ABSTRACT

Databases are the main component of modern systems. We may find Database Management Systems (DBMS) in the backend of many software systems, such as Enterprise Resource Systems (ERPs), websites, accounting systems, scientific applications and many others. DBMS systems are responsible for storing, analyzing, managing and querying large volumes of data. Querying of data forms is the most time-consuming workload of DBMS systems. The querying comprises of different operations, mostly aggregations, selections and join operations. The most known queries and also the most time-consuming are the Decision Support System (DSS) queries. DSS queries are designed to process large data sets and use large-scale multiprocessors.

For this work we decided to analyze the benefits of accelerating DSS queries on different GPU and the CELL processors. Both architectures, aside from promising high-performance, are of low-cost. We use Rapidmind to implement the parallel version of three representative DSS queries found in the TPC-H Benchmark. As a baseline for our comparisons we use a single-core native C++ code execution on a Core 2 Duo 2.13GHz CPU. We compare the execution of the three queries with two different GPU models of the same generation (NVidia Geforce 8500GT and 8800GT) and with the CELL processor found on Playstation 3.

Results show that GPUs are very promising as far as exploiting parallelism in database queries. The speedup observed was up to 21x compared to the single processor CPU execution.

Exploiting the use of GPUs and CELL Broadband Engine in Parallel Acceleration of Decision Support Queries using Rapidmind Data-Parallel Platform

Despo Othonos

A Thesis

Submitted in Partial Fulfilment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

By the Department of Computer Science

June, 2009

APPROVAL PAGE

Master of Science Thesis

Exploiting the use of GPUs and CELL Broadband Engine in Parallel Acceleration of Decision Support Queries using Rapidmind Data-Parallel Platform

Presented by

Despo Othonos

Research Supervisor ______________________
Petro Trancoso

Committee Member ______________________
Yiorgos Chrysanthou

Committee Member ______________________
Chryssis Georgiou

University of Cyprus

June, 2009

ACKNOWLEDGEMENTS

I would like to thank my Supervising Professor, Dr. Pedro Trancoso for his valuable and direct guidance through the development of this research work and the preparation of the Thesis. He helped me to develop understand the challenges I had to face in this research work and to develop certain ways of thinking for facing these challenges.

I thank Dr. Chryssis Georgiou and Dr. Yiorgos Chrysanthou that were in the evaluation committee and contributed with their comments and suggestions regarding my dissertation.

Furthermore, I would like to thank the CASPER Group members, and especially Panayiotis Petrides and Frederico Pratas who provided me technical assistantship for better understanding of my experimental results, as well as important knowledge.

In addition, I want to say a big thank you to my friends and especially Artemakis Artemiou – former CASPER member – for being there for me in every important step of my life.

Finally, my gratitude and warm appreciation goes to my husband Marios Soteriou and my family who emotionally and practically supported me during this difficult period of my life and giving me important advices.

Table of Contents

Chapter 1: Introduction 1

1.1 Motivation 2

1.2 Main Contributions 3

1.3 Organization 4

Chapter 2: Related work 5

2.1 General-Purpose Operations Using Graphics Hardware and CELL 6

2.2 Database Operations Using Graphics Hardware and CELL 8

2.3 CUDA programming model 10

Chapter 3: Architectures and Programming Platforms **.. 12**

3.1 Multicore processors and Parallel Programming 12

3.2 CPU Architecture 13

3.3 GPU Architecture 17

3.4 CELL Architecture 20

3.5 Stream Computing 25

3.6 The Porting Process 27

Chapter 4: Rapidmind 29

4.1 Rapidmind API 31

4.2 Rapidmind processor support 32

4.3 Parallelizing an Application 33

4.4 How to write a parallel program 35

4.5 Rapidmind platform modes 40

4.6 Rapidmind programming mode 40

Chapter 5: Database Operations on CPU, GPU and CELL 43

5.1 Implementation 43

5.2 TPC-H Benchmarks 44

5.3 TPC-H Q3: Shipping Priority Query 46

5.4 TPC-H Q6: Forecasting Revenue Change Query 51

5.5 TPC-H Q12: Shipping Modes and Order Priority Query 55

Chapter 6: Experimental Results 60

6.1 Experimental Setup 60

6.2 Selection of float types in Rapidmind 65

6.3 Results on GPUs 71

6.4 Results on CELL 79

6.5 Comparison 84

Chapter 7: Conclusions and Future Work 90

7.1 Conclusions 90

7.2 Future Work 93

REFERENCES 95

LIST OF TABLES

Table 1: Scale Factor and Data Table Sizes 61

Table 2: Scale Factor with In and Out Data Sizes for each Query 62

Table 3: NVIDIA GeForce 8500 and 8800 Specifications 64

Table 4: CELL Specifications 64

LIST OF FIGURES

Figure 1: CUDA Software Stack 11

Figure 2: Subscalar CPU model 13

Figure 3: Basic Five-step Pipeline 13

Figure 4: Simple superscalar pipeline 13

Figure 5: Basic unified GPU architecture 18

Figure 6: Overview of the CELL architecture 21

Figure 7: SISD, SIMD, and stream processing compared 26

Figure 8: The RapidMind architecture 30

Figure 9: How a program is created (in RM) and then can be used to operate as an input array A to generate output array B 34

Figure 10: Parallel processing of program in Figure 9. 29

Figure 11: C++ example code 29

Figure 12: Translation of C++ defines and declarations into Rapidmind 29

Figure 13: The example of C++ code in Figure 11, translated into Rapidmind code 29

Figure 14: TPC-H Q3 in SQL code 43

Figure 15: Our TPC-H Q3 implementation in SQL code 43

Figure 16: Sample parts from Rapidmind code for Q3 43

Figure 17: TPC-H Q6 in SQL code 43

Figure 18: Our TPC-H Q6 in SQL code 43

Figure 19: Sample parts from Rapidmind code for Q6 43

Figure 20: TPC-H Q12 in SQL code 43

Figure 21: Our TPC-H Q12 in SQL code 43

Figure 22: Sample parts from Rapidmind code for Q12 43

Figure 23 (a) Q3 execution on GPU GeForce 8500 with different float types 66

Figure 23 (b) Q6 execution on GPU GeForce 8500 with different float types 66

Figure 23 (c) Q12 execution on GPU GeForce 8500 with different float types 67

Figure 24(a) Q3 execution on CELL SPEs with different float types............................. 69

Figure 24(b) Q6 execution on CELL SPEs with different float types............................. 69

Figure 24(c) Q12 execution on CELL SPEs with different float types........................... 70

Figure 25(a): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q3 query ... 71

Figure 25(b): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q3 query ... 72

Figure 26: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q3 query 73

Figure 27: Execution time of CPU, 8500GT GPU and 8800GT GPU for Q6 query........ 74

Figure 28: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q6 query 76

Figure 29(a): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q12 query . 77

Figure 29(b): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q12 query . 77

Figure 30: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q12 query 78

Figure 31: Speedup given from Rapidmind execution on CELL SPE processors, for Q3 query, using as a baseline the native C++ execution time on CPU 80

Figure 32: Speedup given from Rapidmind execution on CELL SPE processors, for Q6 query, using as a baseline the native C++ execution time on CPU 81

Figure 33: Speedup given from Rapidmind execution on CELL SPE processors, for Q12 query, using as a baseline the native C++ execution time on CPU 81

Figure 34: Speedup given from Rapidmind execution on CELL SPE processors, for Q3 query, using as a baseline the native C++ execution time on CELL PPE processor 82

Figure 35: Speedup given from Rapidmind execution on CELL SPE processors, for Q6 query, using as a baseline the native C++ execution time on CELL PPE processor 83

Figure 36: Speedup given from Rapidmind execution on CELL SPE processors, for Q12 query, using as a baseline the native C++ execution time on CELL PPE processor 83

Figure 37(a): All speedups for Q3 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution 85

Figure 37(b): Speedups for Q3 query given CELL PPE and CELL SPE executions, using as a baseline the CPU execution 86

Figure 38: All speedups for Q6 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution 87

Figure 39(a): All speedups for Q12 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution 88

Figure 39(b): Speedups for Q12 query given CELL PPE and CELL SPE executions, using as a baseline the CPU execution 89

Chapter 1

Introduction

The most important part of modern software systems is their database. There is a Database Management System (DBMS) in the backend of many software systems like Enterprise Resource Systems (ERPs) [47], websites and web-applications [46], accounting systems, scientific applications, gaming software and many others. Being characterized as the most time-consuming part of an application, DBMS are responsible for storing, managing, sorting and querying data. The querying and sorting of data is what actually causes the delays and bottlenecks and classifies the DBMS systems as time-consuming systems [45]. In this project we will deal only with the querying of data, which includes predicates evaluation, aggregations, as well as join operations.

Researchers have been studying the development of optimized algorithms for efficient database operations. Some of these algorithms are already build-in modern DBMSs. DBMS systems are often characterized as I/O bound or data-intensive applications [45, 50]. In most cases, DBMSs operate in large volumes of data. One possible aspect for improving the cost of an overall DBMS operation execution (i.e. data querying) is by using parallelism. The greatest capability of parallel systems is the concurrent execution of multiple tasks. Graphics Processor Units (GPU) currently offer this capability not only for gaming but for general-purpose computations and application development. Also, the Playstation 3 with a CELL processor allows the developers to take advantage of six of the eight SPE processors in the CELL for developing parallel applications.

Multi-core processors are designed to offer parallelism and high performance. In theory it seems easy to understand their functionality and looks like they have a good potential in parallel programming. But, in reality, it is not so easy to get everything to work in parallel [10].

1.1 Motivation

For multi-core processor systems, its hardware is "ready" to run parallel applications, but applications must be re-written in such a way that they could run in parallel. Non-parallelized applications will not run faster on multi-core systems. On the contrary, their performance might get worse than in single-core systems, since each core, of a multi-core system, is often slower than an existing single-core processor [10].

Parallel software developing is challenging and also hard to debug, even for the most experienced developers. A developer needs to change the whole programming philosophy and way of thinking that he built with all the years of experience in sequential and object-oriented programming, in order to start writing parallelized code.

The parallel hardware exists. We need to take advantage of it and its power. In this project we are going to experiment with two main multi-core architectures. The first is Graphics Processing Unit (GPU) found in the NVIDIA graphics hardware. The second is the CELL processor unit found in the Sony's Playstation 3. GPUs are proven to have high speedups in general purpose applications [16, 22, 33, 34]. The high computational power of GPUs and the large number of processors of the CELL, has motivated us to test their execution on database queries, since they are very time-consuming and also a lot demand a lot of processing power.

Another factor that motivated us to look for other methods and hardware for executing time-consuming database queries is the cost. The price of GPUs or Playstation 3 is relatively low compared to powerful CPUs or other multicore architectures.

1.2 Main Contributions

In this project we use Rapidmind development platform and write code that runs in parallel. We get our results by testing our execution code in different multi-core processors such as GPUs and CELL. Also, we want to evaluate the use of Rapidmind platform, which allows single applications to be executed on different parallel hardware.

Also, in this thesis, we focus on another major factor in software applications; the rather demanding database operations. We use Rapidmind to implement TPC-H Queries on real TPC-H data. We implement Q3, Q6 and Q12 queries, that consist of many and different operations, such as joins, Boolean combinations, sum aggregations, etc. We test their performance on a Core 2 Duo 2.13GHz CPU (using the one of the two cores), on two different GPUS, NVIDIA GeForce 8500 and 8800, and on the CELL processor which is included in the Playstation 3.

Rapidmind development platform is a tool that allows single threaded applications to fully access multi-core processors [11]. With the use of Rapidmind, developers can still write in standard C++ language, as they are used too. They can use Rapidmind's types and extensions where they want to have optimized and parallelized code. Rapidmind will do the rest and "spread" the execution across the available cores. With the use of Rapidmind in software developing, it allows the software companies to release high-performance applications while minimizing the risk of unstable solutions and failures, since the coding language is more-or-less known. The knowledge also minimizes the development costs.

1.3 Organization

In the next Chapter we will present some of the related work. Next, in Chapter 3 we refer to the different related architectures. Chapter 4 is about Rapidmind platform architecture. Chapter 5 describes the TPC-H database queries that we will use, the implementation using Rapidmind platform and how the porting process is done in this project. Chapter 6 gives our results and finally in Chapter 7 we present the conclusions of the project.

Chapter 2

Related Work

Developers showed a lot of interest in exploiting the power of multi-cores by taking advantage the parallel resources they offer. Two known multicores that we are going to exploit here are the GPU and the CELL. Both GPUs and CELL offer streaming capabilities [5, 8, 9, 12, 13, 14, 17, 26, 39] and programmability [3, 4, 19, 24, 28]. Several studies were done for using GPUs for general purpose applications and database management operations [2, 6, 16, 22, 25, 33, 34].

Lots of work has been done on exploiting the GPU processor, its capabilities and limitations. It’s being compared against modern CPUs in executing general purpose computations. Recent studies scientists have started experimenting on database queries execution on GPUs. Scientific floating point operations and database queries are known time consuming applications. Less work has been done to the newly released CELL processor.

2.1 General-Purpose Operations Using Graphics Hardware and CELL

On early stages of programmable GPUs, work has been done from Trancoso and Charalambous [22], in order to show the limitations we have when using then for general purpose applications. With the use of BrookGPU [27], they test different ways of sending data to GPU and present the best, which achieves higher speedup up to 8.1x using an NVIDIA GeForce FX 5700LE GPU versus an Intel Pentium 4 3.2GHz CPU. In their research they also stated that there are cases where it is best to upgrade a computer with a new high-end GPU than replacing it with a new one. When doing the upgrade of the GPU instead of the CPU, in order to execute our applications on the GPU, it is inevitable that we need to design and port the general-purpose applications on the graphics processor.

Rapidmind presented their Platform in November of 2006 [10], shown the use of the three main types of the programming model – values, arrays and program. They have also shown the use of Rapidmind and ways of development in GPUs, as well as in CELL.

McCool et. Al, in 2007 [43], used Rapidmind platform to implement a Parallel Set Intersection for Keyword search. They perform experimental comparison of performance on web indexes and queries provided by Google. Results showed improvement of performance, in respect to the level of parallelism, when using up to four processors.

Several researchers have used systems with CELL Broadband Engine for porting different applications, trying to improve their performance. Williams et. al. [57] showed the potential of CELL processor in scientific computing. CELL also proved that it can be used for medical reasons. Servat et. al. [57] used a blade system with two 3.2GHz CELL Broadband Engine processors for evaluating the performance of a well known protein docking application in the Bioinformatics field, Fourier Transform Docking (FTDock). CELL has demonstrated impressive performance in several applications and computational kernels with high vectorizable data parallelism, such as signal processing, compression, encryption, dense and sparse numerical kernels [59, 60, 61].

Rapidmind Development Platform is one of the tools that exist today for developing parallel applications [2, 10, 11]. That's because of its simplicity and easy way of deploying parallel applications. Rapidmind is like an embedded programming language within C++. The use of Rapidmind in both GPU and CELL is studied [10]. You can use Rapidmind code to run a program either on the GPU or on the CELL processor, with some minor differences. It's being specified efficiently onto the capabilities of both processor types. A program written in Rapidmind can scale automatically according to the available number of cores, and also can be extended to other multicore processors in the future.

Different environments exist for stream computing, like Rapidmind and CUDA. In this chapter, we will give some information about CUDA, an NVIDIA proprietary programming model.

2.2 Database Operations Using Graphics Hardware and CELL

Database Management Systems' purpose is to handle large amount of data. Hence, most of database operations are executed against this large amount of data. Although most of these operations are not complex, they are considered as very time-consuming. Lots of effort is given nowdays for optimizing traditional database algorithms [2, 39, 40, 41, 42].

Research in executing Database Operations on GPUs has been done before using several development platforms, including the Cg compiler [6]. Artemiou in [2] used Rapidmind development platform and has shown very good speedup results on executing several database operations, two TPC-H Queries and join operations on an NVIDIA GeForce 8500 GPU against an Intel Core 2 Duo CPU.

An important part of a DBMS system and a considerable factor in its performance are the sorting algorithms. But, because sorting is both computation and memory-intensive, many algorithms were proposed from time to time [39, 40, 41, 42] trying to reduce their cost. Though these algorithms were based on CPU execution and therefore sequential execution, after the development of graphics processors into programmable graphics processor units research focused on investigating their compute power. Govindaraju et. al [33] presented an algorithm used for data mining and executed on the GPU. The algorithm included equi-join and non-equi-join queries as well as static queries on data streams. GPU execution on NVIDIA GeForce 6800 Ultra showed a 1.3x speedup against the execution on an Intel Pentium 4 with 3.4GHz CPU.

Other algorithms for sorting using the GPU processor were later presented by many researches, including again Govindaraju et. al [34] who proposed the GPUTeraSort algorithm. This algorithm was used for sorting billions of records using the GPU processor unit. They compare its execution with the nsort [35] CPU-based algorithm. For CPU they used a high-end dual Xeon processor and for GPU three different NVIDIA GeForce processors (6800, 6800 Ultra and 7800GT). Results were competitive, considering the fact that CPU was a high-end processor and very expensive and the GPUs were cheap mid-range processors.

There is also a study related to our work was done from Gedik et. al. [58], which deals with join operations on the CELL.

The work presented in this Thesis is a continuation of Artemiou's [2] work on Database Operations and DSS Queries, using Rapidmind platform. I'm focusing on the three TPC-H Benchmarks (Q3, Q6 and Q12) using real TPC-H Data exported by DBGEN engine instead of synthetic data. I am executing these queries on different GPUs (NVIDIA Geforce 8500 and 8800) and also on the CELL processor.

2.3 CUDA programming model

An alternative to Rapidmind platform is the CUDA programming model. Although we will not do any work in this project using CUDA, it is a new and very promising model and it is imperative to say a few words about its architecture.

Compute Unified Device Architecture (CUDA) is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [26]. CUDA can be used for any GPU since the GeForce 8 Series Graphics Processors, the Tesla solutions, and some Quadro. The operating system's multitasking mechanism is responsible for managing the access to the GPU by several CUDA and graphics applications running concurrently.

CUDA is an NVIDIA graphics hardware proprietary programming model. It can be used either on Linux or Windows, for performing general purpose computing. Using the NVIDIA GPU, the CUDA Toolkit accelerates SIMD parallel jobs [15].

The benefit from using CUDA is that the parallel jobs are "self-contained". That means they can be executed and completed as a batch of GPU threads completely without any interference by the initial (host) process.

Jobs created by the host process, in CUDA, are in the form of Remote Procedure Calling (RPC). Code is C language based, with extra features and functions. CUDA applications basically consist of two main parts. The “host” code, which is C++ code, and the GPU functions. In order to compile a CUDA application, two different compilers collaborate; a general purpose C/C++ compiler for the C++ code and an NVIDIA compiler/assembler for the GPU code. After the compiling, it comes the linking stage, were some specific CUDA libraries are loaded. These libraries support remote SIMD procedure calling and allow the applications to run.

