der Tabelle 2 ist aufgelistet wieviel Speicherplatz die Matrizen in verschiedenen Formaten brauchen. Ausserdem ist angegeben wieviel das in Relation zum CRS Format ist. Für die BCRS und BCOO Formate wird eine Partitionierung angenommen, die zu 256x256 Blöcke grossen Partitionen führt. Weil die Reduzierung auf ein Byte für die Koordinaten so gravierende Auswirkungen hat, ist auch aufgeführt, wie gross ein CRS in 256x256 grossen Partitionen ist.

#	CRS		PCRS		BCRS 2x2		BCOO 4x4		CBCOO		Imp.	
	MiB	%	MiB	%	MiB	%	MiB	%	MiB	%	MiB	%
1	12,60	100	8,58	68,08	8,61	68,33	10,86	86,19	6,88	54,55	7,25	57,48
2	13,00	100	12,98	99,81	10,75	82,69	21,91	168,5	7,76	59,66	8,48	65,19
3	3,80	100	2,42	63,75	2,51	65,92	3,33	87,47	2,08	54,61	2,19	57,59
4	2,85	100	2,07	72,56	3,49	122,71	8,79	308,51	1,82	63,75	2,09	73,47
5	8,50	100	5,50	64,71	10,44	122,90	20,63	242,71	5,44	64,00	6,09	73,47
6	1,03	100	0,79	76,63	1,00	96,87	1,99	193,33	0,60	58,48	0,67	64,83
7	1,52	100	1,16	76,29	1,84	120,95	3,34	219,3	0,90	58,78	1,00	65,89
8	7,83	100	4,97	63,52	7,45	95,16	11,18	142,78	4,55	58,11	4,91	62,77
9	2,81	100	2,01	71,48	3,30	117,52	5,97	212,59	1,62	57,74	1,81	64,57
10	1,40	100	1,15	81,77	1,94	138,27	3,56	253,68	0,89	63,74	1,01	71,86
⌀		100		73,86		103,13		191,51		59,34		65,53

CRS:	CRS Format
PCRS:	CRS Format mit Partitionen
BCRS 2x2:	BCRS mit 2x2 Blöcken
BCOO 4x4:	BCOO mit 4x4 Blöcken
CBCOO:	Komprimiertes BCOO mit 4x4 Bloecken, wie in den Kapitel 2.3.1 beschrieben
Impl.:	Format der Implementierung, wie in Kapitel 2.4 beschrieben

Tabelle 2: Speicherverbrauch der einzelnen Matrizen in unterschiedlichen Formaten

Das komprimierte BCOO ist bei den getesteten Matrizen das kompakteste Format. Das bei der Implementierung verwendete Format ist nicht ganz so klein, aber deutlich schlanker als BCRS.

3.3 Gflop/s

Um die Ergebnisse zu vergleichen wurde eine einfache multithreaded Implementierung für einen Core2Quad geschrieben. Getestet wurde diese Version auf einem Q6600 mit vier Gigabyte RAM.

[1] Anzahl von Nicht-Null-Eintraege (number of non-zeros)

Ausserdem wurden zum Vergleich mit einer 2x2 BCRS Implementierung die Ergebnisse von [BCR] übernommen. Die Implementierung wurde sowohl auf der Playstation mit sechs als auch auf einer QS22 mit acht SPEs getestet. Um die Gesamtleistung zu beurteilen und zu vergleichen eignet sich der Gflop/s Wert. Dieser ist für die genannten Systeme und den ausgesuchten Matrizen in der Tabelle 3 angegeben.

	Intel	BCRS [2]	PS3	QS22
3dtuber	1,72	5,30	7,70	8,64
av41092	0,96	4,93	4,27	5,23
crystk02	1,00	5,39	7,39	8,21
finan512	0,53	2,35	2,28	2,69
gupta1r	0,98	5,20	3,64	4,31
memplus	0,63	2,43	2,15	1,95
pwtr	0,59	2,56	2,55	2,81
rim	1,78	5,00	5,87	6,76
shyy161	0,63	1,99	3,08	3,41
vibrobox	0,73	4,87	2,45	3,23
⌀	0,95	4,00	4,14	4,72

Keiner der Algorithmen nutzt die Symmetrie der symmetrischen Matrizen.

