Praktikumsbericht

Jens Breitbart

01.08.2006 - 31.10.2006

Inhaltsverzeichnis

1 IBM

1.1 Weltweit

Die Industrial Buisness Machines Cooperation (IBM) wurde im Jahr 1888 gegründet und ist seit dem 15.06.1911 als Kapitalgesellschaft amtlich eingetragen. Das Hauptquartier von IBM liegt in Armonk, New York, USA. Die etwa 329.000 Beschäftigten von IBM erwirtschaften jährlich eine Umsatz von ca. 91 Milliarden US Dollar [IBM05]. Mit Entwicklern und Beratern in 170 Ländern und weltweit ca. 30 Entwicklungslaboratorien gehört IBM zu einer der weltweit größten IT Firmen.

1.2 IBM Deutschland GmbH

Die IBM Deutschland GmbH beschäftigt 25.000 Mitarbeiter an rund 40 Standorten. Die Zentrale liegt in Stuttgart-Vaihingen.

1.2.1 Forschung und Entwicklungszentrum in Böblingen

Das Forschungs- und Entwicklunszentrum in Böblingen, wo auch mein Arbeitsplatz lag, ist das größte IBM Entwicklungszentrum außerhalb der USA. Dort sind im Moment 1.700 Mitarbeiter tätig [IBM]. Im Gegensatz zu den meisten IBM Laboratorien ist das Labor in Böblingen nicht Teil einer Entwicklungsabteilung (z.B. Software), sondern arbeitet für alle.

1.2.2 Abteilung - System z Software Evaluation and Test

Die Abteilung, in der ich gearbeitet habe, ist Teil der IBM System & Technology Group und gleichzeitig in das Linux Technology Center (LTC) eingebunden. Das LTC ist eine weltweite Entwicklergruppe innerhalb der IBM. Diese arbeitet einerseits mit Open Source Projekten und andererseits mit IBM selbst, um Linux als Enterprise OS zu etablieren und gleichzeitig IBMs Marktanteile im Linuxmarkt zu erhöhen. Hauptaufgabe der Abteilung ist es, Linux und Virtualisierungssoftware für IBM System z zu unterstützen und zu entwickeln. Die Aktivitäten konzentrieren sich aktuell unter anderem auf folgende Gebiete:

- Linux on System z und Linux on Cell Broadband Engine Test
- Linux on System z, Linux on Cell Broadband Engine und z/VSE Entwicklung
- Linux und Open Virtualization Entwicklung und Evaluierung

- Perfomance Evaluierung
- z/VM Produktentwicklung

2 Projekt

2.1 Umfeld

Während der Bearbeitung meines Projektes mit dem Test Team habe ich für Linux on Cell Broadband Engine (CBE) earbeitet. Die CBE ist die erste Inkarnation einer sehr neuen Prozessorarchitektur, die aktuell noch nicht in frei verfügbaren Produkten Verwendung findet.

Linux wurde bereits sehr früh in der Entwicklung der CBE für eben diese portiert, die Bibliotheken zur Benutzung spezieller Funktionalitäten befinden sich allerdings weiterhin aktiv in der Entwicklung.

2.2 Übersicht

Die Aufgabe meines Praktikums bestand im Entwurf und der Implementation einer C++ Bibliothek. Diese Bibliothek sollte die Benutzung CBE-spezifischer Funktionalitäten ermöglichen. Der Funktionsumfang umfasst u.a. das Erzeugen spezieller Threads zum Ablauf auf den Synergistic Prozessor Element und entsprechende Möglichkeiten zur Synchronisation bzw. Kommunikation. Das Hauptdesignziel beim Erstellen der Bibliothek war eine einfache Handhabung für unerfahrene Benutzer. Meine Aufgabe habe ich hauptsächlich auf einem mir zur Verfügung gestellten Blade mit 2 Cell Broadband Engine Prozessoren und einer speziell angepassten Version der GNU-Toolchain[1] bearbeitet.

2.3 Nutzen für das Team

Die Bibliothek wird hauptsächlich vom Linux on Cell Broadband Engine Test Team genutzt werden. Zum einen werden die von mir entwickelten Tests als allgemeine Testcases zum Testen der Bibliothek verwendet. Zum anderen ermöglicht die Bibliothek auch relativ einfaches und schnelles Erstellen neuer Testcases.

Im Gegensatz zu den bis jetzt Existierenden sind meine Testcases nicht in C sondern in C++ geschrieben und ermöglichen somit Tests speziell angepasster C++ Kompiler.

[1] Bestehend aus GNU-Kompilern (gcc, g++) und Debugger (gdb).

3 Cell Broadband Engine Architecture

Literatur für diesen Abschnitt, wenn nicht anders angegeben: [IBM06a] und [IBM06b].

3.1 Einführung

Die Cell Broadband Engine Architecture (CBEA) ist eine neue Prozessorarchitektur, welche die bereits bestehende 64bit PowerPC-Architektur erweitert. Entwickelt wurde sie in Kooperation von Sony, Toshiba und IBM (STI).

Bei der CBEA handelt es sich um eine heterogene Multicore-Architektur. Solche Architekturen unterscheiden sich zu den, z.B. im x86-Bereich verbreiteten Multicore-Architekturen dadurch, dass sie aus unterschiedlichen Prozessorelementen bestehen. Im Fall der CBEA hat man zwei unterschiedliche Elemente. Zum einen die so genannten PowerPC Prozessor Elemente (PPE) und zum anderen die Synergistic Prozessor Elemente (SPE).