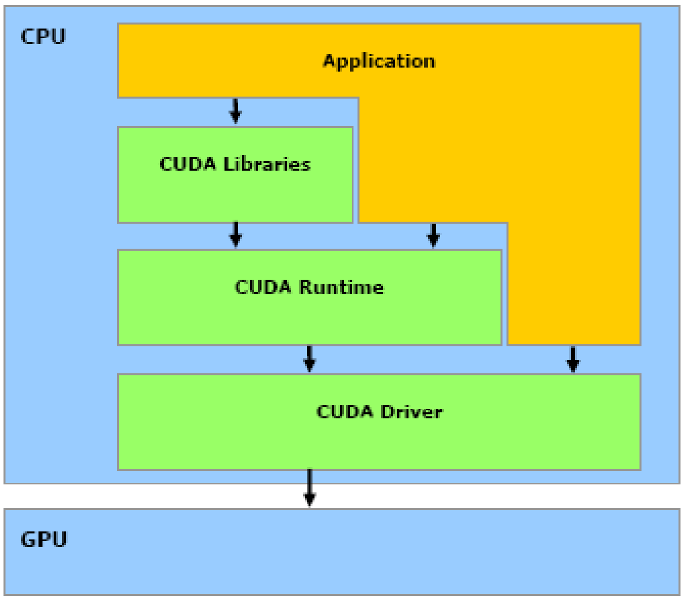


Figure 1: CUDA Software Stack [15]

As shown in Figure 3, the CUDA software stack is composed by several: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance [15].

Chapter 3

Architectures and Programming Platforms

In this chapter I will describe the hardware architectures used in this research: CPU, GPU, and CELL. In addition I will talk about the meaning of stream computing and its benefits. Also, i explain the porting process of the execution of a parallel program into a parallel hardware.

3.1 Multicore processors and Parallel Programming

Several multicore-processor architectures exist in the area of CPUs and GPUs. While some of the desktops today can even include processors with up to 4 processor cores, soon we will have more. AMD is planning to release the Bulldozen chip with 16 cores in 2011 [64]. Also, high-end graphics accelerator cards can include up to 800 cores in one GPU processor (i.e. ATI Radeon HD 4000 Series).

Unfortunately, parallel processors do not lead to higher application performance, especially when a large number of cores are considered [10]. Existing application codes may not expose sufficient parallelism to scale well to larger numbers of cores. In order to take advantage of parallel processing, we need to obtain an important amount of programming effort and skills.

Parallel programming has a different philosophy and, in many cases, is more difficult than sequential programming. We have a pool of different algorithms, software architectures, languages, development tools, and programming techniques to choose from. We have to choose what is best and suitable in association with performance, robustness and portability. If we want to have portability, we have to give emphasis to the level of abstraction of our code.

Although multicore processors have been in the market for some time, parallel programming is not used as much as it should. In early stages of multicore processors we didn't have any really good debuggers [44] or compilers [49] to use. Now we have many platforms that allow us to develop parallel applications: Rapidmind [30], CUDA [26], Accelerator [65], Brook [27] are some of them.

3.2 CPU Architecture

We call a processor or Central Processing Unit (CPU) an electronic circuit that can execute sequences of stored instructions called computer programs.

The simplest type of CPU, called subscalar (Figure 2), can on and execute one instruction on one or two pieces of data at a time. Since only one instruction is executed at a time, the entire CPU must wait for that instruction to complete before proceeding to the next one. This might result in “hanging” of the CPU, on instructions that take more than one cycle to complete execution.

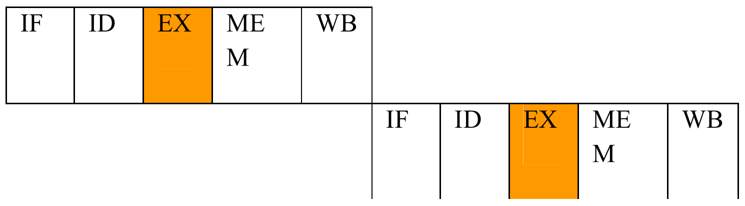


Figure 2: Subscalar CPU model

There are three methods which accomplish parallelism. The first and simplest one is Instruction-level parallelism, also known as Instruction Pipelining (Figure 3, 4). It allows instructions to begin the first steps of instruction fetching and decoding before the prior instruction finishes executing. That way more than one instruction is executed at any given time. However, pipelining faces the issue of date dependency when the result of previous operation is needed to complete the next one.

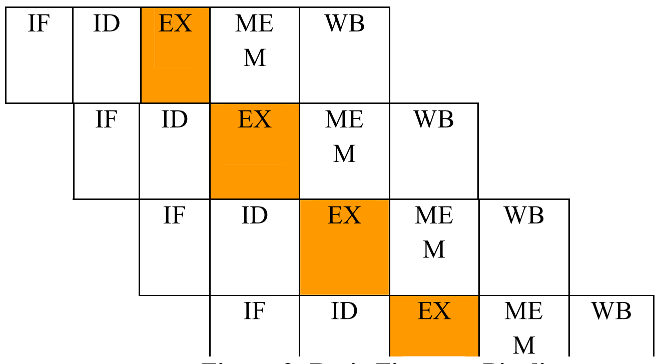


Figure 3: Basic Five-step Pipeline

The second method for parallelism is called Threat-level parallelism. Multiple programs or threats are executed in parallel. This strategy is also known as parallel computing or Multiple Instructions – Multiple Data (MIMD). Multiprocessing (MP) and Multi-threading are techniques of this parallelism category. Last year emphasis was given in developing of dual and multiple core CMP (chip-level multiprocessing) designs, since the CPU designers consider that the throughput computing (the aggregate performance of multiple programs) is more important than the performance of a single thread or program [62]. We can find the CMP in x86 or x64 Opteron, Athlon, 64 X2, SPARC, Ultra SPARC T1 [66], IBM POWER4 [67], IBM POWER5 [68], etc. Also we can find it in many video game console CPUs like Xbox 360's triple-core PowerPC and Playstation's 3 8-core CELL microprocessor (SPEs).

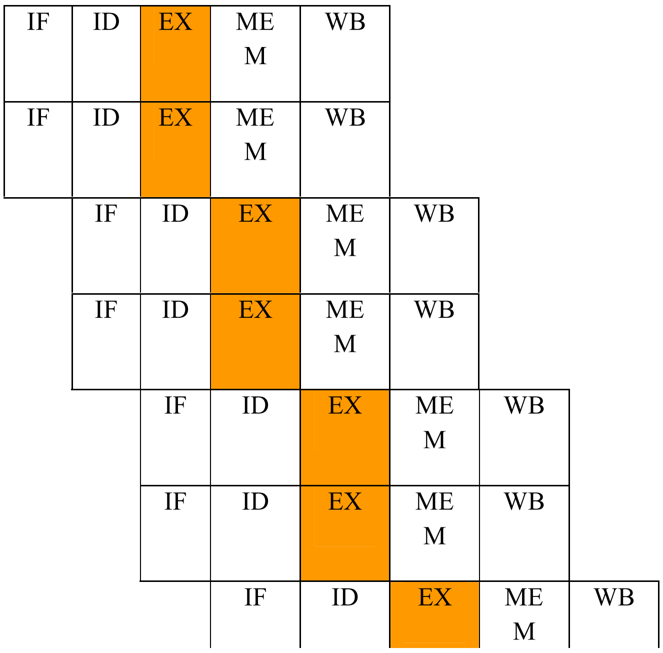


Figure 4: Simple superscalar pipeline

The third important method of parallelism, but less common in CPU designs, has to do with data parallelism. This method implies the need of vector processors that deal with multiple pieces of data for each instruction. For instruction-level and thread-level parallelism, several types of scalar processors are used, which deal with one piece of data for every instruction. The scheme from Flynn's taxonomy [51] that represents data parallelism is Single Instruction – Multiple Data (SIMD). CPUs dealing with vector processing tend to optimize tasks which require the same operation to be performed on large data sets. Examples of such tasks are multimedia applications as well as some scientific and engineering tasks. Some known vector CPUs are the Cray-1, Intel's SSE and PowerPC-related Altivec [70].

After giving a description of how the CPU works and the different types of parallelism it can perform, we need to say a few words about CPU cache and how it is used by the central processing unit in order reduce the average time needed for accessing the memory. Cache memory is smaller and faster than main memory and stores copies of data from the most frequently used main memory locations. Latency from cache memory access is a lot less that main memory access.

When a processor needs to read from or write to a location in main memory, it checks first whether there is a copy of that data in the cache. If there is a copy there, the processor uses the cache to read from or write to [63]. Modern CPUs have at least three independent caches. An instruction cache used for instruction fetch speedup, a Data cache used for data fetch and store speedup, and a Translation Lookaside Buffer for speedup the process of virtual-to-physical address translation for both instructions and data.

3.3 GPU Architecture

GPU architecture has two major benefits in comparison to the CPU architecture. It has much simpler architecture than CPU and also offers a lot of parallelism in exchange of a relatively low cost. Its operation is more or less similar to the SIMD model. SIMD model is referenced below in the Stream Computing paragraph. GPU is designed to execute specific code very fast; code that includes many floating point operations. On the other hand, CPU is designed to execute general-purpose code, which includes a lot of different operations and types; that’s why it is more complicated in terms of architecture.

So far, the GPU architecture was composed of vertex and pixel processors. This architecture did not encourage the general-purpose applications execution on GPUs [56]. When the interest on GPGPUs grew, the architecture of GPUs changed. The vertex and pixel processors are now unified processors, called Stream Processors [14]. Stream Processors are also called Floating point processors, since we can only execute floating point operations on them. All the pixel and vertex operations, as well as geometry and physics operations, are now executed on the stream processors.

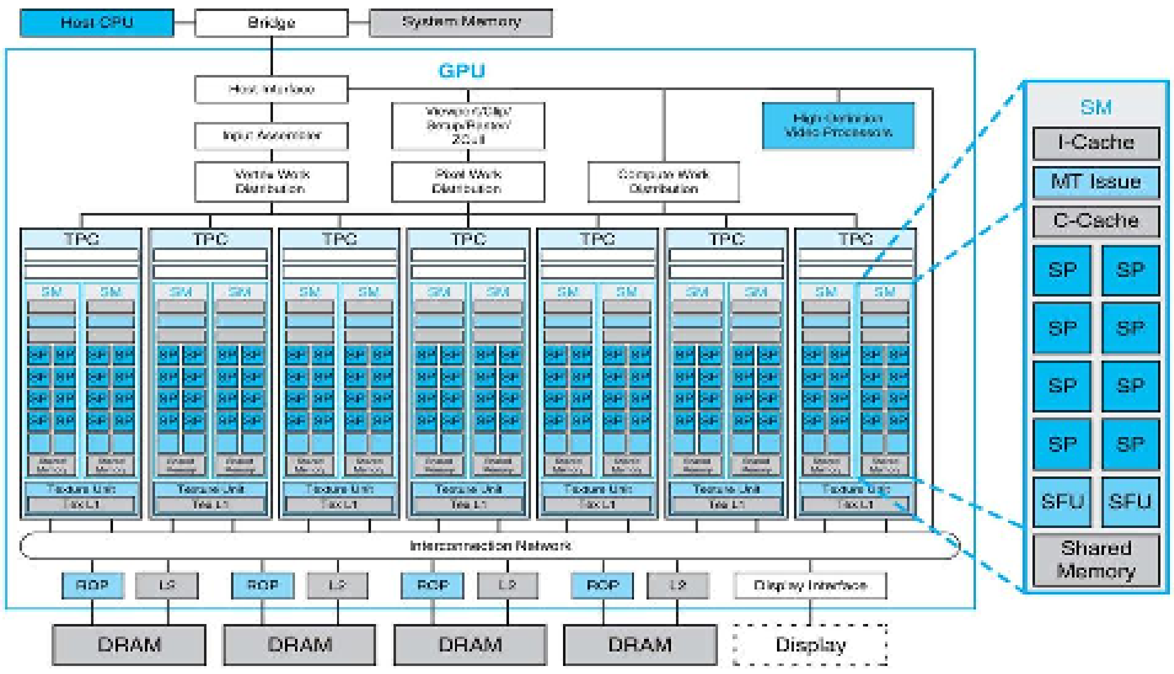


Figure 5: Basic unified GPU architecture
(Copyright © 2009 Elsevier, Inc. All rights reserved)

In Figure 5 we can view an example of GPU architecture with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs). The figure represents the basic architecture of an NVIDIA GeForce 8800. Each streaming multiprocessor has eight multithreaded SP cores, two Special Function Units (SFUs), an instruction cache, a constant cache, a multithreaded instruction unit (MT Issue) and a shared memory. All the processors connect with a total of four 64-bit DRAM partitions via an interconnection network.

Like in a CPU, a Graphics processor with a graphics cache memory unit is coupled to the system memory bus as a peer. The GPU and the CPU have the same priority in accessing the main memory. The graphics processor and the graphics cache unit store the retrieved input data from the main memory in a high-speed memory in the graphics cache unit. Data that represent a three-dimensional array are stored in the high-speed memory in the graphics cache unit in spatially contiguous blocks.

Stream processors on the graphics processing unit, are highly efficient computing engines that perform calculations on an input stream, while producing an output stream that can be used by other stream processors. High-speed instructions decode and execution logic is build into a stream processor and similar operations are performed on the different elements of a data stream. On-chip memory is mostly used for storing the output of a stream processor, which can be quickly read as input by other stream processors for subsequent processing. Groups of stream processors can efficiently execute SIMD instructions.

GPUs are designed for graphical operations. This poses some limitations that we have to consider when we want to use them for general-purpose applications. One critical limitation is the data transfer between CPU and GPU. Although the CPU access to the GPU is more complex than accessing the system's main memory, this is not actually the biggest problem. Limitation comes to the fact that these accesses are mainly one-way operations, when it comes to graphic applications. Data sent to GPU are destined to be displayed on the screen monitor. So, there is no need for data to be sent back to the GPU. For general-purpose computing, the two-way communication is needed. The "send data back to CPU" operation is known as Host Readback (HRB).

Another thing we should have in mind, as far as it concerns the overheads, is the interface design types of the communications between CPU and GPU. Older graphics card were mainly Peripheral Component Interconnect (PCI). This interface was common for many different interconnections, like modems, Ethernets, etc. After PCI, Accelerated Graphics Port (AGP) was released for graphic cards interconnections. Currently we have the PCI-Express interfaces, which offers a data transfer bandwidth of 8GB/s, in contrast to 0.1GB/s of PCI and 2.1GB/s of AGP. Also, PCI-Express technology offers two-way communication, with the same bandwidth, so HRB will not cause us any major overhead.

A new product coming to the market, probably in 2009, is the AMD Fusion [53]. It will certainly be an evolutionary product for GPGPU, since it integrates the CPU and GPU into the same chip. This way, the access and transfer overheads from one Processing Unit to the other will be minimal. Data will be easily shared among the CPU and GPU as they may be accessing a common cache within the chip.

3.4 CELL Architecture

CELL was first introduced by a partnership of Sony, Toshiba, and IBM (STI) to be the processing component of Sony’s PlayStation3 gaming system. CELL has not a common multiprocessor or multi-core architecture. Although it is a multi-core processor, is does not consist of multiple identical processors. It has a total of nine cores, from which one is a PowerPC core, called PPE, and the rest are short vector SIMD processors, called Synergistic Processing Elements (SPEs). PPE is mainly designed for high performance general purpose computations. SPEs, on the other hand, are specially designed for high performance numerical computations. Figure 6 shows an overview of CELL’s architecture [4, 5, 10, 24].

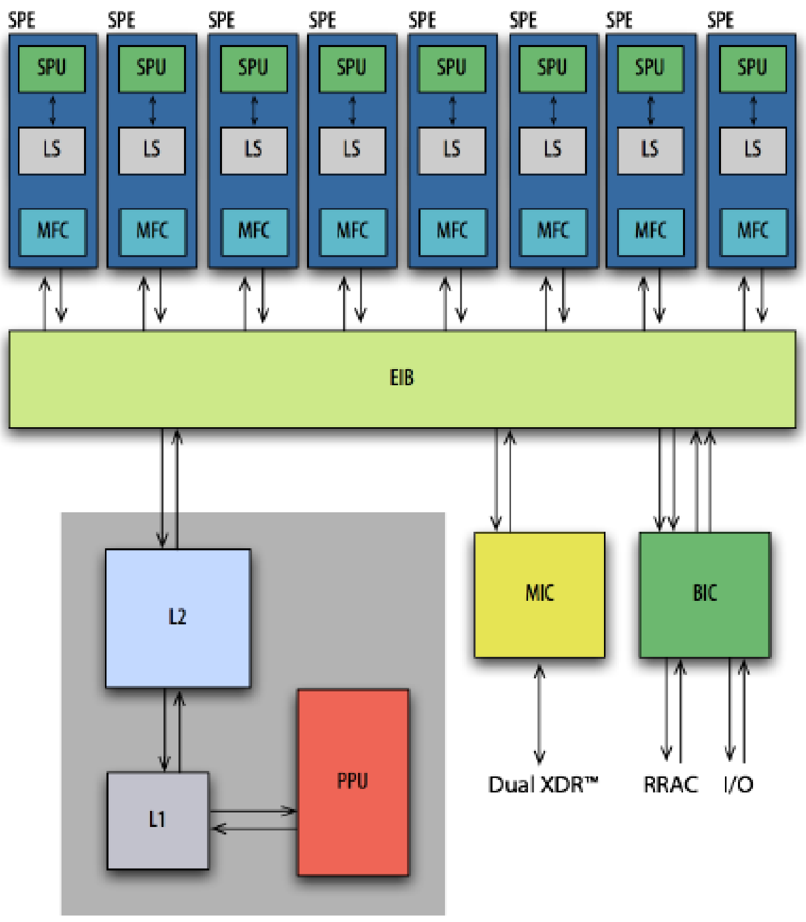


Figure 6: Overview of the CELL architecture

The most important parts of the CELL processor are: the main processor called Power Processing Element (PPE), eight fully functional co-processors called Synergistic Processing Elements (SPEs), a high bandwidth circular data bus connecting the PPE, input/output elements and the SPEs called Element Interconnection Bus (EIB) and the memory system. [3, 4, 7, 24, 29, 55]. The PPE is a 64-bit 2-way simultaneous multithreading (SMT) processor compliant with PowerPC 970 Architecture. It can run both 32-bit and 64-bit operating systems and applications. The PPE consists of the

Power Processing Unit (PPU) and a unified – instruction and data- 512KB 8-way set associative write-back cache. PPU includes a 32KB 2-way set associative reload-on-error instruction cache and a 32KB 4-way set associative write-through data cache. L1 caches are parity protected, L2 caches are protected with error-correction code (ECC). All caches have the same cache line size, which is set to 128 bytes. PPU includes a standard floating point unit (FPU) and a short vector SIMD engine, called VMX. The PPE contains a 64-bit general purpose register set (GRP), a 64-bit floating point register set (FPR) and a 128-bit vector register set. Since PPE is RISC architecture with fixed-width 32-bit instruction format, each register set of the PPE must contain 32 registers.