Tabelle 3: Geschwindigkeitsmessung

Man kann erkennen, dass das komprimierte BCOO Format bei manchen Matrizen schneller und bei anderen langsamer als die BCRS Variante ist. Der Kompremierungsgrad und auch die Differenz des Komprimierungsgrades scheint damit wenig zu tun zu haben.

3.4 Speicherbandbreite

Das Ergebnis soll anhand der PS3 Daten genauer analysiert werden. Was man neben den Rechendurchsatz messen kann, ist die genutzte Speicherbandbreite. In der Tabelle 4 ist diese aufgelistet.

	GB/s
3dtuber	19,94
av41092	15,84
crystk02	18,36
finan512	11,77
gupta1r	11,45
memplus	9,36
pwtr	12,74
rim	15,92
shyy161	14,79
vibrobox	9,96

Tabelle 4: Bandbreitensmessung

Bei den zwei Matrizen, bei deren Multiplikation mehr als 18 GB/s Bandbreite ausgenutzt werden, ist auch der Geschwindigkeitgewinn hoch.

3.5 Takte pro Block

Es ist aber schwer abzuschätzen, ob man die Rechenleistung voll ausnutzt. Um das besser abschätzen zu können wurde ein weiterer Wert gemessen. Das ist der "Takte pro Block" Wert. Er gibt an wieviele Takte man für einen Block rechnen kann bzw. wieviele man gerechnet hat.

Da die SPEs so einfach gebaut sind, kann man sehr gut vorhersagen, wie lange ein Durchlauf eines Stück Assemblercode dauert. Man kann sich dieses Timing auch vom Kompiler berechnen lassen.

[2] Die Werte stammen von [BCR]

Damit lässt sich eine Schranke berechnen, wie viele Takte man mindestens für einen Block braucht.. Für die innerste Schleife, die acht Blöcke multipliziert, ergeben sich für die Implementierung 236 Takte für einen Durchlauf. Das sind 29,5 Takte für einen Block. Das ist der Wert der theoretisch von dieser Implementierung erreicht werden kann, wenn alle DMA Aktionen kostenlos wären.

Man kann auch berechnen wie schnell eine Implementierung sein muss, damit sie bei der Berechnung mit einer bestimmten Matrix die volle Bandbreite nutzt. Dazu muss man bestimmen welche Daten übertragen werden müssen und wie lange das dauert. Daraus kann man berechnen wieviele Takte man Zeit hat um die übertragenen Blöcke zu berechnen.

Aus der Laufzeit lässt sich ausserdem berechnen, wie lange man tatsächlich an einem Block gerechnet hat. In der Tabelle 5 sind diese Zahlen aufgeführt.

	Takte/Block	
	nötig	erreicht
3dtuber	32,11	44,56
av41092	19,39	41,87
crystk02	31,29	45,72
finan512	11,42	35,36
gupta1r	13,30	34,36
memplus	16,87	64,35
pwtr	15,31	45,28
rim	20,45	36,06
shyy161	15,22	37,52
vibrobox	13,53	46,37

Tabelle 5: Takte pro Block

Für die beiden Matrizen, die einen Wert über den theoretisch erreichbaren 30 haben, wird auch die beste Bandbreitenauslastung erreicht. Viele der Werte, die nötig wären um die Bandbreite voll auszunutzen, sind unter den 30. Bei solchen Matrizen erwartet man aber, dass der gemessene Wert näher am Limit liegt.

3.6 Load Balancing

Es soll untersucht werden, wieso das bei vielen Matrizen nicht erreicht werden kann.