Erstgenannte sind vollwertige 64-bit PowerPC-Prozessoren. Sie können 32bit- und 64bit-Betriebsysteme und -Anwendungen ausführen. Im Gegensatz dazu sind SPEs daraufhin optimiert, rechenlastige Anwendungen auszuführen. Jede SPE verfügt über eigenen Speicher und hat gleichzeitig auch Zugriff auf den Hauptspeicher inklusive der in den Hauptspeicher eingeblendeten Speicherbereiche. SPEs sind grundsätzlich unabhängig voneinander, meistens allerdings abhängig von einer PPE, welche das Betriebssystem oder einen Kontrollthread für eine SPE-Anwendung ausführt.

3.2 Probleme bei dem Entwurf einer Prozessorarchitektur - und die Lösung der CBEA

Aktuell gibt es drei limitierende Faktoren für die Prozessorperformance: Energieverbrauch, Speicherperformance und Taktfrequenz.

3.2.1 Energieverbrauch

Die Leistungsfähigkeit eines Prozessors wird immer mehr davon abhängig, wie hoch seine Leistungsabgabe ist und immer weniger von der Anzahl der auf dem Prozessor platzierbaren Transistoren. Somit ist es nötig, die Leistungsabgabe so stark wie möglich zu reduzieren.

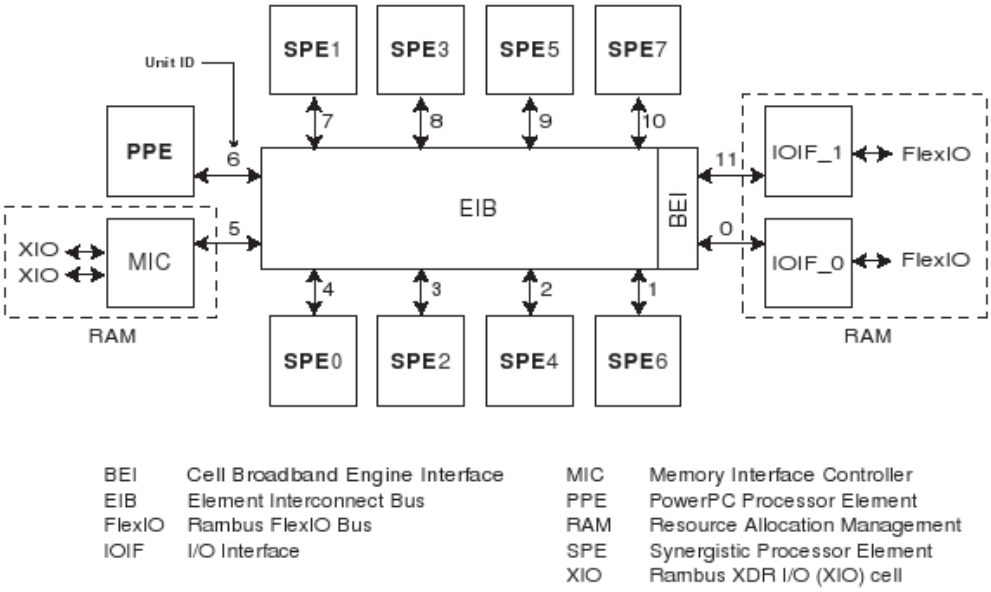

Abbildung 3.1: Übersichtsskizze der Cell Broadband Engine [IBM06a]

Die CBEA versucht dies durch Spezialisierung einzelner Prozessorelemente zu erreichen, damit diese ihre Aufgabe effizienter abarbeiten können. So wurden die PPEs, wie bereits erwähnt daraufhin optimiert, Betriebsysteme ausführen zu können.

3.2.2 Speicherperformance

Die Verzögerung beim Zugriff auf den Hauptspeicher nähert sich immer weiter 1.000 Takten an. Aufgrund einer solch großen Verzögerung hängt die Performance der meisten Programme recht stark von der Performance der Speicherzugriffe ab. Die SPE bietet zwei Mechanismen, um diesem Problem entgegenzuwirken:

- eine 3-stufige Speicherhierarchie bestehend aus Hauptspeicher, lokalem Speicher und relativ großen Registern.
- asynchrone Zugriffe auf den Hauptspeicher, die eine Überlappung von Berechnung und Datentransfer ermöglichen.

3.2.3 Taktfrequenz

Eine direkte Methode zur Performancesteigerung eines Prozessors ist die Erhöhung der Taktfrequenz. Viele konventionelle Prozessorarchitekturen haben in den letzten Jahren versucht, dies durch die Verlängerung ihrer Befehlspipeline zu erreichen.[1] Dieses

[1] siehe z.B. Intels Netburst Architektur (Pentium4).

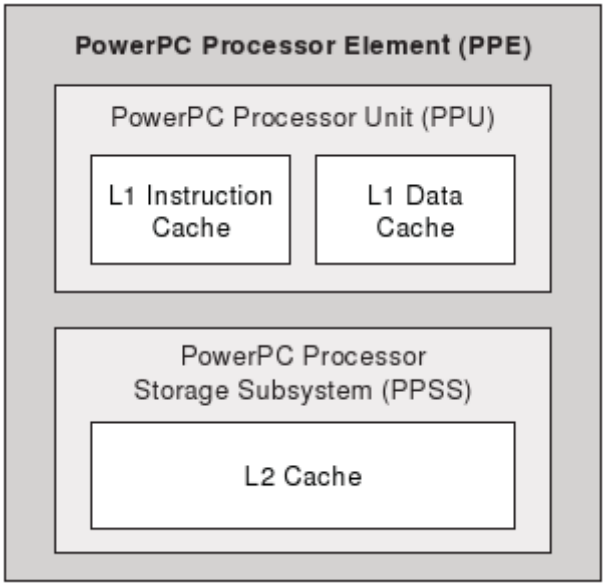

Abbildung 3.2: Übersichtsskizze einer PPE [IBM06a]

Vorgehen hat allerdings einen Punkt erreicht, an dem kaum noch Fortschritte erzielt werden können.