Although PPE uses the PowerPC 970 instruction set, its architecture is much simpler. For example, it doesn't support out-of-order executions, which can result in lower performance, especially in applications heavy in branching. However, PPE has some capabilities that could probably make up for any potential performance loss, i.e. high clock rate, high memory bandwidth and dual threading.

SMT is a very important feature of the PPE unit. Though it seems that the PPE can offer two independent execution units, actually their resources are shared. But still, this technology can provide 10 to 30% increase in performance with a 5% increase in the cost [54]. PPE's clock rate is 3.2 GHz. Although this looks like a very potential processor, its main job is to supervise and control the SPEs on the chip.

The SPEs work as independent processors and run their own individual application programs. Each SPE processor has full access to the main memory. There is a two-way

dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the OS. The PPE depend on the SPEs to provide performance from application execution.
To achieve best performance of an application running on CELL, you have to use the SPEs more and the PPEs less for the execution. The true power of the CELL processor does not lie in the PPE but on the eight SPEs. The Playstation 3 uses the CELL processor designed by IBM as its main processor, utilizing fully the PPE but only the seven of the eight SPEs [36]. The eighth is disabled in order to improve chip performance [37, 38]. From the seven SPEs left, only six are accessible to developers since the one is reserved by the OS [4].

Each SPE consists of three main parts: a Synergistic Processing Unit (SPU), a 256KB of local memory called Local Store (LS) for both code and data, and a Memory Flow Controller (MFC). Each SPE of the CELL can only execute code from its LS and operate only on data from its LS. Code and data can be moved between main memory and local store through the Element Interconnection Bus (EIB) using Direct Memory Access (DMA) capabilities of the MFC. The EIB is the interface that interconnects all the components of a CELL processor, including PPE, SPEs, main memory and I/O. One main advantage offered by DMA capabilities is that some date can be processed while they are still in flight.

SPEs are short vector SIMD processors. They contain a large 128-entry 128-bit vector register file. An SPE can operate on 16 8-bit integers, 16 8-bit chars, 2 double precision floating-point numbers or 8 single precision floating-point numbers in a single clock cycle.

SPEs have two pipelines, one for arithmetic and one for data motion. Both pipelines issue instructions in order. Sometimes both pipelines can issue an instruction in the same clock cycle.

As mentioned before, the power of the CELL lies on its eight vector processors. SPEs might be the most powerful short vector SIMD engines that exist today [4]. But, although they can process multiple data elements in the same clock cycle, they are not very efficient in scalar/non-vector operations.

In order to write code that targets the SPU cores, you need to write very low level language code, because of the specialized nature of these processors. Rapidmind platform makes things easier for a developer, by allowing the use of high level language programming with C++ for executing code in the SPU cores [3].

3.5 Stream Computing

Stream computing refers to the live processing of multiple data streams, provided from multiple sources at the same time. Many streams flow into the processor and as soon as they are processed, they flow out of the processor as a single flow. The data are analyzed in real time, as soon as they enter the processor.

In June 2007, IBM announced its stream computing system, called System S [8, 9]. It uses 800 multiprocessors and allows both, the split up software applications and the reassemble of the produced data into an answer. System S is a high-performance computer system that is job is not only to analyze data very fast as they stream in from many resources, but to do it accurately.

Stream computing offers faster data handling and analysis. We find it useful in business, mostly in decision making, science, e-mail, video clips, telephone conversations, electronic sensors, transaction data, etc.

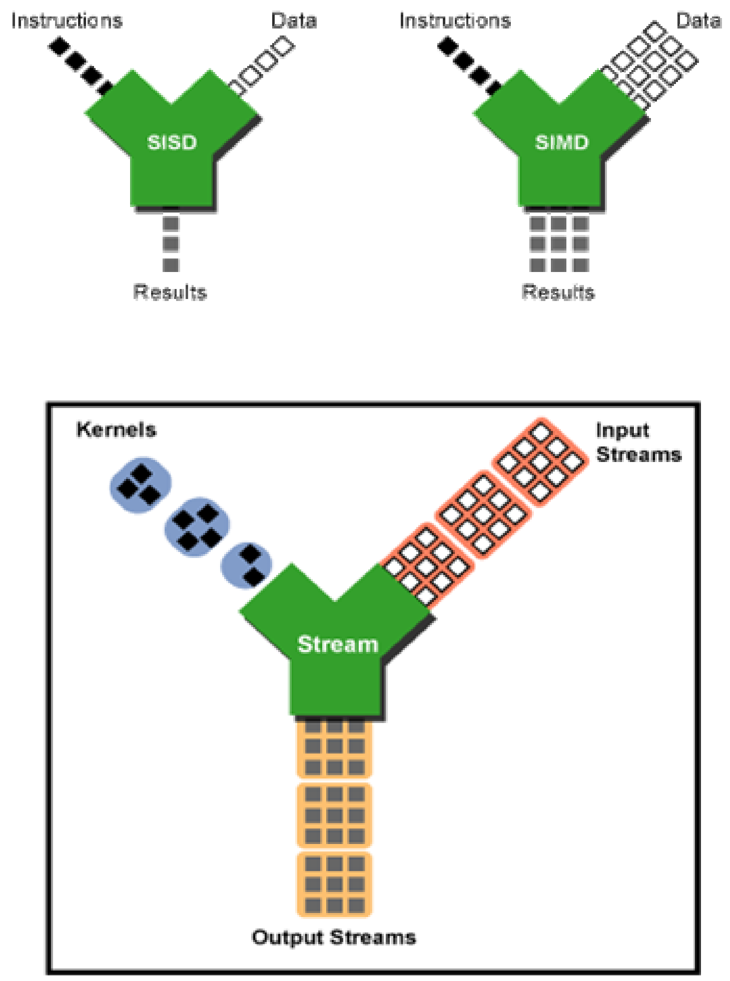


Figure 7: SISD, SIMD, and stream processing compared

In Figure 7, we can see the differences between standard serial Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), and stream programming models. As we can see, the kernels flow into the stream processor along with input streams, while the output streams flow from the processor and back into storage [19].

Another important difference between them is that data and instructions, for SISD or SIMD, are stored in registers, whereas kernels and streams are both stored in the cache.

Stream processing is very promising in areas with high degrees of data parallelism. Researchers have been able to use GPUs as stream processors by coding kernels in a graphics language as vertex shaders and storing streams as textures.

3.6 The Porting Process

The traditional way to create a parallel program consists of four tasks. Firstly, you need to "break" the sequential program into several smaller tasks. Then, you assign these tasks in processes. Third, you create a parallel program that uses the processes of the previous step. And last, map the parallel program in processors. With the new perspective that includes modern and programmable GPUs and newly developed programming environments, these steps have changed. The simpler porting process consists again of four steps: (i) identify the loops in the sequential program, (ii) find and solve any dependencies that the loops might have inside, (iii) design the streams, and finally, (iv) include the streams in the "streamed program".

In sequential programming we don't really care about loop dependencies since everything is going to be executed on its own time. But in parallel programming that's one major issue. We can't have parallelization of loops if we can't solve their dependencies. If we don't solve them, then, either we will not be able to execute our program or we will have unexpected and falsely results. After we break our code into paralleled executed parts then we design our streams. Streams are then filled with data and sent to the processors in parallel, for processing.

The final step of the porting process is the creation of the “Stream Program”. A Stream program consists of input and output streams. It has the ability to perform various operations on the input streams in parallel, and generate output streams.

After porting process has finally completed and we have the stream program, we need to “bind” it with the native code. In our case the stream program is written in Rapidmind and the native code is the C++ code is to read the data, from the database files, that will fill later the streams. Also it coordinates the whole execution and presents the output and results in the end. The bind procedure involves operations that must be done before and after the call of the stream program, i.e. the data filling of streams, the stream program call with the correct input stream parameters and the “host-readback” part of transferring the output streams from the graphics card memory back to the host system’s memory.

In Rapidmind, we have the ability to optimize our GPU code by including some techniques like forceful compilation. This compilation technique is used for avoiding overheads that we have from the translation of the high-level code to the low-level than the GPU uses. Also, we find it useful in the host-readback calls.

Chapter 4

Rapidmind

Rapidmind is a Development Platform used for parallel applications. It supports many different graphics processor units, as well as the CELL processor. It is quite simple C++ language based, with some extensions. Those extensions provide the capability of high performance applications to run on multi-core processors, such as GPUs, CELL Broadband Engine, etc.

Rapidmind platform has roots in the Sh platform [25]. It is actually an embedded programming language inside C++. It uses small set of types and arbitrary functions [4]. These functions are applied to arrays which generate new arrays.

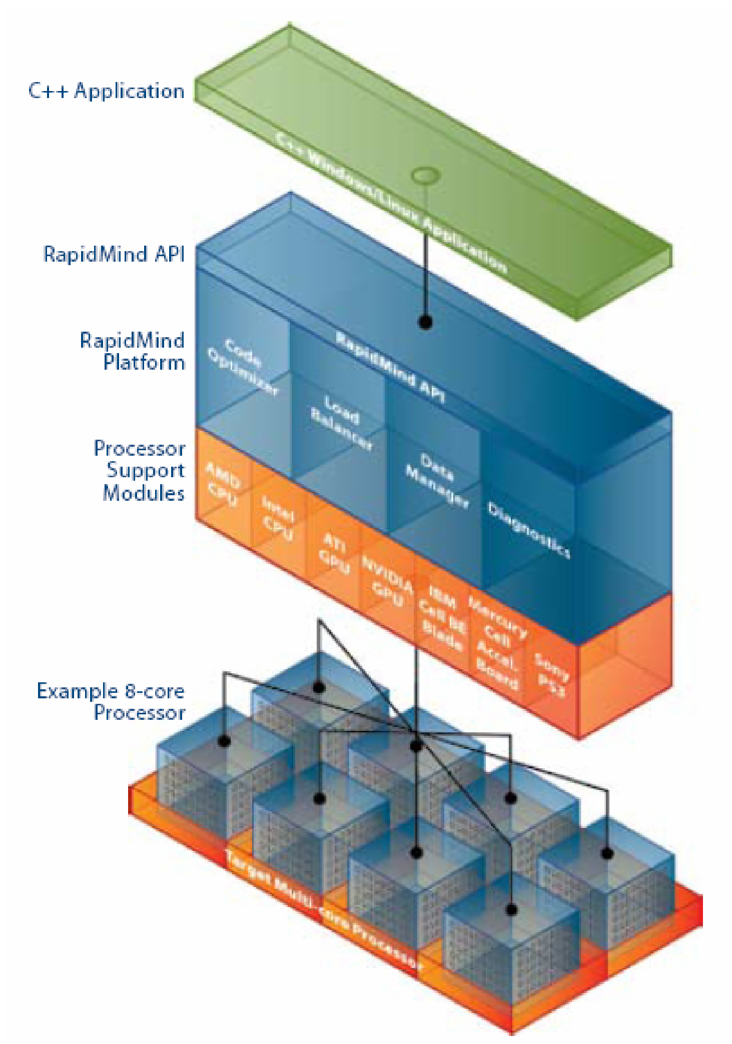


Figure 8: The RapidMind architecture [11]

Rapidmind platform not only allows you to improve performance of applications by parallelizing them, but also allows them to be scalable. The applications written in Rapidmind, automatically scale to the number of available cores.

The architecture of Rapidmind contains three main parts (Figure 8), the API, the Platform and the Processor Support Modules. The first provides the ability of creating Rapidmind applications with the use of existing tools and compilers of C++. The second does all the background low-level work. It manages the communication and data exchange between the host and the target processor. It doesn't require any interaction from the developer. The Platform acts like a fully automated system. The last one refers to the supported backends. Those include all the x86 Intel and AMD CPUs, many ATI and NVIDIA GPUs and last but not least, the SPEs of the CELL Broadband Engine.

To perform a computation in Rapidmind, you still have to declare the different types of variables, like you do in C++. Though, you have to use Rapidmind's types. The three main types are Value, Array and Program. Operations on Rapidmind types can be executed in two ways, [30] either immediately (the usual way) or recorded. For the second, Rapidmind records a sequence of computations compiled dynamically into machine language on the target co-processor. Then, the new code created can be executed later as a function, but can run in parallel over a set of co-processors.

4.1 Rapidmind API

In order to create a Rapidmind program we first need to replace numerical types such as floating point numbers and integers with equivalent Rapidmind types.

It is best to Rapidmind-enable only the most critical parts of an application if we want to achieve results quickly we leave the rest of the existing application code unaffected.

Value type contains a fixed length sequence of floating-point numbers, integers or Booleans. We use them to present data types with multiple components such as (x,y,z) points. The use of 4-component values is sometimes the most efficient way, rather than 1, 2 or 3 component values, because it optimizes the use of registers.

Arrays can be one-, two- or three-dimensional and contain elements of value types. They are used in program processing by holding the data of a program.

Programs in Rapidmind act like C++ functions but they are executed on the target devices. They take Arrays as inputs, operate on them and generate Arrays as output.

4.2 Rapidmind processor support

Rapidmind platform provides an available list of backends. Each backend manages Rapidmind programs execution on a specific processor. The communication and data transfer between host and target processor is managed by the Rapidmind platform. Developer do not need to worry about memory transfers or load balancing. Rapidmind does this for them. This way the developer is free to focus on high-level programming without worrying what is happening on the backstage.

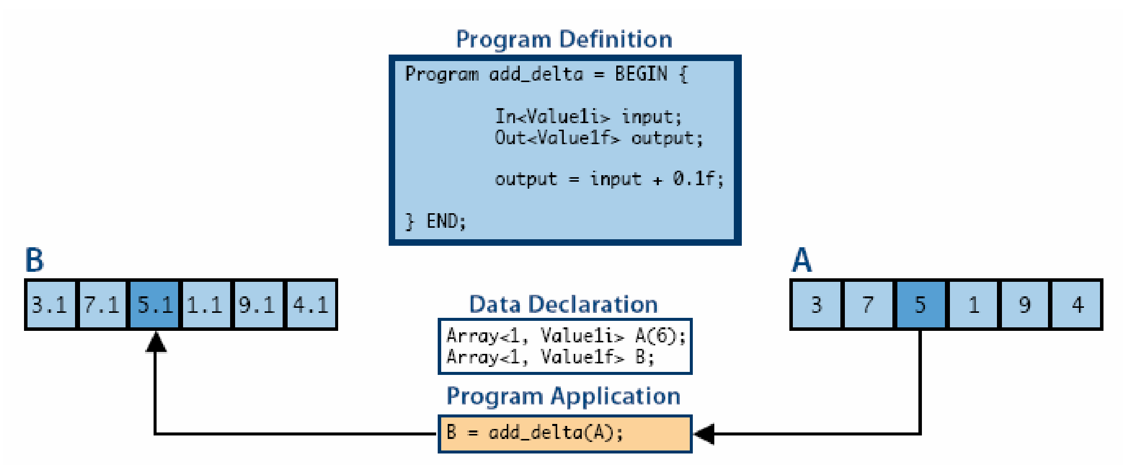
The Processor Support Modules are responsible for compiling Rapidmind programs optimally for the specific processor in use.

The set of backends includes GLSL, Cell BE and Debug. The GLSL backend executes Rapidmind programs on Graphics Processing units (GPUs) using OpenGL Shading Language (GLSL) programs. The Cell BE is used for executing Rapidmind programs on the CELL Broadband Engine. Finally, the Debug backend executes Rapidmind programs on the host processor, using a C compiler for compiling the programs. It is also known as x86 backend. It executes Rapidmind programs on x86 CPUs from Intel and AMD [11].

4.3 Parallelizing an Application

Rapidmind platform uses the data parallel model for parallelizing applications. The data parallel model distributes the application's data among the available cores for processing. The main advantage of this model is that it can easily scale to large number of cores but also maintain the simplicity of a single thread of control. In order to see this benefit we need to have sufficient data for processing.

A parallelized application using Rapidmind platform refers to a set of computations, called Rapidmind programs, on a set of array elements, which are the content of Rapidmind arrays. Developer's job is to design the arrays to contain appropriate sets of data and then write programs that perform operations on these array data.



```
Program add_delta = BEGIN {

        In<Value1i> input;
        Out<Value1f> output;

        output = input + 0.1f;

} END;
```

```
Array<1, Value1i> A(6);
Array<1, Value1f> B;
```

```
B = add_delta(A);
```

Figure 9: How a program is created (in RM) and then can be used to operate as an input array A to generate output array B

An example of a program definition and different types operation is shown in Figure 9. The program takes as an input the array A and performs the same set of computations on each element of A, writing the output to the corresponding element of the output array B. Though the input and output arrays of a program must be of the same size and dimensionality, their elements do not need to be of the same type. Although Figure 9 shows a single element processing, in reality all elements of the array are processed in parallel allowing the program execution to be split over a large number of cores. Figure 10 shows the parallel processing of the same program displayed in Figure 9.

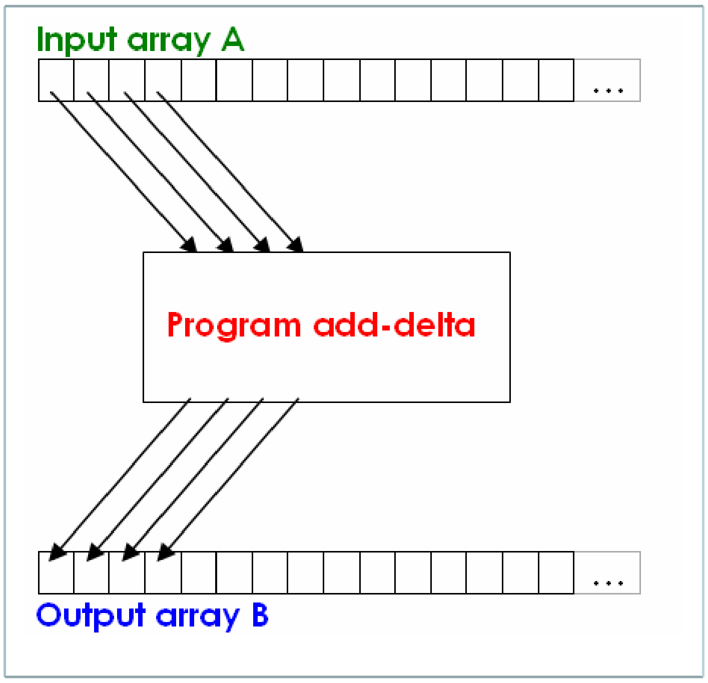


Figure 10: Parallel processing of program in Figure 9.