Ein Problem ist, dass das Load Balancing zwischen den SPEs nur sehr grob ist. Die Implementierung verzichtet auf einen Synchronisierung zwischen den SPEs. Deshalb ist es nicht möglich, dass mehrere SPEs mit den gleichen Matrixzeilen rechnen. Es fehlen auch die Informationen um Partitionen weiter aufzuteilen, weil man nur mit viel Aufwand den Anfang einer Zeile innerhalb einer Partition findet. Deshalb ist es nur möglich die Arbeit in 1024 Zeilen Einheiten zu verteilen. Dadurch werden verschiedenen SPEs unterschiedlich stark belastet. Und einige rechnen am Ende einer Multiplikation nichts mehr. In der Tabelle 6 steht, wieviel Zeit der Rechenzeit, die SPEs nur gewartet haben. Ausserdem ist dort auch der “Takte pro Block” Wert nur für die Zeiten in denen die SPEs rechnen aufgeführt.

	idle%	Takte / Block
3dtuber	9,11	40,50
av41092	20,30	33,37
crystk02	20,10	36,53
finan512	6,38	33,30
gupta1r	8,15	31,56
memplus	41,43	37,69
pwtr	16,81	37,67
rim	8,40	33,03
shyy161	4,74	35,74
vibrobox	25,19	34,69

Tabelle 6: Takte pro Block während wirklich gerechnet wird

Das Load Balancing ist so nicht sehr erfolgreich. Hier kann man mit feineren Partitionen sicher noch mehr Leistung erreichen. Aber die Zahlen zeigen auch, dass die Berechnung grundsätzlich schnell durchgeführt werden können.

3.7 Ausblick

Es konnte gezeigt werden, dass man die Matrix Vektor Multiplikation mit dünn besetzten Matrizen auf der CBEA mit einem geschickten Datenformat beschleunigen kann. Kapitel 3.6 zeigt ausserdem, dass es noch Potenzial für weitere Verbesserungen gibt. Eine weitere Möglichkeit einer Verbesserung ist ein kombiniertes Verfahren, bei den einzelne Blöcke als komprimiertes BCOO oder als BCRS abgespeichert werden. Pro Partition kann man jeweils einen Stream speichern. Diese Streams kann man hintereinander abarbeiten, weil die Reihenfolge der Berechnung der Blocke innerhalb einer Partition beliebig ist. Solche Verfahren sind vor allen für hybride Architekturen, wie sie beim Roadrunner [RRU] verwendet werden interessant, weil die unterschiedlichen Prozessorkerne sehr unterschiedliche Eigenschaften haben. Als Entscheidungskriterium kann der in Kapitel 3.5 vorgestellte Wert dienen. Diesen kann man auch für einzelne Blöcke oder Partitionen berechnen.

Das Verfahren, dass man ein Bitmap parallel zu einem Datenstream speichert um damit Nullen darzustellen, kann man auch für andere Daten als Matrizen verwenden.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[CEL] C. R. Johns, D. A. Brokenshire: *Introduction to the Cell Broadband Engine Architecture*, IBM J. Res. & Dev. Vol. 51 NO. 5 503-519 (2007)

[PS3] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, George Bosilca: *A Rough Guide to Scientific Computing On the PlayStation 3*, Computer Science Department, University of Tennessee (2007)

[SPM] Shahnaz, R., Usman, A., Chughtai, I.R.: *Review of Storage Techniques for Sparse Matrices*, 9th International Multitopic Conference, IEEE INMIC 2005 Volume , Issue , 24-25 Dec. 2005 Page(s):1 - 7 (2005)

[BCR] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick: *Scientific Computing Kernels on the Cell Processor*, International Journal of Parallel Programming (2007)

[BEN] Richard Vuduc: *Automatic Performance Tuning of Sparse Matrix Kernels*, Ph.D. Thesis Computer Science Division, U.C. Berkeley (2003)

[RRU] John Turner and Andy White: *Los Alamos National Lab and IBM Bring Computing into the Petascale Era*, SIAM News, Volume 41, Number 6, July/August 2008 (2008)