Die CBEA ermöglicht auch hier wieder durch Spezialisierung der einzelnen Prozessorelemente, Overhead zu reduzieren und kann so höhere Taktfrequenzen erreichen. Die SPE wurde z.B. mit vielen Registern ausgestattet, um auf Techniken wie "register-renaming" verzichten zu können.

3.3 PowerPC Prozessor Element

Die PPE ist ein zur PowerPC-Architektur[2] konformer 64bit-RISC[3]-Prozessor. Aufgrund dieser Kompatibilität ist es möglich, Programme, die für die PowerPC-Architektur geschrieben wurden, ohne Modifikation auf der CBEA ausführen zu können. Weiterhin unterstützt die PPE die Vector/SIMD[4]-Erweiterung AltiVec.

Die Aufgabe der PPE ist es, das Betriebsystem auszuführen, welches alle PPE- und SPE-Programme kontrolliert. Wie in Abbildung 3.2 dargestellt, besteht die PPE grundsätzlich aus zwei Teilen. Zum einen aus der PowerPC Processor Unit (PPU), welche die eigentlichen Berechnungen ausführt. Zum anderen aus dem PowerPC Processor Storage Subsystem (PPSS), welches Speicheranfragen von der PPU und Anfragen an

[2] Version 2.02
[3] Reduced Instruction Set Computing
[4] Single Instruction Multiple Data

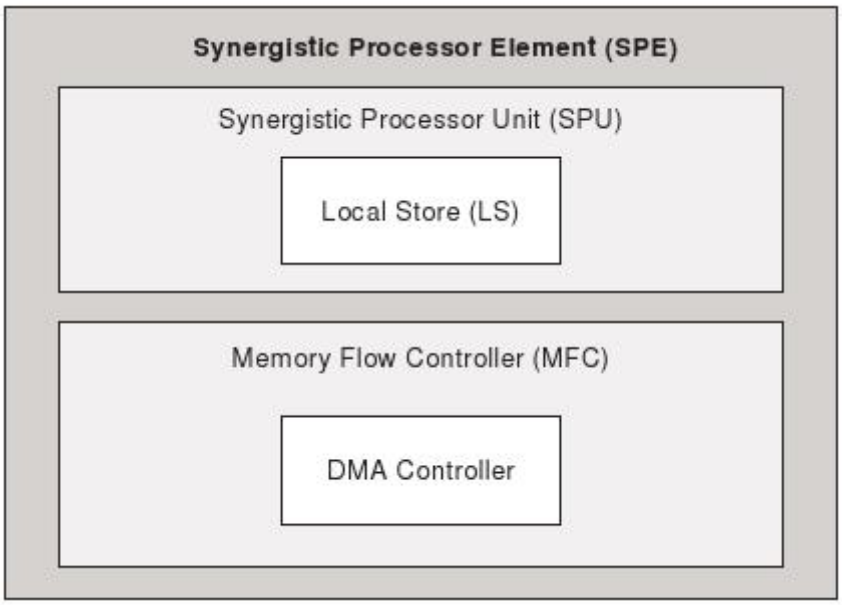

Abbildung 3.3: Übersichtsskizze einer SPE [IBM06a]

die PPE von SPEs oder I/O-Geräten bearbeitet.

Instruktionen werden auf der PPE in der Reihenfolge abgearbeitet, in der sie im Programmcode vorliegen ("in-order"). Um den Performanceverlust im Vergleich zu "out-of-order"-Architekturen zu begrenzen, ist die gleichzeitige Ausführung von zwei Threads möglich.[5]

3.4 Synergistic Prozessor Element

Im Gegensatz zur PPE, welche den PowerPC-Instruktionssatz ausführt, kann die SPE einen neuen Instrutionssatz ausführen, die so genannten Synergistic Processor Unit Instruction Set Architecture. Die SPE ist ein für rechenaufwändige Anwendungen spezialisierter 128bit-RISC-Prozessor. Er unterstützt sowohl skalare als auch SIMD-Berechnungen.

Wie Abbildung 3.3 zeigt, ist auch die SPE grundsätzlich in zwei Teile aufgeteilt. Sie besteht zum einen aus der Synergistic Processor Unit (SPU), zum anderen dem Memory Flow Controller (MFC). Die SPU beinhaltet unter anderem einen relativ kleinen lokalen Speicher und ist in der Lage, Instruktionen auszuführen. In dem lokalem Speicher der SPU müssen sowohl das auszuführende Programm als auch die benötigten Daten abgelegt werden. Die Übertragung von Daten in den lokalen Speicher wird vom

[5] Genau betrachtet werden hier keine zwei Threads echt gleichzeitig ausgeführt, sondern die Befehle der Threads werden taktalternierend ausgeführt [VS05].

MFC übernommen.

Wie ebenfalls auf der Abbildung zu erkennen ist, verfügt eine SPE über keinen Cache. Diese Tatsache zusammen mit den relativ einfachen Regeln zur Abarbeitung der Instruktions-Pipeline (u.a. "in-order") ermöglichen eine statische Vorhersage der Performance des Codes, um z.B. die "dual-issue-pipeline" besser ausnutzen zu können. Eine statische Analyse kann allerdings nur für Bereiche sinnvoll ausgeführt werden, in denen keine Kommunikation mit anderen Prozessorelementen stattfindet.

3.5 Speichermodell

Der Speicher der CBEA besteht aus mehreren Teilen. Zum einen verfügen die SPEs, wie bereits erwähnt, über lokalen Speicher, zum anderen besitzt das System selbst auch einen Hauptspeicher. Beide Speicherbereich werden allerdings von Betriebssystem und Hardware in einen gemeinsamen Adressbereich abgebildet.[6] Eine Adresse aus diesem Bereich bezeichnet man als effektive Adresse ("effective address", EA). Die PPE kann auf eine EA durch normale Speicherzugriffsoperationen zugreifen, die SPE kann dies nur über einen DMA-Transfer (siehe Abschnitt 3.6).