4.4 How to write a parallel program

To show how we parallelize an application we are going to give an example of C++ code language program and then translate it using Rapidmind platform types and structure. The example code is shown in Figure 11. The application takes a two-dimensional array, multiplies it by using a scalar and then adds it to another array.

```
#include <cmath>
int main () {
    float f;
    const unsigned int w=512,h=512;
    float a [w*h*3], b[w*h*3];
    for (int y=0; y<h; y++)
       for (int x=0; x<w; x++)
          for (int e=0; e<3; e++) {
               a[(y*w+x)*3+e] += b[(y*w+x)*3+e]*f;
          }
  }
```

Figure 11: C++ example code

Firstly we take the first lines of code, which include the defines and declarations, but not the loop and replace the C++ types with the Rapidmind platform types. The translation is shown in Figure 12.

BEFORE:

```
#include <cmath>
int main () {
    float f;
    const unsigned int w=512,h=512;
    float a [w*h*3], b[w*h*3];
```

AFTER:

```
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
using namespace rapidmind;
int main () {
    value1f  f;
    rapidmind :: init ();
    const unsigned int w=512; h=512;
    Array <2,value3f> a(w,h), b(w,h);
```

Figure 12: Translation of C++ defines and declarations into Rapidmind

In the second step, we examine the loops. This is the code of the operations that we want to execute in parallel. We define a program, for example mul-add, and declare two inputs and one output. For the inputs we use the In<Value3f> and for the output the Out<Value3f>. “In” and “Out” are called binding types. There is another one available binding type, the “InOut”, which is both input and output. In our example, the output is generated by multiplying the second input by the scalar f and adds the result to the first input.

When defining a program in Rapidmind we use BEGIN and END signals (or RM-BEGIN and RM-END if we don’t use the Rapidminds shortcuts header file). These macros signal Rapidmind to enter or exit retained mode.

To run the program and have it operate in parallel over the entire array we need only one line of code. We call the program like we would have called a function in C++. No loops are needed but the program will still be executed many times. For our example, the code we need to execute the Rapidmind program is:

```
a=mul-add (a,b)
```

The final translated application is showed in Figure 13.

```
#include <rapidmind/platform.hpp>

#include <rapidmind/shortcuts.hpp>

using namespace rapidmind;

int main () {

    value1f  f;

    rapidmind :: init ();

    const unsigned int w=512; h=512;

    Array <2,value1f> a(w,h), b(w,h);

    Program mul-add=BEGIN {

            In <Value3f> i1, i2;

            Out <Value3f> o;

            o = i1 + i2 * f ;

    } END;

    a=mul-add (a,b);

}
```

Figure 13: The example of C++ code in Figure 11, translated into Rapidmind code

4.5 Rapidmind platform modes

Rapidmind platforms have two modes of execution, retained and immediate. These modes relate only statements that use Rapidmind types.

Rapidmind statements within a program definition are executed in retained mode. When the application and the statements are executed, the operations are recorded and executed at a later point. So, the program object is compiled only once, at runtime, but then can be executed multiple times with no compilation overhead.

Immediate mode refers to the execution of Rapidmind statements outside of a program definition. Their operations are executed immediately, like the statements using only C++ types which are always in immediate mode.

4.6 Rapidmind programming mode

Rapidmind platform offers a quick and easy way to parallelize an application. The developer needs to express an application as a sequence of functions that operate on arrays and the rest is handled by Rapidmind. Rapidmind platform is responsible for dividing an application’s data and operations among the available cores for processing.

Rapidmind uses Single-Program Multiple Data (SPMD) [52] stream processing model which can easily scale to a large number of cores and still keep the simplicity of a single threat of control.

The SPMD model generalizes the Single-Instruction Multiple Data (SIMD) model. The SIMD model is a technique that achieves data level parallelism [52]. It is very similar to vector processing. The main idea of SIMD processing model is to operate the same sequence of instructions simultaneously on a large number of discreet data sets. SIMD machines are very efficient when the applications executed on them present massive amounts of data parallelism without complicated control flow or a lot of inter-processor communication.

The SPMD's main difference from SIMD, is that the first allows multiple autonomous processors to concurrently execute the same program at different points, in contrast to SIMD which requires a lockstep on different data. SPMD also allows tasks to be executed on general purpose CPUs. SIMD cannot allow this since it uses vector processing, which requires vector processor existence in order to operate on data streams. Although SIMD model is very efficient, it is not very easy to find applications that it is applicable. Not all algorithms can be vectorized and this is a prerequisite for SIMD model to work. For example, a flow-control-heavy task wouldn't get any benefit from SIMD. SPDM model is proven to be more flexible, by allowing arbitrary functions to be applied to elements of an array. In addition to this, those functions can contain branching and control flow [11]. Another benefit we get from using SPMD model, is that it includes a definition of how data is decomposed and distributed to the tasks. By using function, we get the privilege of having large amount of computation for every memory access, and a mechanism for expressing data locality. This is important for good efficiency on modern processors. This model of parallel computation is also known as stream processing, since it does a lot of streaming of data into and out of the computational kernel. A lot of systems are using the SPMD stream processing model, including Brook, CUDA and Rapidmind platform. Note that the array abstraction can support both shared and distributed memory.

Unlike the other alternative systems, Rapidmind's platform interface is embedded in standard C++ and available for usage from existing compilers. As said before, we use Rapidmind types, replacing C++ standard types, and we create program objects. Program objects represent arbitrary computations. They act similarly to C++ functions, though they can operate on entire arrays and do computations in parallel. If control flow exists in a program object, the platform uses dynamic load balancing automatically. If not, then static scheduling is being used, because we have a pure SIMD data parallelism. One Rapidmind application might have several such program objects and can all be executed asynchronously in parallel. However, when there is a need for reading the output produced by a program invocation, the platform automatically blocks the host process until any pending computation generating that data is complete.

To this end, by using Rapidmind platform, the developer can write an application using simple operations and sequential semantics. The use of program objects is the way to express parallelism by using asynchronous execution. The handling of data dependencies is performed by the platform.

Chapter 5

Database Operations on CPU, GPU and CELL

In this research we test the performance of two different GPUs and the CELL processor, in executing three of the Transaction Processing Performance Council's Decision Support Benchmark (TPC-H) queries, using real data created from the DBGEN tool. Queries implemented from TPC-H are Q3, Q6 and Q12, described in the following pages. All three of these queries perform operations like Boolean combinations, sum aggregations as well as join operations.

Further I will give analytical description of how the queries, used in this research, were implemented.

5.1 Implementation

There are three types of join algorithms for the databases: nested-loop, merge and hash join. For the implementation I used the nested-loop join algorithm, combining native and GPU execution. To be more specific, I have implemented the following join-algorithm: iterate though all the elements of the one table (sequential scan) and perform the related

Boolean combinations and CASE operations (if any) in native code, and each time invoke the GPU stream program. Every time the stream program is invoked, it iterates in parallel all the elements of the other table and whenever the foreign key matches with the primary key of the “outer” table, it performs all the other Boolean combinations and checks. The sum aggregations of each query are performed in the native code.

Although hash join is more efficient than nested-loop join and index scan more efficient than sequential scan, but I have chosen the less efficient algorithms because they map better to the stream model and are easier to implement in Rapidmind.

5.2 TPC-H Benchmarks

Transaction Processing Performance Council (TPC) [33] is a non-profit corporation founded to define transaction processing and database benchmarks and to deliver trusted results about performance, to the industry. The word “transaction” refers to a set of operations, including disk read/write, operating system calls or data transfers from one subsystem to another. TPC produces benchmarks that measure transaction processing and database performance in terms of how many transactions a given system and database can perform per unit of time.

One of the TPC benchmarks is the TPC-H. It is a Decision Support System (DSS) benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modification. Queries of this benchmark deal with large volumes of data, are very complicated and their results actually give answers to some critical business questions.

A tool provided by TPC is DBGEN. It is used for database table creation for the TPC-H benchmark. We used it in this research for creating the data for the LINEITEM, ORDERS and CUSTOMERS tables.

The three queries were chosen based on simplicity of implementation. Also we have considered the number of join operations in each query. Q6 does not have any join operations, Q12 has one and Q3 has two. From the 22 queries of the TPC-H, we chose the three we considered that they fit better to the stream model and could be represented using tables of floats. They are queries with and without data dependencies that can be parallelized.

5.3 TPC-H Q3: Shipping Priority Query

The actual query retrieves the ten unshipped orders with the highest revenue, where revenue refers to the sum of l_extendedprice * (1 - l_discount). Both fields (l_extendedprice and l_discount) are taken from the LINEITEM table. The query, as found in the TPC-H publication, is shown in Figure 14.

```
Return the first 10 selected rows

select
        l_orderkey,
        sum(l_extendedprice*(1-l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = '[SEGMENT]'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '[DATE]'
        and l_shipdate > date '[DATE]'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate;
```

Figure 14: TPC-H Q3 in SQL code

For sake of simplicity and focusing on what we want to investigate, we didn't implement the "group by" and "order by" part. We want to count the total execution time for querying all the data, execute all the predicate evaluations from the "where" statement, and performing the sum aggregation from the "select" statement. What we actually implemented is stated in Figure 15, described again in SQL format.

```
select
        sum(l_extendedprice*(1-l_discount)) as revenue,
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'FURNITURE'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995'
        and l_shipdate > date '1995'
```

Figure 15: Our TPC-H Q3 implementation in SQL code

For experimental purposes, we have assumed that the DATE constants and all date-related columns should be represented in terms of years only and in float format. Also, SEGMENT constant and c_mktsegment field will be represented in the float format. The c_mktsegment field of the CUSTOMER table has five possible values {AUTOMOBILE, BUILDING, FURNITURE, HOUSEHOLD, MACHINERY}. These values are actually character values. For our experiments, we want to perform operations and comparisons only in floats. So, we represent each segment value with a different float number. For example, the FURNITURE value will be the number 3,00.

The GPU operates based on the Stream Programming Model. Owens defines stream as an “ordered set of data of the same data type” [69]. This data type can vary from simple integers or floating-point numbers, to points or triangles. Computations can be performed on streams with the use of special “stream” programs. The Rapidmind Platform adopts this model and that is the reason for "encoding" all values into floats.

```
// Import libraries
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
#include <rapidmind/platform/Context.hpp>

//Set the use of Rapidmind
using namespace rapidmind;

                                <...>

//Declaration of Rapidmind types
Value4f curr_ord;
Array<1,Value4f> l_orderkeyx(LINEITEM/4);
Array<1,Value4f> l_shipdatex(LINEITEM/4);
Array<1,Value4f> result1(LINEITEM/4);
Array<1,Value4f> result2(LINEITEM/4);
Array<1,Value4f> tmp1(LINEITEM/4),D;
Array<1,Value4f> tmp2(LINEITEM/4),E;

int main(){
      //Rapidmind initialization
      init();
      //Set the backend for GPU
      set_backend("glsl");

      //Define the streams
      float* l_orderkeyx_data=l_orderkeyx.write_data();
      float* l_shipdatex_data=l_shipdatex.write_data();
      float* result1_data=result1.write_data();
      float* result2_data=result2.write_data();
                                <...>
      //Define Rapidmind Program for the first join of the query
      Value4f o_orderkeyx;
      Value4f o_custkeyx;
      Program join1= BEGIN {
            In<Value4f> l_orderkeyx;
            In<Value4f> l_shipdatex;
            InOut<Value4f> feededResult;
            IF(any(l_orderkeyx==o_orderkeyx)&&(l_shipdatex>1995.0f)){
                  feededResult=o_custkeyx;
            }ENDIF;
      } END;

      //Define Rapidmind Program for the second join of the query
      Value4f c_custkeyx;
      Program join2= BEGIN {
            In<Value4f> tmpresults;
            InOut<Value4f> feededResult;
            IF((tmpresults==c_custkeyx)){
                  feededResult=c_custkeyx;
            }ENDIF;
      } END;

                  <...continue at next page...>
```

```
                    <...continue from previous page...>

       //Fill the input streams and call of RM program for first join
       for(int i=0;i<ORDERS;++i){
             if(orders[i][2]<1995) {
                   o_orderkeyx = (Value4f)orders[i][0];
                   o_custkeyx = Value4f(orders[i][1]);
                   result1=join1(l_orderkeyx,l_shipdatex,result1);
             }
       }

       //Fill the input streams and call of RM program for second join
       for(int i=0;i<CUSTOMER;++i){
             if ( customer[i][1] == 2 ) {
                   c_custkeyx = (Value4f)(customer[i][0]);
                   //use the result of 1st join to the 2nd
                   result2 = join2(result1, result2);
             }
       }

       //Get the output from the result stream
       read=result2.read_data();
                                   <...>
```

Figure 16: Sample parts from Rapidmind code for Q3

The constant values that we used in our code for the comparisons in all queries (i.e. the 1994 and 1995 dates, the FURNITURE value, etc) are taken from the TPC-H publication, from the example of each query. This made it easier for us to validate our query results.

For the implementation of this query in Rapidmind, we had to include two join operations. Part of our code in Rapidmind is shown in Figure 16. Firstly we have the definition of libraries, the declaration of Rapidmind types and initialization of Rapidmind. Then there is the Rapidmind code for the query. The two joins are implemented in two separate Rapidmind programs. The second join uses the result of the first join. After the definition of the Rapidmind programs we load the input streams and call the Rapidmind programs. At the end, we read the output stream for the results.

In order of complexity, Q3 is the most complex of the three queries implemented and therefore the most time-consuming. The two joins of this query are implemented in sequential order. We need the first join's result in order to proceed to the second.

5.4 TPC-H Q6: Forecasting Revenue Change Query

The easiest query to be implemented, between the three that we study, was Query 6. The forecasting Revenue Change Query gives the amount of revenue increase that would have resulted from eliminating certain company-wide discounts given a percentage range and a year. The revenue value is the product of l_extendedprice and l_discount. The SQL code for this query, as found in TPC-H Publication, can be seen in Figure 17.

```
select
        sum(l_extendedprice*l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '[DATE]'
        and l_shipdate < date '[DATE]' + interval '1' year
        and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + 0.01
        and l_quantity < [QUANTITY];
```

Figure 17: TPC-H Q6 in SQL code

This query uses only the LINEITEM table. For this implementation we didn’t need to include any joins. It includes only one sum aggregation, the calculation of revenue. Our implementation, described in terms of SQL, is shown in Figure 18. Again, as in the previous query, all constant values and related fields are presented in float format.

```
select
        sum(l_extendedprice*l_discount) as revenue
from
        lineitem
where
        l_shipdate >= date '1994'
        and l_shipdate < date '1995'
        and l_discount between (0.01 - 0.01) and (0.01 + 0.01)
        and l_quantity < 24;
```

Figure 18: Our TPC-H Q6 in SQL code

```
//Import libraries
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
#include <rapidmind/platform/Context.hpp>

//Set the use of Rapidmind
using namespace rapidmind;
                                      <...>

//Declaration of Rapidmind types
Array<1,Value4f> l_extendedprice(LINEITEM/4),A;
Array<1,Value4f> l_quantity(LINEITEM/4),B;
Array<1,Value4f> l_year(LINEITEM/4),C;
Array<1,Value4f> l_discount(LINEITEM/4),D;
Array<1,Value4f> output;
int main(){
      //Rapidmind initialization
      init();
      //Set the backend for GPU
      set_backend("glsl");

      //Define the streams
      float* l_extendedprice_data = l_extendedprice.write_data();
      float* l_quantity_data = l_quantity.write_data();
      float* l_year_data = l_year.write_data();
      float* l_discount_data = l_discount.write_data();
                                      <...>
      //Define Rapidmind Program
      Program addition_program = RM_BEGIN {
            In<Value4f> l_extprice_v1;
            In<Value4f> l_gty_v2;
            In<Value4f> l_year_v3;
            In<Value4f> l_disc_v4;
            Out<Value4f> out;

            IF((l_year_v3>=(Value4f)(float)DATE1) &&
                  (l_year_v3<(Value4f)(float)DATE2) &&
                  ((l_gty_v2>((Value4f)(float)DISCOUNT-(Value4f)(float)0.01)) &&
                  (l_gty_v2<((Value4f)(float)DISCOUNT+(Value1f)(float)0.01))) &&
                  (l_disc_v4<(Value4f)(float)QUANTITY)){
                  out=(l_extprice_v1*l_year_v3);
            }ELSE{
                  out=(Value4f)-1;
            }ENDIF;
      } RM_END;
                                      <...>
      //Call Rapidmind Program
      output=addition_program(l_extendedprice,l_quantity,l_year,l_discount);
      //Get the output from the result stream
      result=output.read_data();
                                      <...>
```

Figure 19: Sample parts from Rapidmind code for Q6

Part of our code in Rapidmind is shown in Figure 19. Firstly we do the defines, declarations and initialization and then we have the code for the query, written as a Rapidmind program. Then we call the program and read the result from the output stream.