Im Gegensatz zu der "in-order"-Ausführung der Instruktionen ist es bei Speicherzugriffen nicht garantiert, dass eine Instruktion abgeschlossen ist, bevor die nächste ausgeführt wird. Innerhalb eines Threads führt dies zu keinem Problem. Sind aber mehrere Threads beteiligt, kann es passieren, dass nicht alle Threads den selben Wert von einer EA lesen. Möchte man zu einem bestimmten Zeitpunkt sicherstellen, dass alle Threads den selben Wert lesen, muß eine Synchronisation des Speichers durchgeführt werden.

3.6 Kommunikation

Um eine Architektur wie die CBEA effektiv ausnutzen zu können, muß es möglich sein, zwischen den einzelnen Prozessorelementen Daten austauschen zu können. Die CBEA sieht hierfür drei unterschiedliche Kommunikationsmethoden vor:

- Mailboxen

  Mailboxen sind eine Hardwareimplementierung von Nachrichtenwarteschlangen. Jede SPE verfügt über 3 solche Mailboxen, wobei allerdings nur 2 vom Entwickler genutzt werden sollten. Eine der Mailboxen wird für die Interruptbearbeitung benötigt. Die anderen beiden werden verwendet, um Nachrichten von der SPU bzw. an die SPU zu speichern. Eine Nachricht hat immer eine Größe von 32 Bit.

- Signale

  Signale sind wie auch Mailboxen in der Lage, maximal 32 Bit pro Signal zu übertragen. Im Gegensatz zu Mailboxen bieten sie allerdings keine Warteschlange

[6] Zusätzlich zu den hier genannten Speichern werden auch noch die Register der Mailboxen und Signalkanäle abgebildet.

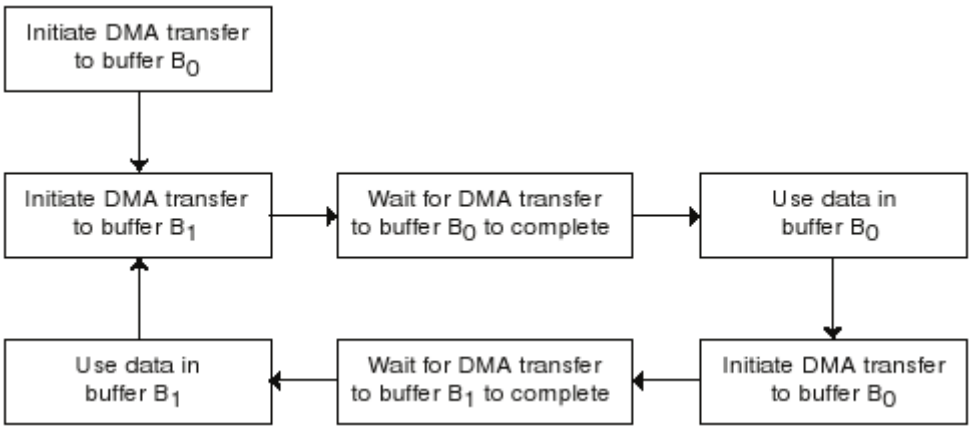

Abbildung 3.4: Ablaufdiagramm des "double buffering" [IBM06b]

an. In jedem Signalkanal kann immer nur ein Signal aktiv sein. Jede SPE stellt 2 Kanäle, die vom Entwickler genutzt werden können, zur Verfügung. Man kann Signale an ein Prozessorelement schicken (Unicast) oder auch gleichzeitig an mehrere (Multicast).

- DMA-Transfer

  Ein DMA-Transfer ermöglicht es einem Prozessorelement, auf eine EA zuzugreifen. Eine SPE kann also durch einen DMA-Transfer auf Daten außerhalb ihres lokalen Speichers zugreifen bzw. eine PPE auf den lokalen Speicher einer SPE. DMA-Transfers unterscheiden sich grundlegend von den beiden bisher genannten Kommunikationsmethoden. Durch einen DMA-Transfer können bis zu 16 KB an Daten übertragen werden. Weiterhin ist es möglich, dass mehrere DMA-Transfers gleichzeitig ausgeführt werden. Durchgeführt werden sie vom MFC einer SPE bzw. dem PPSS einer PPE und sind somit unabhängig von den Instruktionen, die von der SPU bzw. PPU ausgeführt werden. Dies ermöglicht, wie bereits in Abschnitt 3.2.2 erwähnt, dass gleichzeitig Daten transferiert und Berechnungen ausgeführt werden. Um eine SPE möglichst effizient nutzen zu können, empfiehlt sich deshalb, abhängig von ihrer Aufgabe, eine Technik namens "double buffering". Hierbei werden, während die SPU gerade auf aktuellen Daten rechnet, die Daten für die nächste Berechnung in den lokalen Speicher der SPE transferiert (siehe Abbildung 3.4).

Alle drei Kommunikationsarten werden intern über den so genannten Element Interchange Bus (EIB, siehe Abbildung 3.1) durchgeführt. Mailboxen und Signale werden üblicherweise dafür verwendet, um Speicheradressen oder Benachrichtigungen, z.B. über das Ende einer Berechnung, zu übertragen.