5.5 TPC-H Q12: Shipping Modes and Order Priority Query

Query number 12 is a bit more complex than Q6 but less complex than Q3. The query is used to determine whether by selecting less expensive models of shipping is negatively affecting the critical-priority orders by causing customers to receive more parts later than committed date. The TPC-H Query 12 is shown in Figure 20.

```
select
        l_shipmode,
        sum    (case
                when o_orderpriority ='1-URGENT' or o_orderpriority ='2-HIGH'
                then 1
                else 0
                end) as high_line_count,
        sum    (case
                when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH'
                then 1
                else 0
             end) as low_line_count
from
        orders,
        lineitem
where
        o_orderkey = l_orderkey
        and l_shipmode in ('[SHIPMODE1]', '[SHIPMODE2]')
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate >= date '[DATE]'
        and l_receiptdate < date '[DATE]' + interval '1' year
group by
        l_shipmode
order by
        l_shipmode;
```

Figure 20: TPC-H Q12 in SQL code

For this query, like in Q3, we didn't implement the "group by" and "order by" parts. Also, from the two sum-aggregations that this query includes, we choose to implement only one. This was done for simplification without affecting the overall complexity and computation-intensiveness of the query. All the fields and constants of the query are again represented in float format, like in the other queries. The l_shipmode field of the LINEITEM table has seven different possible values {TRUCK, MAIL, REG AIR, AIR, FOB, RAIL, SHIP}. Just like segment values of Q3 query, we represent each shipmode with a unique float value. The simplified version of query that we implemented is presented in Figure 21.

```
select
        sum     (case
                when o_orderpriority ='1-URGENT' or o_orderpriority ='2-HIGH'
                then 1
                else 0
                end) as high_line_count,
from
        orders,
        lineitem
where
        o_orderkey = l_orderkey
        and l_shipmode in ('MAIL', 'SHIP')
        and l_commitdate < l_receiptdate
        and l_shipdate < l_commitdate
        and l_receiptdate >= date '1994'
        and l_receiptdate < date '1995'
```

Figure 21: Our TPC-H Q12 in SQL code

This query implementation includes only one join. Most important parts of our code implementation in Rapidmind is shown in Figure 22. We can see the definitions of Rapidmind arrays, the initialization of Rapidmind, the definition of the Rapidmind Program that represents the query code and the call of the Rapidmind program. The program produces an output stream which we read afterwards.

```
//Import libraries
#include <rapidmind/platform.hpp>
#include <rapidmind/shortcuts.hpp>
#include <rapidmind/platform/Context.hpp>

//Set the use of Rapidmind
using namespace rapidmind;
                                                <...>
//Declaration of Rapidmind types
Array<1,Value4f> order_key(LINEITEM/4);
Array<1,Value4f> shipmode(LINEITEM/4);
Array<1,Value4f> commit_date(LINEITEM/4);
Array<1,Value4f> ship_date(LINEITEM/4);
Array<1,Value4f> receipt_date(LINEITEM/4);
Array<1,Value4f> temp(LINEITEM/4),A;
Array<1,Value4f> result(LINEITEM/4);

int main(){
      //Rapidmind initialization
      init();
      //Set the backend for GPU
      set_backend("glsl");

      //Define the streams
      float* d_order_key=order_key.write_data();
      float* d_shipmode=shipmode.write_data();
      float* d_commit_date=commit_date.write_data();
      float* d_ship_date=ship_date.write_data();
      float* d_receipt_date=receipt_date.write_data();
      float* d_result=result.write_data();
      float* d_temp=temp.write_data();
                                                <...>
      //Define Rapidmind Program
      Value4f o_orderkeyx;
      Program gpu_join= RM_BEGIN {
            In<Value4f> l_orderkey; // order key - result 1
            In<Value4f> l_shipmode; // shipmode
            In<Value4f> l_commitdate; // commid date
            In<Value4f> l_shipdate; // ship date
            In<Value4f> l_receiptdate; // receipt date
            InOut<Value4f> output;

            IF ((all(l_orderkey==o_orderkeyx))&&
               ((l_shipmode==(Value4f)2)||
                (l_shipmode==(Value4f)7))&&
               (l_commitdate<l_receiptdate)&&
               (l_shipdate<l_commitdate)&&
               ((l_receiptdate>=(Value4f)1994)||
                (l_receiptdate<(Value4f)1995))){

                  output=(Value4f)1;

            }ENDIF;
      }RM_END;
```

<...continue in the next page...>

<...continue from the previous page...>

```
//Fill the input stream and call Rapidmind program
for(int i=0;i<ORDERS;++i){
      o_orderkeyx=Value4f(orders[i][0]);
      result = gpu_join(order_key,shipmode,commit_date,ship_date,
               receipt_date,result);
}

//Get the result from the outpus stream
read=result.read_data();
```

<...>

Figure 22: Sample parts from Rapidmind code for Q12

Chapter 6

Experimental Results

6.1 Experimental Setup

My implementation consists from two main parts: the Rapidmind code and the native C++ code. The Rapidmind code is used for the GPUs (both 8500 and 8800 NVIDIA chipset) and the CELL SPEs. The C++ code is implemented for the CPU and the CELL PPE processor.

For this project we used Rapidmind Development Platform Version 2.0. As for the operating systems, on the desktop machine we used Windows XP x86 Service Pack 2, with Microsoft DirectX 9.0c Application Programming Environment (API) for graphics rendering. Also we have installed the latest NVIDIA drivers for the GeForce 8500 and 8800 (Version 178.13). On the Playstation 3 we installed Fedora Core 5 OS. On the desktop we used Microsoft Visual C++ 2005 development system for writing, compiling and executing our code. For the CELL code we used a GNU GCC compiler.

The Host-Readback procedure is always being counted in the performance results of Rapidmind codes, in order to be “fair” to the native C++ code. The readback of the results is something that we want to do almost always in general-purpose applications and most definitely in database queries.

The scale factor that we use refers to the table sizes and the relation between them. Scale factor and table data sizes are shown in Table 1 below.

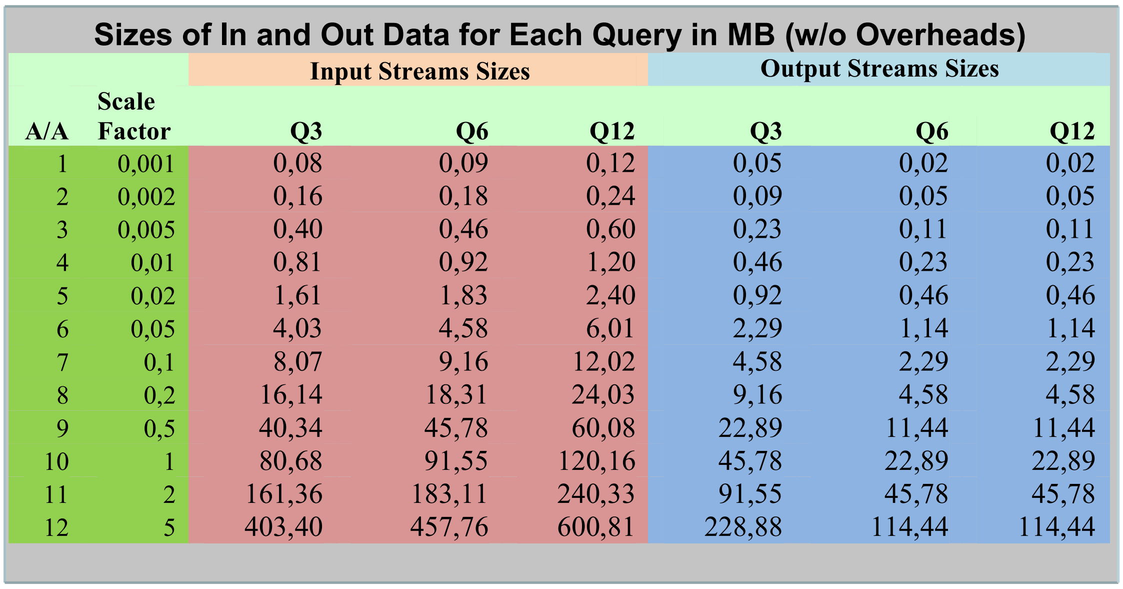
Table 1: Scale Factor and Data Table Sizes

DATA SIZES

		Number of Elements			Memory Residence		
A/A	Scale Factor	LINEITEM	CUSTOMER	ORDERS	LINEITEM	CUSTOMER	ORDERS
1	0,001	6.000	150	1.500	0,712 MB	0,0232 MB	0,165 MB
2	0,002	12.000	300	3.000	1,425 MB	0,0464 MB	0,33 MB
3	0,005	30.000	750	7.500	3,562 MB	0,116 MB	0,825 MB
4	0,01	60.000	1.500	15.000	7,12 MB	0,232 MB	1,65 MB
5	0,02	120.000	3.000	30.000	14,12 MB	0,464 MB	3,3 MB
6	0,05	300.000	7.500	75.000	35,62 MB	1,16 MB	8,25 MB
7	0,1	600.000	15.000	150.000	71,2 MB	2,32 MB	16,5 MB
8	0,2	1.200.000	30.000	300.000	141,2 MB	4,64 MB	33 MB
9	0,5	3.000.000	75.000	750.000	356,2 MB	11,6 MB	82,5 MB
10	1	6.000.000	150.000	1.500.000	712 MB	23,2 MB	165 MB
11	2	12.000.000	300.000	3.000.000	1412 MB	46,4 MB	330 MB
12	5	30.000.000	750.000	7.500.000	3562 MB	116 MB	825 MB

We need to note something very important here, regarding Table 1. For example, in Q6 query, where we use only LINEITEM table, we don’t transfer the whole table in GPU. We only transfer the records of the table needed for that query in streamed-floats form. In the case of Q6 we need 4 input streams (l_extendedprice, l_shipdate, l_discount, l_quantity) and we export one output stream (l_extendedprice*l_discount). Each float has a size of 4 bytes. In the case of scale factor 1 of Q6, we need 4 streams of 6.000.000 floats for each stream. That means we transfer 4 streams * 6.000.000 floats * 4 bytes, which results to 96.000.000 bytes of data. This result translates to approximately 100MB of input data, without any transfer overheads and output data. We can see the sizes of data in and out for each query and each scale factor in Table 2.

Table 2: Scale Factor with In and Out Data Sizes for each Query



Sizes of In and Out Data for Each Query in MB (w/o Overheads)

		Input Streams Sizes			Output Streams Sizes		
A/A	Scale Factor	Q3	Q6	Q12	Q3	Q6	Q12
1	0,001	0,08	0,09	0,12	0,05	0,02	0,02
2	0,002	0,16	0,18	0,24	0,09	0,05	0,05
3	0,005	0,40	0,46	0,60	0,23	0,11	0,11
4	0,01	0,81	0,92	1,20	0,46	0,23	0,23
5	0,02	1,61	1,83	2,40	0,92	0,46	0,46
6	0,05	4,03	4,58	6,01	2,29	1,14	1,14
7	0,1	8,07	9,16	12,02	4,58	2,29	2,29
8	0,2	16,14	18,31	24,03	9,16	4,58	4,58
9	0,5	40,34	45,78	60,08	22,89	11,44	11,44
10	1	80,68	91,55	120,16	45,78	22,89	22,89
11	2	161,36	183,11	240,33	91,55	45,78	45,78
12	5	403,40	457,76	600,81	228,88	114,44	114,44

When referring to speedup, we use the CPU execution time as a baseline and divide the execution time of the platform we want to compare with the one of the CPU. In cases that we don't compare with CPU execution we note that the comparison is with another platform, i.e. the SPE comparison with the PPE.

The formula for calculating speedup for the GPUs, assuming that the baseline for each experiment is the CPU execution, is:

$$Speedup_{GPU} = \frac{ExecutionTime_{CPU}}{ExecutionTime_{GPU}} \quad [71]$$

For calculating the speedup of CELL's SPEs, we use two formulas. One uses as a baseline the CPU execution and the other the PPE execution. Both of the baselines are sequential executions of native C++ code. Both formulas as shown below:

$$SpeedupA_{CELL\_SPE} = \frac{ExecutionTime_{CPU}}{ExecutionTime_{CELL\_SPE}}$$

$$SpeedupB_{CELL\_SPE} = \frac{ExecutionTime_{CELL\_PPE}}{ExecutionTime_{CELL\_SPE}}$$

In Table 3 we can see the technical specifications, focusing on the stream processors, of the two GPU processors that we are going to examine in this project. The main differences between the two GPUs are the number of stream processors and the memory bandwidth. The 8800 GPU has 112 stream processors and 64GB/sec memory bandwidth. The 8500 GPU has 16 stream processors and 12.8GB/sec memory bandwidth. These two factors play significant role upon utilization of the GPU for the execution of our experiments. The number of stream processors controls the degree of parallelism for the GPUs. The memory bandwidth controls the amount of data that can be transferred from and to the graphics card memory.

The baseline CPU system is an Intel core 2 Duo processor, clocked at 2.13GHz with 2GB of DDR2 RAM .

Table 4 specifies the main specifications of the CELL processor and memory. The PPE and the SPEs of the CELL processor are clocked at 3.2GHz. We have eight SPEs but in the Playstation 3 we can use only six for programming purposes. So, in comparison to the 8500 and 8800 GPUs, the CELL has less stream processors.

Table 3: NVIDIA GeForce 8500 and 8800 Specifications

	GeForce 8500 GT	GeForce 8800 GT
Stream Processors	16	112
Core Clock (MHz)	450	600
Shader Clock (MHz)	900	1500
Memory Clock (MHz)	400	900
Memory Amount	512MB DDR2	512MB DDR3
Memory Interface	128-bit	256-bit
Memory Bandwidth (GB/sec)	12.8	64
Texture Fill Rate (billion/sec)	3.6	41.6

Table 4: CELL Specifications

	CELL
PPE:	PowerPC – base core @ 3.2GHz
	1 VMX vector unit per core
	512 KB L2 cache
8 x SPE:	3.2GHz
	128bit 128 SIMD GPRs
	256 KB SRAM for SPE
Memory:	256MB main RAM @ 3.2GHz
	256MB GDDR3 VRAM @ 700MHz

6.2 Selection of float types in Rapidmind

Given the characteristics of the graphics applications, the GPU programming model offers some extra data types that are interesting to analyze. These data types are data structures with multiple float elements that are used in graphics applications to represent, for example, the position of a point in a multi-dimensional space or its colour. As such, the programming model offers the floatN types where N is a number between 1 and 4. For example, a float4 variable has four fields x, y, z, and w.

We had to choose between different types of floats in the Rapidmind code. Selection was between float1, float2, float3 and float4. For code that was going to run in the GPU, we used the float4 type. We can see from the Figures 23 (a, b, c) below the difference between the execution of all float types in the GeForce 8500. In these Figures, we use as a baseline the CPU execution time. These results have a logical explanation. The graphic cards are designed to accelerate images using 4 dimensions for the colour. The 4-dimension model is known as the RGBA Colour Space model, where R stands for Red, G for Green, B for Blue and A for Alpha. The alpha value has to do with colour transparency. By using float4 type we take advantage of this 4-dimension ability. Also, with float4 we transfer smaller streams, therefore we gain in transfer time and transfer overhead.

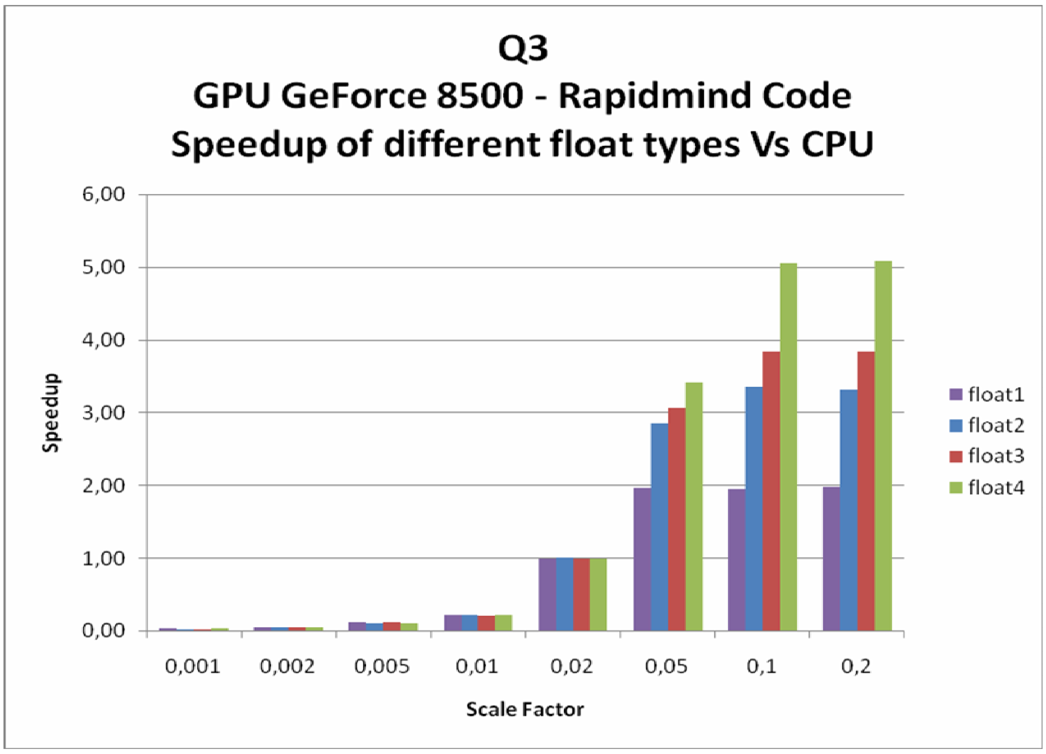


Figure 23 (a) Q3 execution on GPU GeForce 8500 with different float types

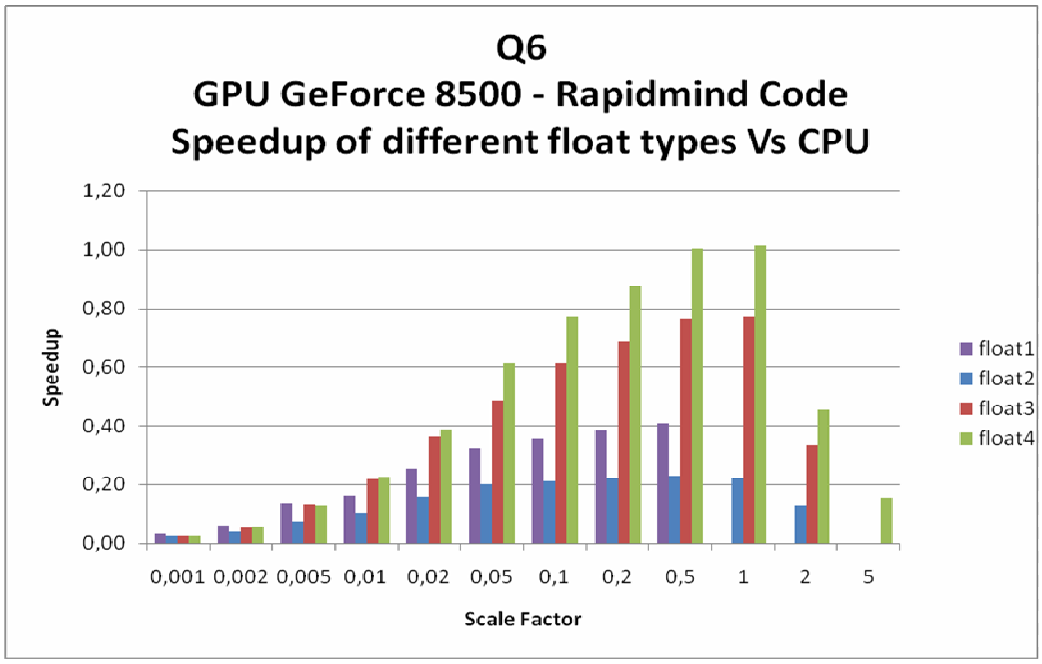


Figure 23 (b) Q6 execution on GPU GeForce 8500 with different float types

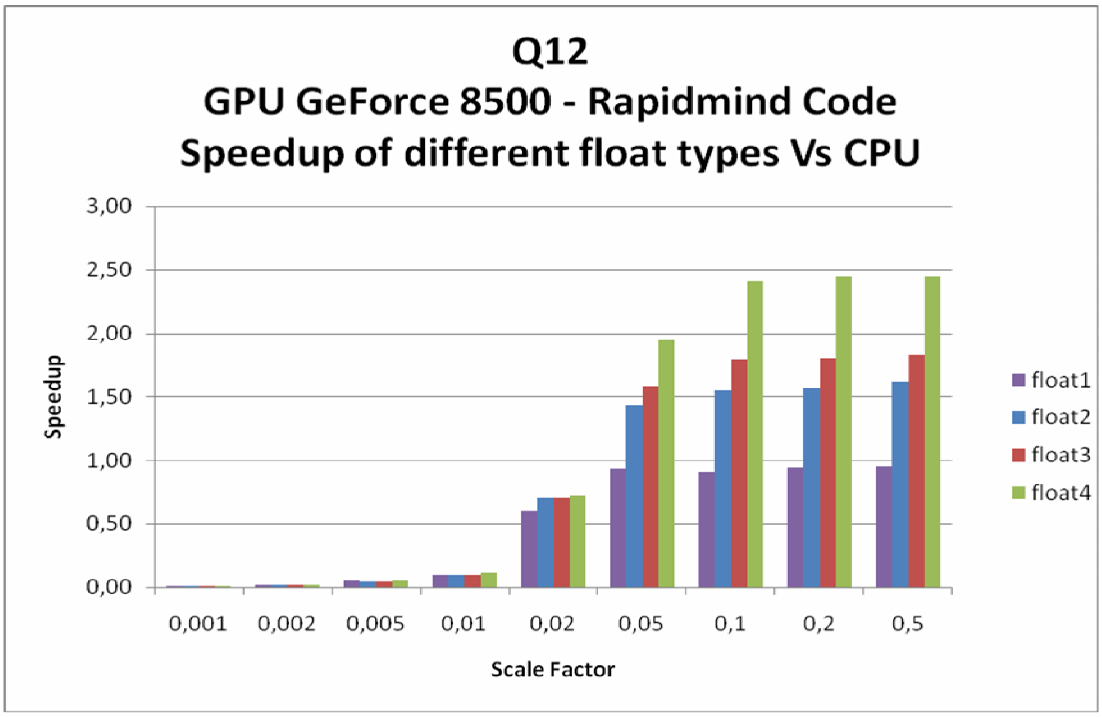


Figure 23 (c) Q12 execution on GPU GeForce 8500 with different float types

For Q3 (Fugure 23a) and Q12 (Figure 23c), we didn't execute them for a larger scale factor than 0,2 and 0,5 respectively, like we did for Q6 (Figure 23b). The reason for this was the execution time needed for them to execute. Both queries, for the larger data sets, needed about a day or more to execute once. We were taking our results by executing all queries ten times for each scale factor, then removed the minimum and maximum execution time, and then calculate the average execution time for the rest of the values. So, we didn't have the available time to do this for larger scales of Q3 and Q12 queries.