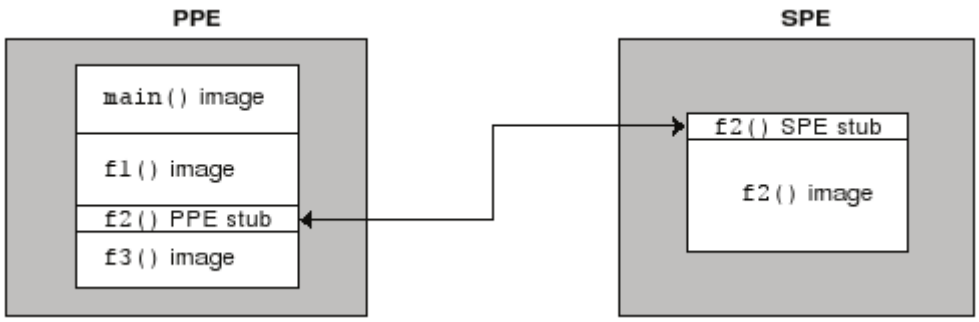

Abbildung 3.5: Beispiel für das "Function-Offload" Modell [IBM06a]

3.7 Programmiermodelle

Die CBEA unterstüzt viele verschiedene Programmiermodelle. Im folgenden werden nur einige ausgewählte vorgestellt.

3.7.1 Function Offload Model

Im so genannten "Function-Offload"-Modell (oder auch "RPC"-Modell genannt) werden SPEs benutzt, um performancekritische Berechnungen zu beschleunigen. Dies geschieht, indem die eigentliche Berechnung dieser Funktion auf der SPE ausführt wird. Auf der PPE wird dafür nur ein Stub erzeugt, der dafür sorgt, dass ein SPU-Thread erzeugt wird und alle nötigen Daten der SPE zur Verfügung gestellt werden. Der Vorteil dieses Modells ist, dass bestehende Programme relativ leicht angepasst und die benötigten Stubs durch einen Kompiler erzeugt werden können.

3.7.2 Streaming Model

Im "Streaming"-Modell werden mehrere SPEs in einer Pipeline zusammengeschlossen. Die zu berechnenden Daten werden von einer SPE zur nächsten transferiert, wobei jede SPE eine bestimmte Berechnung durchführt. Die PPE fungiert hier nur als Kontrollinstanz und überwacht die einzelnen SPEs. Dieses Modell ermöglicht eine sehr gute Performance, da die Daten sehr lange innerhalb des Prozessors gehalten werden und somit kaum Datentransfer aus dem Hauptspeicher nötig ist.

3.7.3 Shared Memory Multiprocessor Model

Die CBEA kann wie ein symmetrisches Multiprozessorsystem (SMP) mit zwei unterschiedlichen Instruktionssätzen programmiert werden. Zugriffe auf gemeinsamen Speicher müssen nur durch entsprechende DMA-Transfers ersetzt werden.

3.8 Cell Broadband Engine

Die Cell Broandband Engine ist die erste Ausführung der CBEA. Diese verfügt über eine PPE mit acht SPEs. Jede SPE verfügt über 256 KB lokalen Speicher.

4 C(ell)++

4.1 Einführung

C(ell)++ bezeichnet die während meines Praktikums entwickelte Bibliothek. Sie wurde vollständig in C++ implementiert und benutzt selbst folgende Bibliotheken bzw. C- und C++ Spracherweiterungen:

- SPE Runtime Management Library Version 1.1 (libSPE) [IBM06d]
- Cell Broadband Engine SDK Libraries Version 1.1 - Sync Library [IBM06c]
- SPU C/C++ Spracherweiterungen [IBM06e]
- AltiVec C/C++ Spracherweiterungen [AVP99]

4.2 Designziele

Beim Entwurf der Bibliothek gab es drei grundlegende Ziele, welche alle während der Planung und Entwicklung die selbe Priorität hatten.

4.2.1 Einfache Benutzung durch Klienten

Die CBEA fordert vom Klienten[1] nicht nur die Nutzung mehrerer neuer Bibliotheken bzw. Spracherweiterungen, sondern verlangt auch ein starkes Umdenken bei der Planung und dem Entwurf einer Anwendung. Dies liegt am, im Vergleich zu den allgemein weit verbreiteten Prozessorarchitekturen, ungewöhnlichen Aufbau der CBEA. Ziel beim Entwurf der Bibliothek war es, sowohl die Anzahl der neu zu erlernenden Bibliotheken zu minimieren, als auch dem Entwickler teilweise die Arbeit abzunehmen, sich näher mit der Architektur der CBEA auseinandersetzen zu müssen.

4.2.2 Einheitliches Interface auf SPE und PPE

Dieser Punkt ist genaugenommen ein Teil der Forderung nach einfacher Bedienbarkeit der CBEA. Die Vereinheitlichung der Schnittstellen nimmt dem Klienten die Arbeit ab, für die selbe Funktionalität unterschiedliche Schnittstellen zu benutzen. Aktuell liegen die Unterschiede für die Programmierung der PPE bzw. SPE im Detail. So ist z.B. beim Abruf einer Nachricht aus der Mailbox auf der SPE-Seite die Funktion

[1] Als Klient wird hier jemand (z.B. ein Entwickler) oder etwas (z.B. eine andere Bibliothek) bezeichnet, der/das den von mir geschriebenen Code verwendet.

`spu_read_in_mbox()` aufzurufen, welche blockiert, bis eine Nachricht in der Mailbox der entsprechenden SPE ankommt [IBM06e]. Dagegen muss auf der PPE die Funktion `spe_read_out_mbox()` aufgerufen werden, welche -1 zurückliefert, wenn keine Nachricht in der Mailbox vorhanden ist [IBM06d].

4.2.3 Ermöglichen effizienter Programme

Die C(ell)++ wurde entworfen, um eine Programmperformance auf dem selben Niveau wie die bestehenden Lösungen zu ermöglichen.

4.3 Vorgehen

Im ersten Schritt zum Entwurf der C(ell)++ wurde ein grobes Interface festgelegt. Dies geschah, indem für unterschiedliche Aufgaben relativ einfach zu verstehender Code auf ein Blatt Papier geschrieben wurde, der die Aufgabe lösen sollte.

Als nächstes wurden so genannte Coding Guidelines, also Richtlinien, an die sich der Code halten soll, festgelegt. Diesen Schritt vor dem eigentlichem Design der Bibliothek durchzuführen, hat den Vorteil, dass z.B. die Namen der verschiedenen Klassen bereits die entsprechenden Vorschriften aus den Coding Guidelines berücksichtigen.

Im letzten Schritt vor der eigentlichen Implementierung wurde ein genaueres Design, basierend auf den grundsätzlichen Designzielen und den Forderungen, die aus den bereits oben erwähnten Interfacedefinitionen resultieren, festgelegt.

4.4 Funktionalität

Die C(ell)++ Funktionalität lasst sich in drei Bereiche aufteilen:

- ein neuer AltiVec-Header, der es ermöglicht, die AltiVec-Funktionalität zu nutzen.
- Vektor-Templates, welche automatisch die SIMD-Einheiten der Architektur ausnutzen bzw. automatisch die Berechnung auf mehrere SPU-Threads aufteilen.
- ein generisches Interface zur Verwaltung und Koordination von SPU-Threads.

4.4.1 AltiVec

Der AltiVec-Standard [AVP99] definiert die folgenden neuen typspezifizierenden Schlüsselworte für C und C++:

- `vector`
- `__vector`
- `pixel`

- `__pixel`
- `bool`

Diese überlagern allerdings teilweise die in C++ neu eingeführten Bezeichner. So existiert z.B. bereits ein Typ `bool` in C++ und auch `vector` wurde innerhalb der STL[2] definiert. Abhängig von der AltiVec-Implementierung im Kompiler kann es sein, dass sowohl `vector` wie auch `pixel` als Makros definiert werden und somit in dem zu kompilierenden Quelltext textuell durch `__vector` bzw. `__pixel` ersetzt werden. Somit wird es z.B. unmöglich, den in der STL definierten Vektor zu benutzen, ohne vorher die vom Kompiler gesetzten Makros wieder zu entfernen.

C(ell)++ löst dieses Problem durch die Benutzung von Typedefs und bietet dies zusammen mit den vom AltiVec-Standard vorgeschriebenen Funktionen innerhalb eines Namensraumes an.

Zusätzlich dazu wird ein Ersatz für die in den aktuell verfügbaren GNU-Kompilern, nicht AltiVec-standardkonforme Speicheralokation der Vektortypen angeboten. Der Standard verlangt, dass durch einen Aufruf von `new` für AltiVec-Typen der zurückgegebene Speicher ein Alignment von 16 Bit hat. Die C(ell)++ ermöglicht dies durch Anbieten spezieller Funktionen, welche sich, abgesehen vom Speicheralignment, genauso verhalten wie die im C++ Standard enthaltenden `new`/`delete`-Operatoren.

4.4.2 Vektoren

Die im AltiVec-Standard definierten Vektoren haben, abgesehen von den oben genannten Problemen, noch weitere Eigenschaften, welche eine intuitive Benutzung erschweren. Die Größe eines Vektors ist immer auf 16 Byte beschränkt. Somit ergibt es sich, je nach Vektortyp (z.B. `short` oder `float`) eine unterschiedliche Anzahl von Elementen, die maximal gespeichert werden können. Ein AltiVec-Vektor vom Typ `float` kann so z.B. nur 4 Werte speichern, wohingegen ein Vektor vom Typ `short` 8 Werte speichern kann. Weiterhin ist es nicht möglich, Vektoren durch einfach Benutzung von Operatoren zu bearbeiten, sondern es müssen hierfür spezielle AltiVec-Funktionen aufgerufen werden. Diese Funktionen decken allerdings nicht den zu erwartenden Funktionsumfang ab. So ist es z.B. nicht direkt möglich, eine Multiplikation zweier Vektoren vom Typ `short` durchzuführen.

In der SPU C/C++ Spracherweiterung werden auch Vektoren für die SPU eingeführt. Diese Vektoren verhalten sich gemäß der oben genannten Eigenschaften analog.

Die C(ell)++ bietet dazu ein Template an, welches diese Probleme löst. So werden für alle Vektoren die Operatoren `+`, `-`, `*` und `/` angeboten. Die Anzahl der Elemente, die in einem Vektor gespeichert werden können, ist nur durch die Menge des verfügbaren Speichers limitiert. Das Interface des Templates ist für PPE- und SPE-Code identisch. Um eine bestmögliche Performance zu bieten, werden bei den Berechnungen, sofern möglich, SIMD-Operationen benutzt.

Unabhängig von dem oben beschriebenen Vektor-Template bietet die C(ell)++ noch ein weiteres an, welches speziell daraufhin optimiert wurde, auf sehr vielen Daten

[2] Standard Template Library

die selben Operationen mit möglichst hoher Performance auszuführen. Aufgrund von Limitationen innerhalb der zugrunde liegenden Bibliotheken ist es nur auf der PPE verfügbar.

Abhängig von der Anzahl der zu berechnenden Daten nutzt das Template nur die AltiVec-Einheiten der PPE oder erzeugt automatisch einen bzw. mehrere SPU-Threads, welche die Berechnungen ausführen. Dieses Template bietet aktuell nicht die optimale Performance, ist jedoch voll funktional. Im folgenden wird dieses Template als "Intelligenter Vektor" bezeichnet. Genauere Informationen hierzu folgen in Abschnitt 4.5.

Zusätzlich zu den eigentlichen Templates wird auch noch ein cast-Operator angeboten, der sich genauso wie der im C++ Standard definierte `static_cast` verhält und Vektoren von einem Typ in einen anderen konvertieren kann.