In Q6 (Figure 23b) we observe something important. After the scale factor of 1 we have a decrease of performance. This is due to memory swaps happening in the GPU memory which has only 512MB available and needs to fit more data in the memory. Also, for float1 type we can see that after the scale factor of 0,5 it cannot fit the streams to the GPU memory. Same goes for float2 and float3 types, after the scale factor of 2.

In the CELL there was not a noticeable difference between different float type executions (Figures 24 a, b, c). Baseline for these speedups is the execution on the CELL's PPE processor. We used PPE as a baseline because it is the CELL's CPU, it's on the same system platform as the SPEs – where the Rapidmind code is executed -, and it executes the same code as the desktop CPU.

In the examples given by Rapidmind Development Platform, there was a suggestion for using float3 type on the CELL. So, instead of using float4 as for the GPU, we used float3 for the CELL. For our more complex queries, Q3 and Q12, we are able to notice a slightly higher speedup of the float3 type than the other types, especially in the highest scales. In Q6 case, float4 seems to perform a bit better but we prefer to have best performance in more complex queries than simpler.

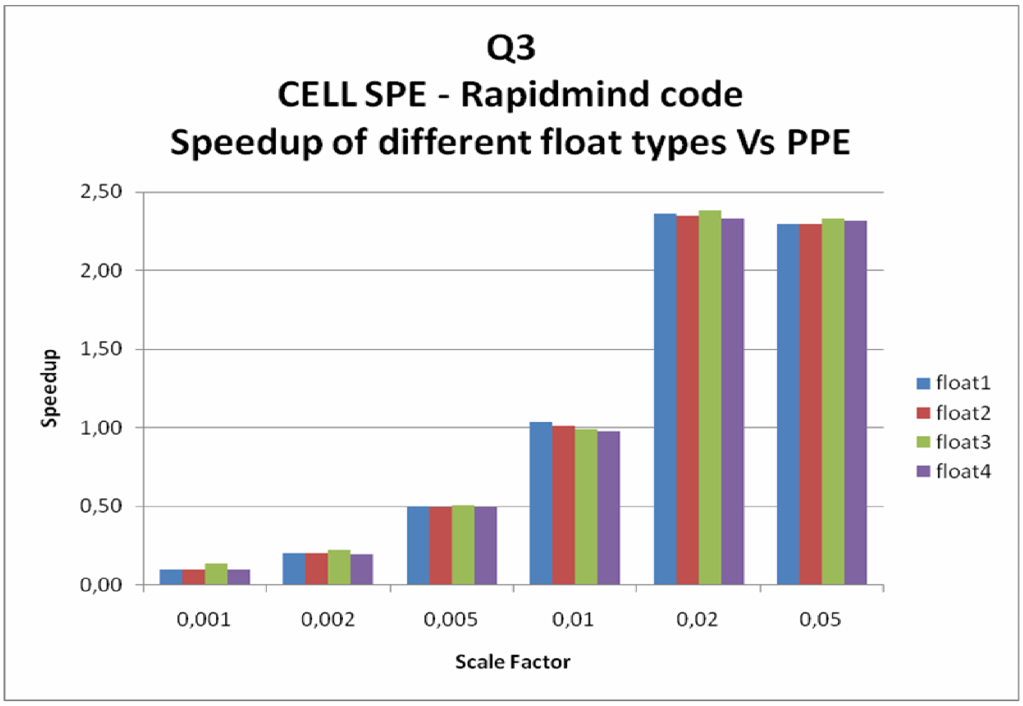


Figure 24(a) Q3 execution on CELL SPEs with different float types

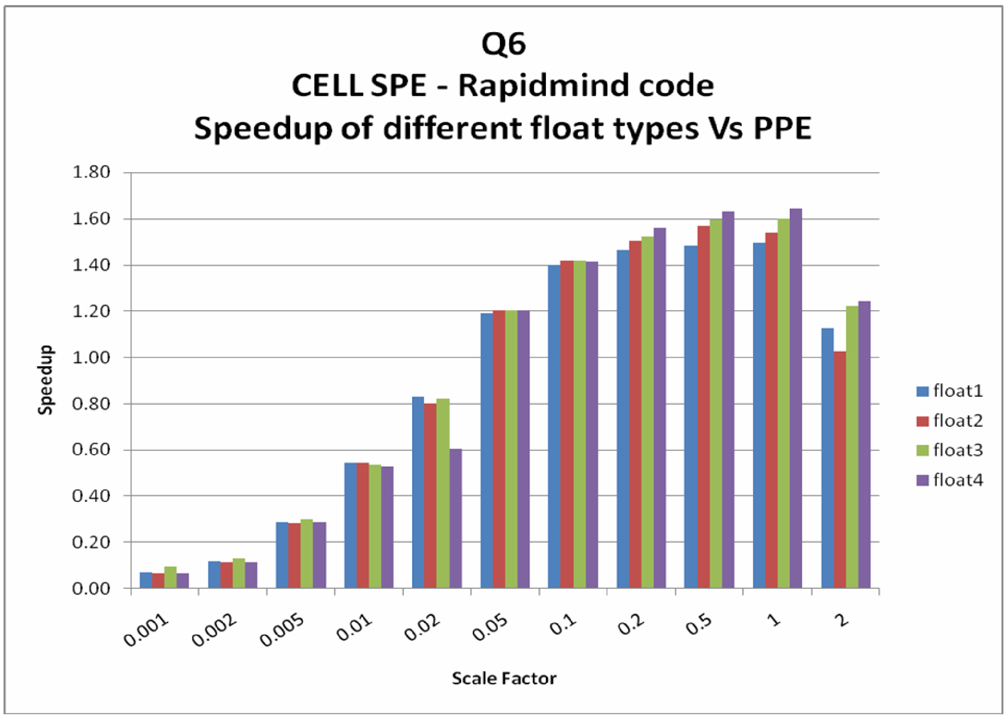


Figure 24(b) Q6 execution on CELL SPEs with different float types

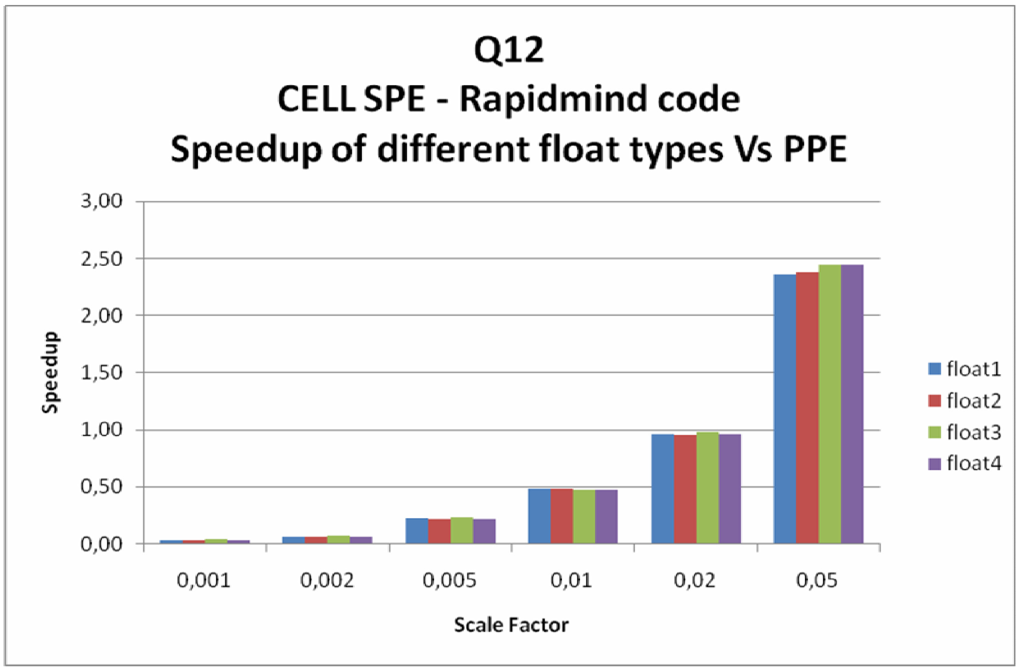


Figure 24(c) Q12 execution on CELL SPEs with different float types

6.3 Results on GPUs

For the experiments on GPU we used the CPU native C++ code executions as a baseline for comparing our results. As we expected, the 8800GT GPU performed better than 8500GT. That's related to the number of stream processors that graphic processor units have. GeForce 8800GT that we used has 7 times more stream processors than GeForce 8500GT.

Firstly, we did the execution for the most difficult and complex query; the Q3. In Figure 25(a) we can see the execution time as it was noted from the smaller scale execution datasets. For all the dataset, we can view the execution times in Figure 25(b). As for the speedup, compared to the CPU execution is shown in the graph Figure 26.

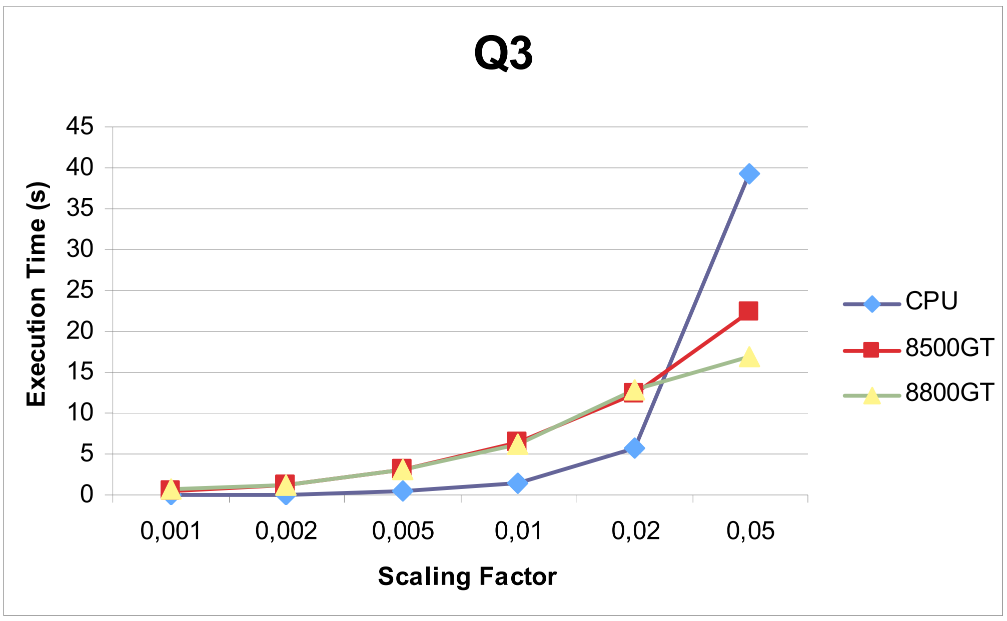


Figure 25(a): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q3 query (shows only smaller scales of data sizes)

From the figure above we notice that for smaller datasets of this query, the CPU non-parallel execution performed better than the GPUs. But then, there is a huge increase of CPU execution time as the data sizes grow. On the contrary, GPU execution times increase is more linear.

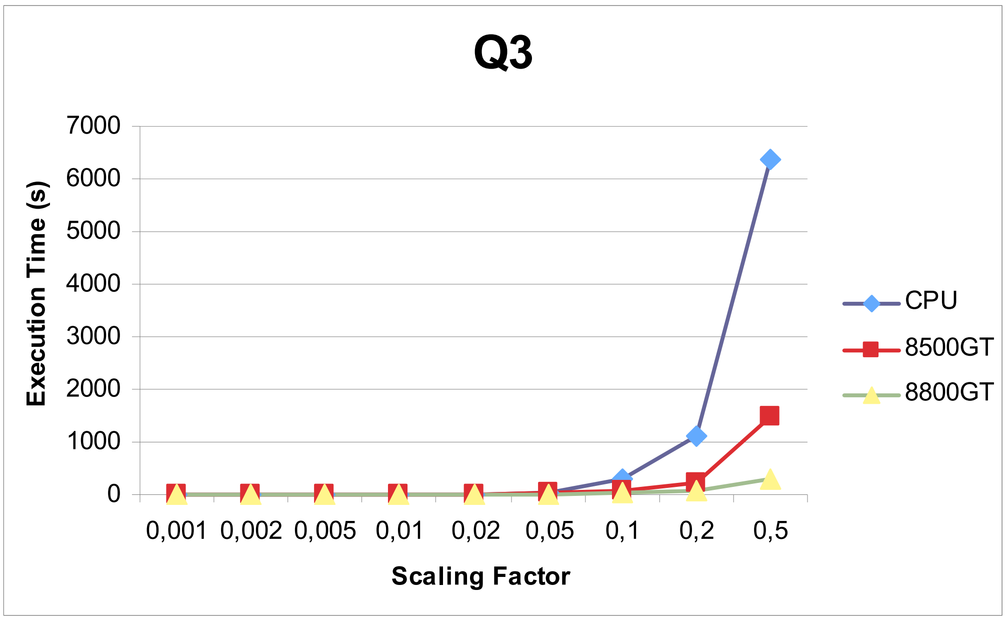


Figure 25(b): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q3 query (shows all scales of data sizes)

As descriped in previous chapter, Rapidmind uses data parallel model. This model scales easily to a large number of cores, but, in order to see the benefit of it we need to have sufficient data for processing. We can verify this statement by looking at the results of Q3. In Figure 26 is shown more clearly that for smaller scled of datasets, CPU outperforms both GPUs. But, for largest scales, Rapidmind execution on GPUs shows a true benefit.

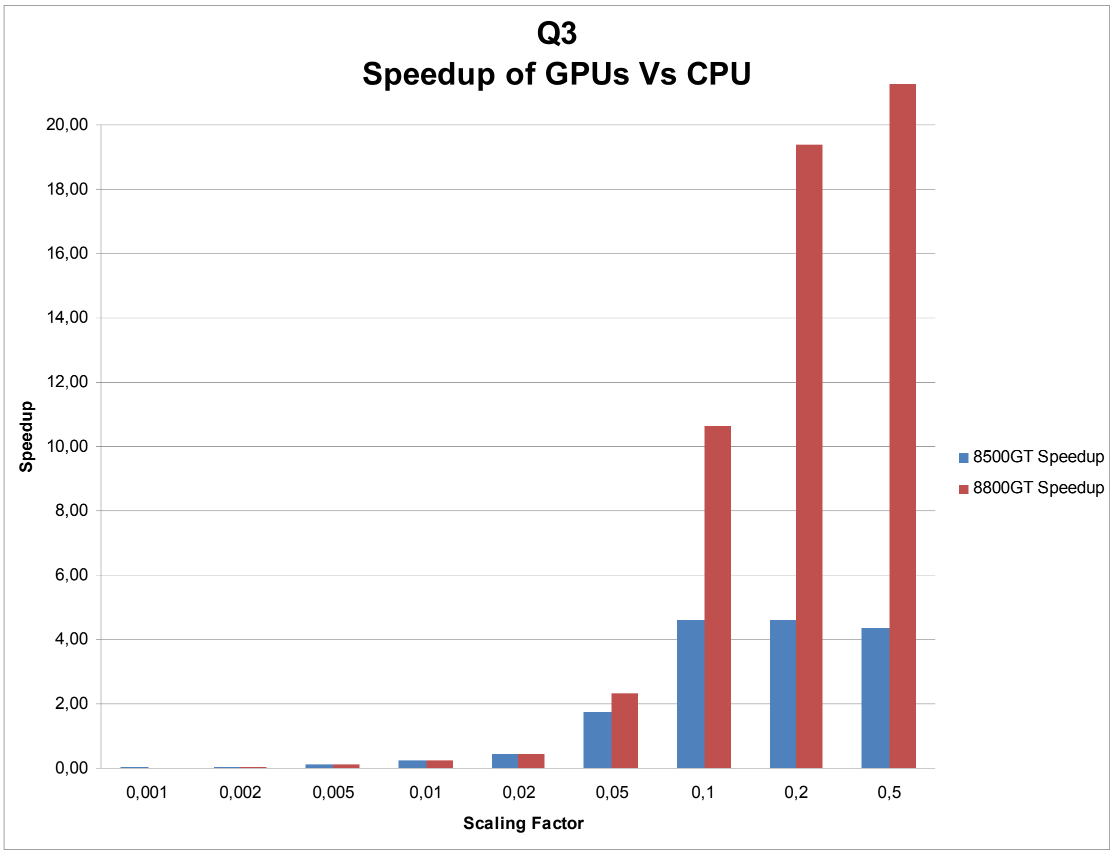


Figure 26: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q3 query

For this particular query we manage to get almost up to 5x speedup with the 8500GT GPU and 21x speedup with the 8800GT GPU. For smaller scales of datasets, both GPUs have almost the same performance. As the data grows, the speedup difference between them grows too. The 8500 GPU gives an almost constant speedup in the highest scales. On the contrary, for the 8800 GPU, the highest the data scale factor is, the highest the speedup we get.

Like we said before, in the Selection of float types section, we didn't run the Q3 query for larger scale factors than 0,5 because of the time needed for execution. Experiments for scale factor 1 and above needed about a day or more to be executed.