4.4.3 SPU-Threadverwaltung und -kommunikation

Threadverwaltung und -kommunikation wird in der C(ell)++ mit Hilfe einer zentralen Klasse durchgeführt. Diese Klasse repräsentiert den Thread selbst und es muß eine Instanz sowohl auf der Seite des erzeugenden Threads als auch des neu erzeugten Threads existieren. DMA-Transfers, Kommunikation oder auch Synchronisation erfolgt dann direkt über das Interface der Klasse.

Es werden grundsätzlich zwei unterschiedliche Arten von DMA-Transfers angeboten. Der eine bietet Zugriff auf den Hauptspeicher, der andere auf den lokalen Speicher der SPE. Dies wird zwar in den zugrunde liegenden Bibliotheken von einer Funktion angeboten, erfordert allerdings vom Klienten Wissen über die interne Speicherarchitektur. Diese Aufgabe wird durch die C(ell)++ allerdings vollständig transparent durchgeführt.

Von der C(ell)++ Bibliothek werden zwei Kommunikationsmethoden angeboten. Zum einen gibt es eine generische Funktion, die einen beliebigen 32bit-Wert senden bzw. empfangen kann. Zum anderen eine Funktion zum Übertragen von Adressen, die sowohl im 32- wie auch im 64bit-Modus der PPE korrekt funktioniert. Intern wird diese Kommunikation immer über Mailboxen durchgeführt. Das ist auch der Grund, warum keine Funktion angeboten wird, die mehr als 32 bzw. 64 Bit übertragen kann. Denn wie bereits erwähnt, wurden die Mailboxen nicht daraufhin optimiert, größere Datenmengen zu transportieren.

Bei der Synchronisation werden auch wieder zwei Fälle unterschieden: Synchronisation auf Befehlsebene[3], d.h. es ist sichergestellt, dass alle teilnehmenden Threads einen bestimmten Punkt innerhalb eines Programs erreicht haben. Sowie Synchronisation des Speichers[4], d.h. es wird sichergestellt, dass alle beteiligten Threads die selben Daten im Speicher sehen. Um dem Klienten die Arbeit mit der C(ell)++ zu vereinfachen, zieht eine Befehlssynchronisation automatisch auch eine Synchronisation des Speichers mit sich. Die Befehlssynchronisation wird intern über atomare Operationen auf eine Speicherzelle durchgeführt.

[3] ähnlich z.B. dem OpenMP barrier Konstrukt [Ope05].
[4] ähnlich z.B. dem OpenMP flush Konstrukt [Ope05].

Anz. Threads	Prozessorelement	Skalar/SIMD	Laufzeit (in Sek.)
1	PPE	Skalar	126
1	PPE	SIMD	116
1	SPE	SIMD	84
2	SPE	SIMD	75
4	SPE	SIMD	73
8	SPE	SIMD	73

Tabelle 4.1: Laufzeit des Programm aus Listing 4.1 mit unterschiedlichen Prozessorelementen

4.5 Performanceuntersuchung für Intelligente Vektoren

Das Programm aus Listing 4.1 wurde benutzt, um eine grobe Einschätzung der Performance der Intelligenten Vektoren durchzuführen. Dieses Programm führt auf 3x81.920 Fließkommawerten eine Reihe von Rechenoperationen aus. Durch den Einsatz des Intelligenten Vektors werden diese Daten automatisch auf mehrere SPEs verteilt und dort die Berechnungen mithilfe von SIMD-Operationen ausgeführt. Die Anzahl der beteiligten SPEs wurde für diesen Test variiert. Es wurde beim Einsatz von 4 SPEs ein Speedup von ca. 1,73 im Vergleich zur Benutzung der PPE ohne AltiVec, bzw. ca. 1,59 im Vergleich zu der PPE-Version mit AltiVec-Code erreicht. Wie man Tabelle 4.1 entnehmen kann, skaliert die aktuelle Implementation kaum durch die Hinzunahme weitere SPEs. Nähere Untersuchungen des Problems zeigen, dass eine weitere Steigerung der Performance durch den EIB limitiert wird.

Wie bereits erwähnt, ermöglicht die aktuelle Implementation noch viele weitere Optimierungen. Aktuell werden für jede auszuführende Operation die beiden beteiligten Vektoren zur SPE und der Ergebnisvektor in den Hauptspeicher transportiert, d.h. in dem Beispielprogramm werden die Daten in jeder Schleifeniteration 14-mal zur SPE und 7-mal ein Ergebnisvektor in den Hauptspeicher übertragen. Dieses Problem wäre z.B. durch das von Scott Meyer in [Mey99] vorgeschlagene "Lazy Express Evaluation"-Konzept lösbar. Durch konsequente Anwenden dieses Konzepts wird die Berechnung des Vektors erst durchgeführt, wenn das Ergebnis benötigt wird bzw. einer der beteiligten Vektoren seinen Wert ändert. In der Beispielanwendung würde das z.B. ermöglichen, nicht nach jeder Operation die Daten zurück in den Hauptspeicher schreiben zu müssen, sondern gleich mehrere Operationen auf der SPE auszuführen. So würden deutlich weniger Daten über den EIB transportiert.

4.6 Vorteile gegenüber der bestehenden Lösung

Abgesehen von den oben bereits explizit genannten Beispielen bietet die C(ell)++ noch weitere Vorteile:

Listing 4.1: Beispielprogramm für Intelligente Vektoren

```
ivec<float, 10240*8> a(1);
ivec<float, 10240*8> b(1);
ivec<float, 10240*8> c(23);

for (int i=0; i<5000; ++i) {
        a = (b+a) * (c-a) * a * c * c + c;
}
```

4.6.1 Zentrale Fehlerbehandlung für die Threadverwaltung

Die libSPE-Funktionen zur Threadverwaltung liefern im Fehlerfall einen Fehlercode zurück und speichern in `errno` nähere Informationen zu dem Fehler. Die C(ell)++ ermöglicht eine zentralisierte Fehlerbehandlung, indem man jedem Thread eine Funktion zuweisen kann, welche im Fehlerfall mit einem bestimmten Parameter aufgerufen wird. Diese Funktion kann dann z.B. versuchen, den Fehler zu beheben und die fehlerhafte Operation erneut ausführen oder den Thread beenden.