Our next query Q6, does not include any join operations. This query is the less complex from the three and therefore the less time-consuming. For this query the GPUs do not have a lot to offer in terms of execution time and speedup. As we can see in Figure 27, the CPU has the less execution time in the highest scales and the 8500 GPU has the highest.

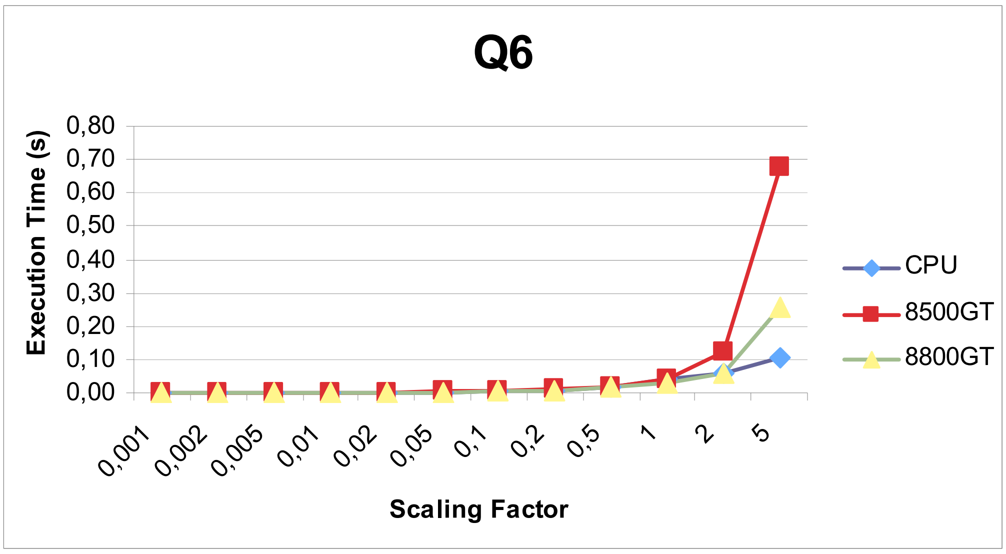


Figure 27: Execution time of CPU, 8500GT GPU and 8800GT GPU for Q6 query

From the speedups graph, in Figure 28, we can see that we have some speedup gained from 8800GT, in three cases; as scale 0.2, 0.5 and 1. this speedup has a peak value of 1.4x only. After the scale factor value of 1, the performance of GPU starts to fall off. This degradation in the performance of the GPU algorithm relies to the fact that due to large sizes of data tables, there are data transfer overheads that limit the GPU performance.

One important overhead is the amount of time needed for the GPU to refill its pipeline by transferring data from the system's main memory into the video memory. Another also significant overhead is the amount of time needed for the GPU to transfer data back to the system's main memory (Host-ReadBack procedure).

At scale factor 2, we only transfer a total of approxemately 250MB of actual data, in and out of the GPU memory. The GPU memory is 512MB large. Though the memory is double the size we actually need, the overheads seems to be a lot and cause the performance to fall. We cannot be very sure what Rapidmind does and how, in the background, but the fact that it needs to separate data to shared memories inside the GPU, might cause these time issues.

This experiment leads us to the conclusion that GPUs are not in favour of simple queries. Queries that include searching of only one table do not benefit from execution in the GPU. We can earn more in terms of performance when using GPUs in more complex queries execution, with join operations and use of more than one table.

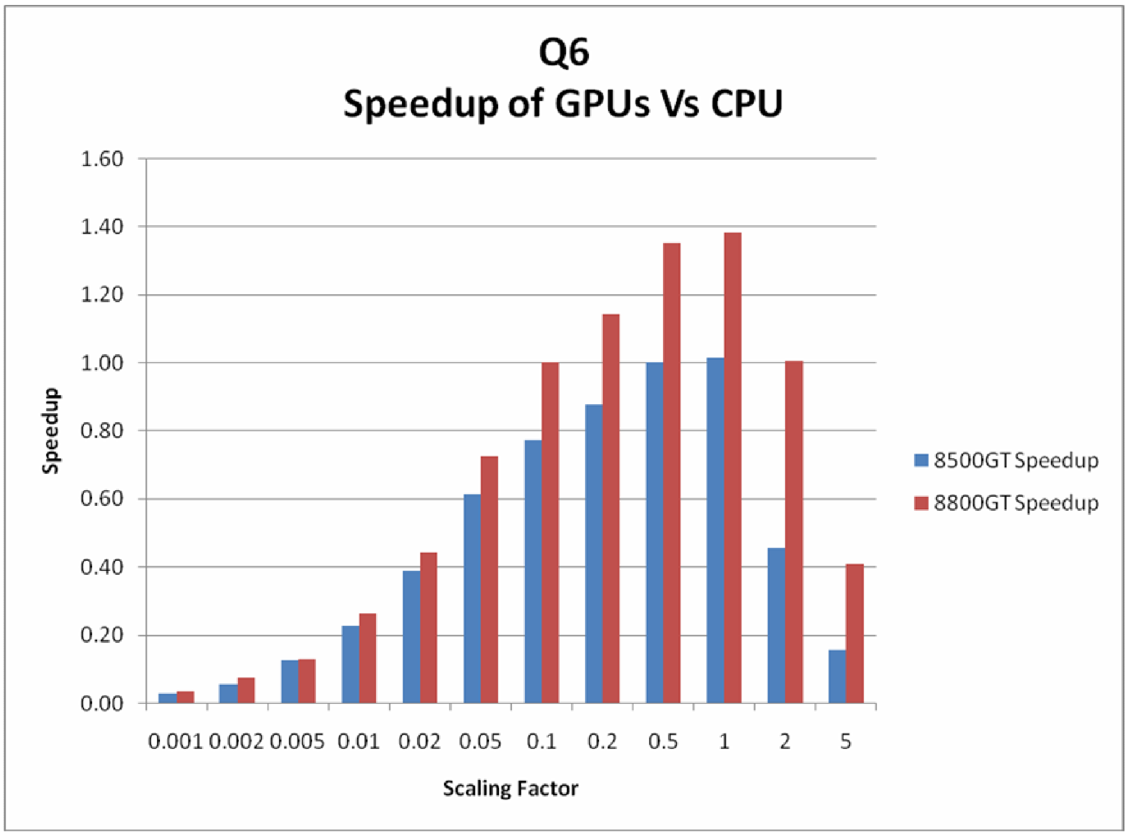


Figure 28: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q6 query

Q12 is the query with one join operation. For the smaller data sets, like in the previous queries, we don't notice any speedup from any of the two GPUs (Figure 29(a)). As the data sets grow speedup is gained, especially from 8800GT (Figure 29(b), 30). For the 8500GT we manage ta achieve a small speedup of approxemately 2x, that remains stable for the larger data sets, but for the 8800GT we achive a speedup of 12x in a constantly growing speedup graph (Figure 30).

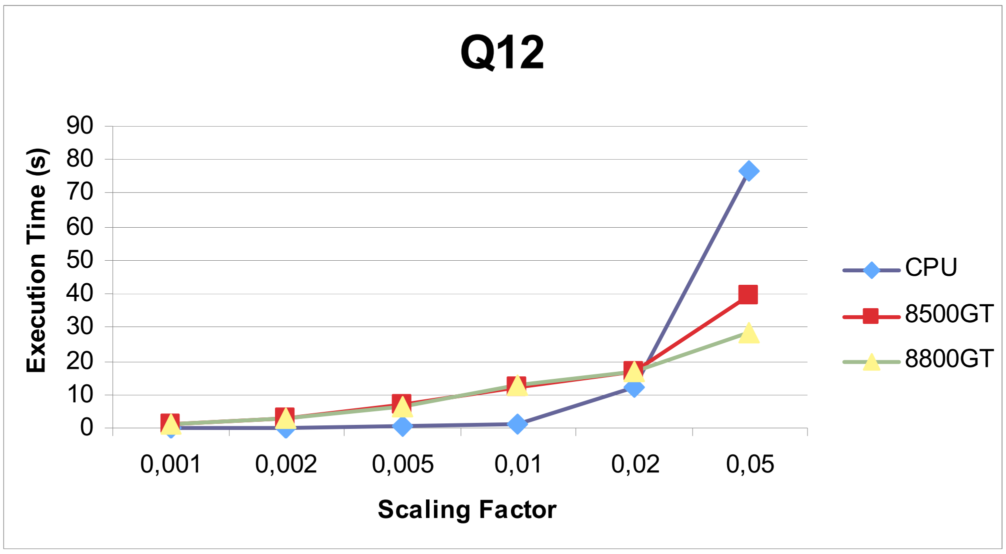


Figure 29(a): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q12 query (shows only small scales of data sizes)

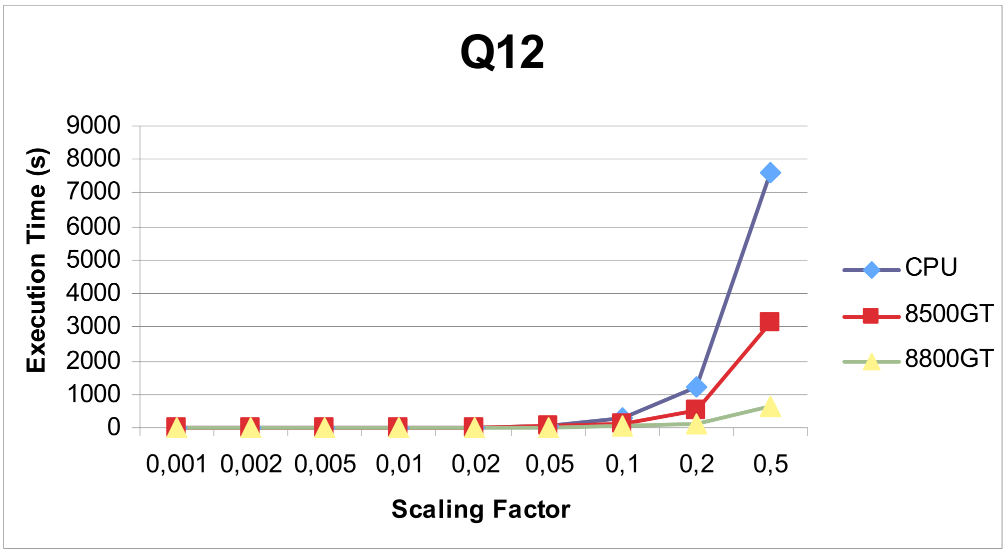


Figure 29(b): Execution time of CPU, 8500GT GPU and 8800GT GPU for Q12 query (shows all scales of data sizes)

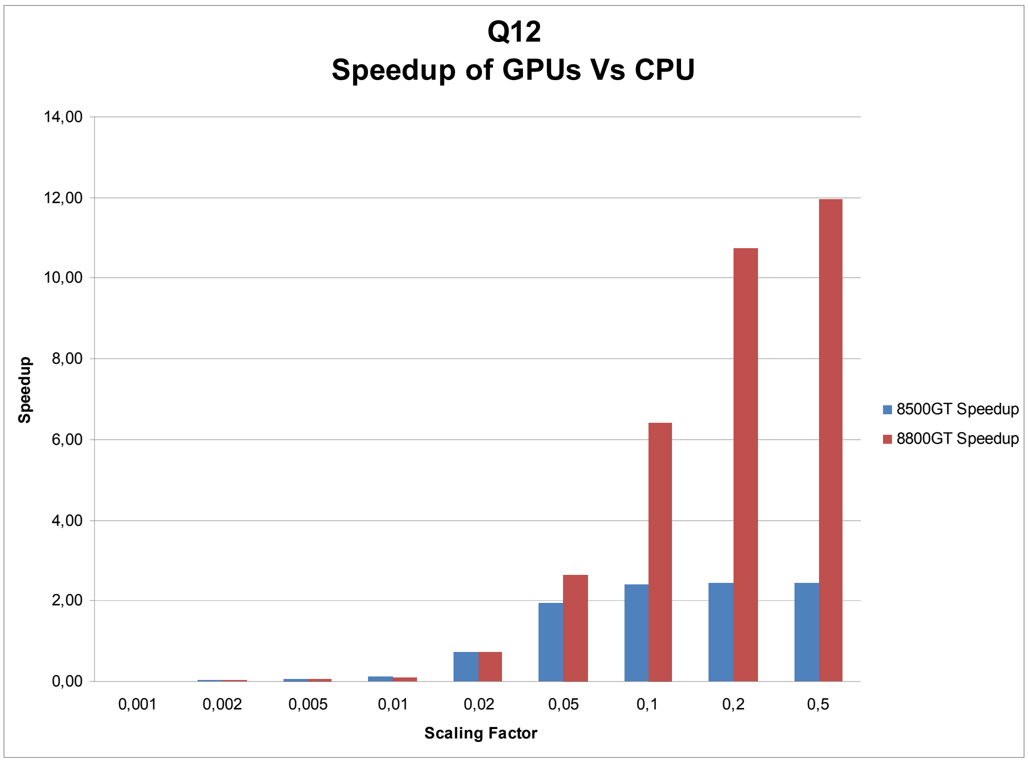


Figure 30: Speedup given from Rapidmind executions on 8500GT and 8800GT, using as a baseline the CPU execution times, in Q12 query

When looking at all the queries' speedups, we are able to say that GPUs do offer parallelism in database queries. The more the complex is the query, the best speedup we get from using GPUs. Also, in order to take advantage the large number of stream processors and the parallelization that they offer we have to send significant amount of data for processing.

6.4 Results on CELL

On the CELL we used two different execution codes. The first is using Rapidmind and is executed on CELL's SPE processors. For the CELL SPE execution on the Playstation 3, we can utilize only the six of the eight SPE processors. The second is a native C++ code and is used one the PPE processor of the CELL. Although the PPE is a dual thread processor, Rapidmind does not support multi-thread execution.

In the charts below, we will compare the SPE execution with both CPU and PPE execution. CPU and PPE execute the same code. We want to compare the SPE execution with the CPU, to see if the combination of Rapidmind and CELL's SPE processors is preferred in executing database queries. Also, we compare the SPE execution with the PPE execution. The PPE and the SPEs belong to the same system platform, the CELL Broadband engine, and also clocked at 3.2GHz. We do this comparison because we want to observe if Rapidmind's parallelization positively affects execution on CELL.

In the Figures below, we will see that when comparing the SPE execution on the CELL with the CPU execution on the desktop machine, we don't get any speedup. On the contrary, CPU outperforms the CELL SPEs in all three queries. But when comparing the CELL SPE execution with the CELL PPE, a speedup of 3x can be achieved from the Rapidmind execution on the SPEs.

When looking at Figure 31 for the Q3 query, we notice that the CELL SPE execution is more than 6.5x slower than the CPU execution. That is something that we didn't expect since we are using six 3.2GHz SPE stream processors from Playstation's 3 CELL. The CPU that we used for our experiments is clocked at 2.13GHz.

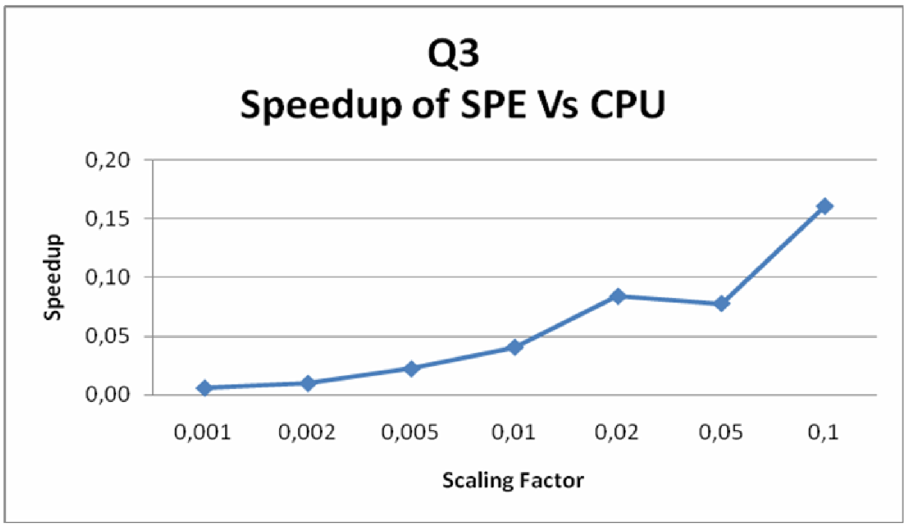


Figure 31: Speedup given from Rapidmind execution on CELL SPE processors, for Q3 query, using as a baseline the native C++ execution time on CPU

By looking at Figure 31, 32 and 33, we notice that all of the queries reach almost the same level of speedup, between 0,14 and 0,16.

Although, the largest the data sets are the better, it is important to note that we have a limit for this. We can see that in Figure 32. Speedup is keep growing as we go from smaller scales of data sets into larger, but after reaching the point where scaling factor value equals to 1, speedup starts to fall off. This is again due to the data transfer overheads in large size of data tables, like in the GPU. Overheads are caused from the need to transfer data from the main memory to each SPEs local store, and the opposite. Local stores are small and can not hold many data in their memory.