4.6.2 Festlegung des Vektortyps durch ein Template Argument

Aktuellen ist es nicht möglich, den Typ eines Vektors über ein Template Argument zu bestimmen, d.h. Code der Art

```
template <typename T>
function foo (T _skalar, vector T _vector);
```

ist nicht einsetzbar. Das in der C(ell)++ enthaltene Vektor Template kann allerdings problemlos dafür eingesetzt werden:

```
template <typename T>
function foo (T _skalar, vector<T,
 vec_specialization<T>::factor> _vector);
```

Der Einsatz des Templates mit den hier gezeigten Parametern erzeugt auch keinen Overhead im Vergleich zu dem direkten Einsatz des Altivec `vector`-Schlüsselwortes.

4.6.3 SPE als Kontrollinstanz

Wie bereits in Abschnitt 4.4.3 erwähnt, verfügt die C(ell)++ Klasse zur Threadverwaltung über eine Möglichkeit, relativ einfach auf Daten im lokalen Speicher einer SPU zuzugreifen. Zusätzlich dazu ermöglicht diese Klasse auch, die Kontrolle über einen SPU-Thread an einen anderen Thread zu transferieren. Dieser Thread könnte dann als zentrale Kontrollinstanz eingesetzt werden, um weitere Threads zu steuern. Es ist aber auch denkbar, dass ein SPU-Thread den lokalen Speicher eines anderen

als Zwischenspeicher nutzt, da der Zugriff auf den lokalen Speicher einer anderen SPU schneller ist als der Datentransfer aus dem Hauptspeicher.

4.7 Probleme bei der Entwicklung

Während der Entwicklung gab es verschiedene Probleme. Ein grundlegendes Problem entstand dadurch, dass sich die eingesetzten Bibliotheken selbst noch in Entwicklung befinden. So waren z.B. verschiedene Funktionen dieser Bibliotheken nicht in der `extern "C"` Speicherklasse definiert, so dass das Linken mit einem C++ kompilierten Programm nur über Umwege möglich war. Weiterhin wird auf der SPE keine vollständige C++ Implementierung angeboten. Im Rahmen der Entwicklung wurde die Funktionalität des `iostream`-Headers vermisst.

4.8 Ergebnis

Die C(ell)++ erfüllt die meisten der in Abschnitt 4.2 gestellten Anforderungen. Das Interface der Bibliothek wurde so weit wie möglich einfach gehalten und es unterscheidet sich nur minimal bei Verwendung von SPE- und PPE-Code. Ob das Ziel einer einfach zu benutzenden Bibliothek erreicht wurde, konnte bisher noch nicht in der Praxis getestet werden. Eine vollständig verfügbare Dokumentation sollte den Einstieg jedoch relativ einfach ermöglichen. Die endgültige Performance der Bibliothek ist ohne größere Anwendungen nur schwer abzuschätzen. Kleinere Testprogramme lassen allerdings vermuten, dass die Performance identisch zu den bereits bestehenden Bibliotheken ist.

5 Fazit

Das Praktikum im Forschungs- und Entwicklungszentrum der IBM in Böblingen hat meine Erwartungen voll erfüllt. Ich habe während der 3-monatigen Praktikumszeit die Möglichkeit genutzt, einen intensiven Einblick in den Aufbau und die Arbeitabläufe eines großen international agierenden Unternehmens zu erhalten. Es war sehr interessant zu sehen, wie die unterschiedlichen Abteilungen innerhalb der Firma miteinander interagieren. Auch aus der Teamarbeit innerhalb meiner Abteilung konnte ich für mein Projekt stets positive Impulse gewinnen. Als besondere Herausforderung habe ich die völlig eigenständige Planung meines Projektes empfunden, wobei ich nicht nur den sich über mehrere Monate erstreckenden Projektplan, sondern auch die täglichen und wöchentlichen Meilensteine selbständig erarbeiten durfte.

Alles in allem sehe ich mein Praktikum als einen wichtigen Teil meiner Ausbildung an und die dort gesammelten Erfahrungen werden mir im weiteren Studium sowie bei der anschließenden Arbeitsplatzwahl sehr hilfreich sein.

Literaturverzeichnis

[AVP99] *AltiVec Technology Programming Interface Manual*, rev. 0 edition, Juni 1999.

[IBM] Das IBM Entwicklungszentrum in Böblingen. `http://www-5.ibm.com/de/entwicklung/ueberuns/das_entwicklungszentrum.pdf`.

[IBM05] IBM Annual Report, 2005.

[IBM06a] IBM. *Cell Broadband Engine Programming Handbook*, version 1.0 edition, April 2006.

[IBM06b] IBM. *Cell Broadband Engine Programming Tutorial*, version 1.1 edition, Juni 2006.

[IBM06c] IBM. *Cell Broadband Engine SDK Libraries Overview and Users Guide*, version 1.1 (public sdk) edition, Juni 2006.

[IBM06d] IBM. *SPE Runtime Management Library*, version 1.1 edition, July 2006.

[IBM06e] IBM. *SPU C/C++ Language Extensions*, version 2.1 edition, März 2006.

[Mey99] Scott Meyers. *Mehr Effektiv C++ programmieren*. Addison-Wesley, 1999.

[Ope05] OpenMP Application Program Interface, Mai 2005. Version 2.5.

[VS05] Madhavan Srinivasan Vaidyanathan Srinivasan, Anand K. Santhanam. Cell broadband engine processor dma engines, part 1: The little engines that move data. Technical report, IBM India Pvt Ltd., Dezember 2005.