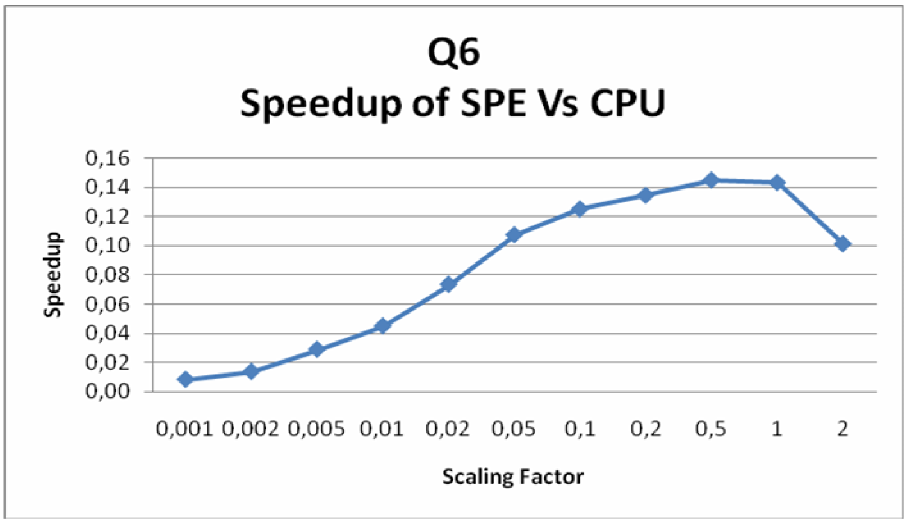


Figure 32: Speedup given from Rapidmind execution on CELL SPE processors, for Q6 query, using as a baseline the native C++ execution time on CPU

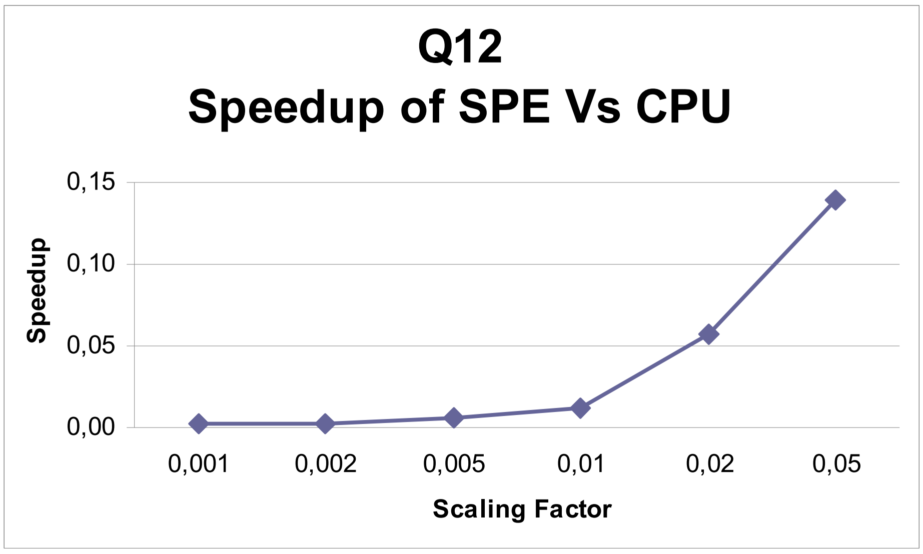


Figure 33: Speedup given from Rapidmind execution on CELL SPE processors, for Q12 query, using as a baseline the native C++ execution time on CPU

From the results above, we come to the conclusion that CELL SPE execution, although it is a parallel execution on six processors, is worse than a single execution of a common CPU processor.

Below we compare the SPE Rapidmind parallel execution with the single-core PPE processor execution.

When comparing the SPE with the PPE execution times, we get a speedup up to 2,7x in favor of SPE in Q3 query (Figure 34), a speedup of 1,6x in Q6 query (Figure 35) and 2,5x in Q12 query (Figure 36). As in the GPU case, the complex the query is the largest speedup we get.

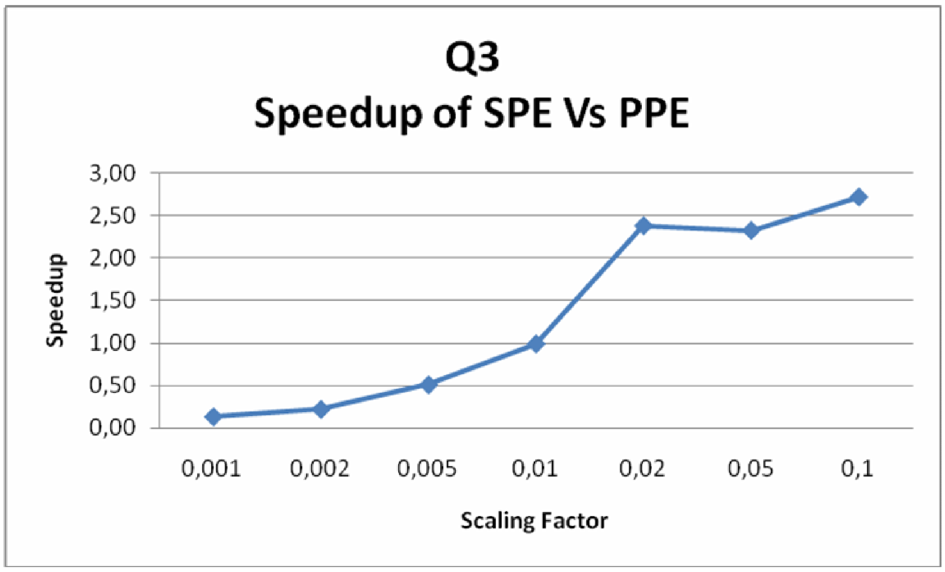


Figure 34: Speedup given from Rapidmind execution on CELL SPE processors, for Q3 query, using as a baseline the native C++ execution time on CELL PPE processor

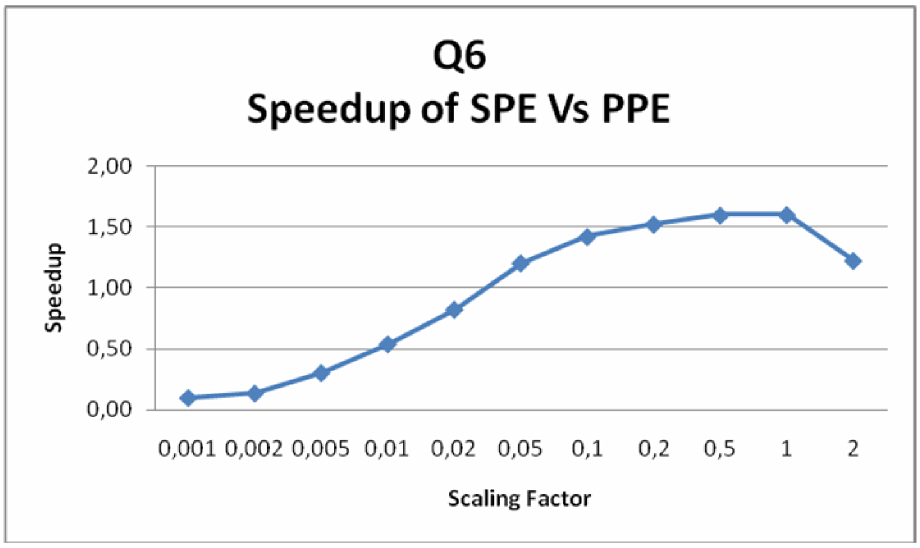


Figure 35: Speedup given from Rapidmind execution on CELL SPE processors, for Q6 query, using as a baseline the native C++ execution time on CELL PPE processor

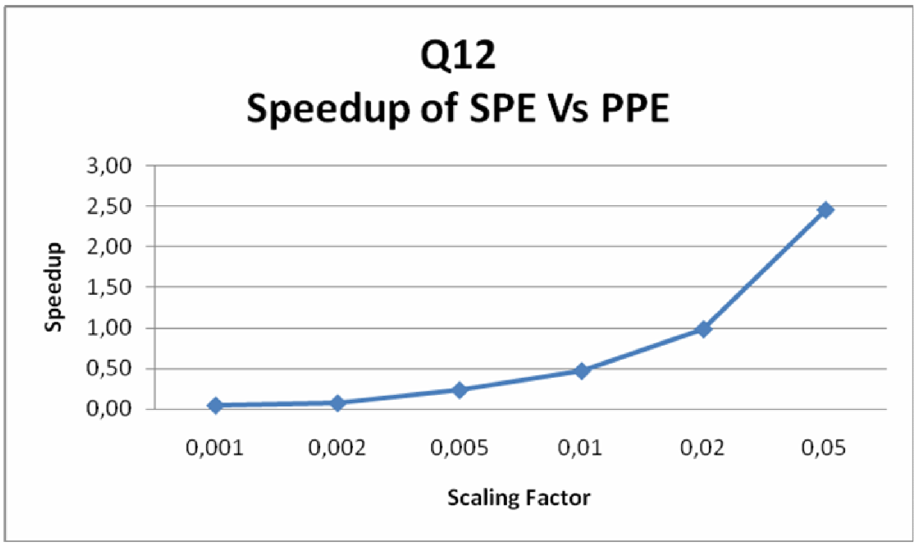


Figure 36: Speedup given from Rapidmind execution on CELL SPE processors, for Q12 query, using as a baseline the native C++ execution time on CELL PPE processor

The fact that we get almost 3x speedup from executing Rapidmind code on SPE processors, in comparison with C++ code execution on PPE, ensures us that we have parallel execution. But the speedup is not sufficient to overcome the CPU execution.

6.5 Comparison

In this section, we compare all the executions with the CPU native C++ single-core execution. The 8500GT GPU, 8800GT GPU and CELL SPE are parallel executions and are written in Rapidmind. The CELL PPE on the other hand is a native C++ code, single-core execution, like the CPU.

By taking a first look at the three speedup graphs (Figure 37(a), 38 and 39(a)), we notice that the highest speedup is taken from the 8800GT GPU. Second in the row comes the 8500GT GPU. As said before, for the GPU executions, the best speedup is achieved from the most complicated query (Q3) and the worst from the simplest one (Q6).

In the case of Q3 (Figure 37(a)) we managed getting 21.3x speedup, when comparing 8800GT to CPU execution. With the 8500GT we get up to 4,6x speedup. After the scale of 0,2 the speedup for 8500GT starts to fall. This does not happen with the 8800GT, where the speedup keeps increasing. As for the CELL's executions it's not so easy to see from this graph. For this reason we examine the Figure 37(b). Giving a first glance at this graph we notice that we do not have any speedup above 1.0 from any of both.

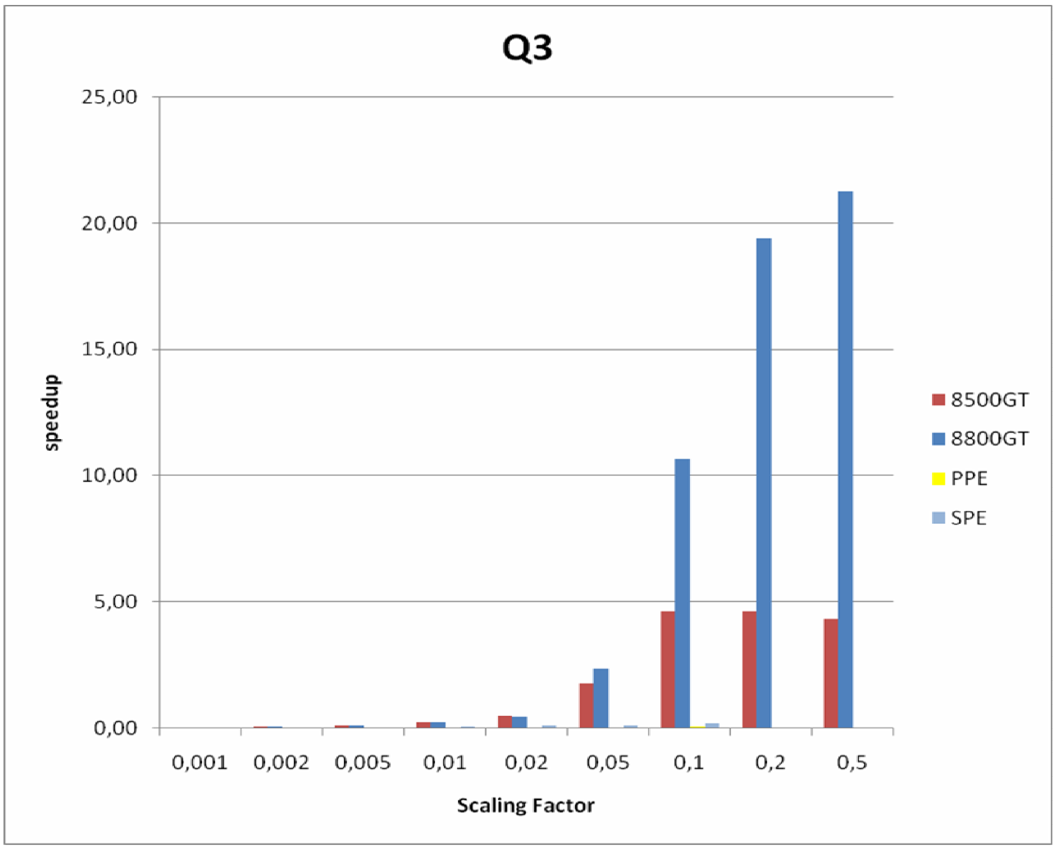


Figure 37(a): All speedups for Q3 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution

PPE is from 16 to 25 times slower than the CPU, in the cases that we examine. PPE manages up to only 6.25 slower than the CPU at the last scale.

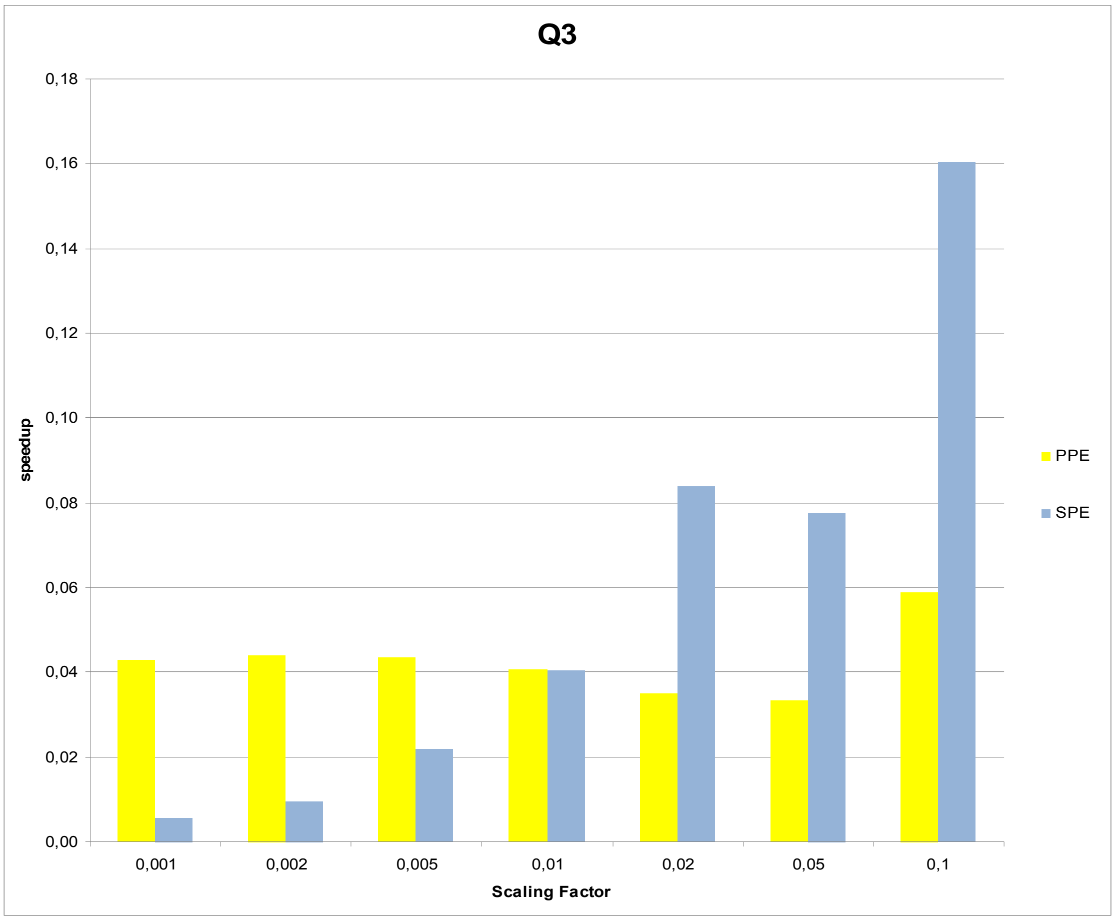


Figure 37(b): Speedups for Q3 query given CELL PPE and CELL SPE executions, using as a baseline the CPU execution

Q6 is the simplest query from the three. There are no joins in this one. Most obvious observation in Figure 38 is the highest speedup gained from 8800GT execution. It is only up to 1.5x. For PPE we have a speedup of up to 0,1x in comparison with CPU execution. For the SPE execution we have a speedup of up to 0,15x. For this query only the execution on 8800 GPU gains speedup against the CPU. The Rapidmind execution on 8500 GPU and CELL SPE does not give any positive results.

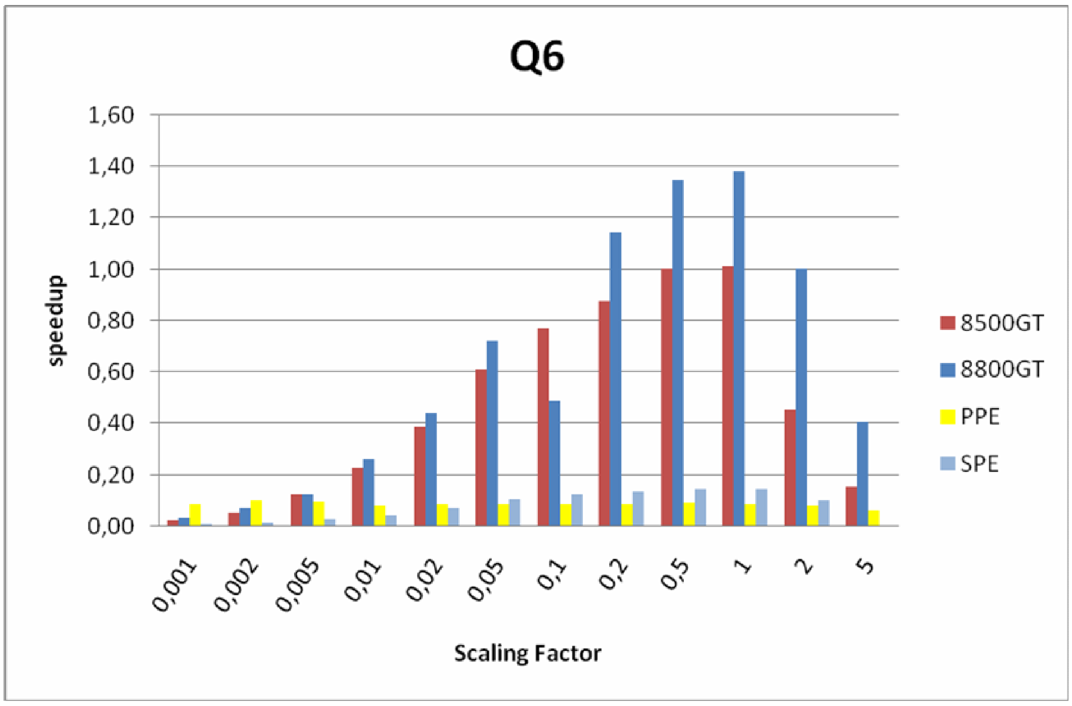


Figure 38: All speedups for Q6 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution

Q12 is a query with one join. More complicate than Q6 but more simple than Q3. We can observe the difference for 8800GT and 8500GT like in Q3, since the highest for 8800GT in Q12 is 12x (Figure 39(a)), which is the half than in Q3's case but 8.5x larger than in Q6's case.

In Figure 39(b), we can see the PPE and SPE speedup in comparison with the CPU. The PPE and the CPU execute the same single-core C++ code. Although PPE processor is clocked at 3.2GHz and the CPU at 2.13GHz, and we were expecting to have at least the same execution times, that does not happen. This is probably due to the fact that PPE is not meant to execute applications. It's role is to accomodate and control the Operating System, and also to allocate and send “work” to the SPE processors. The SPEs are the power of the CELL.

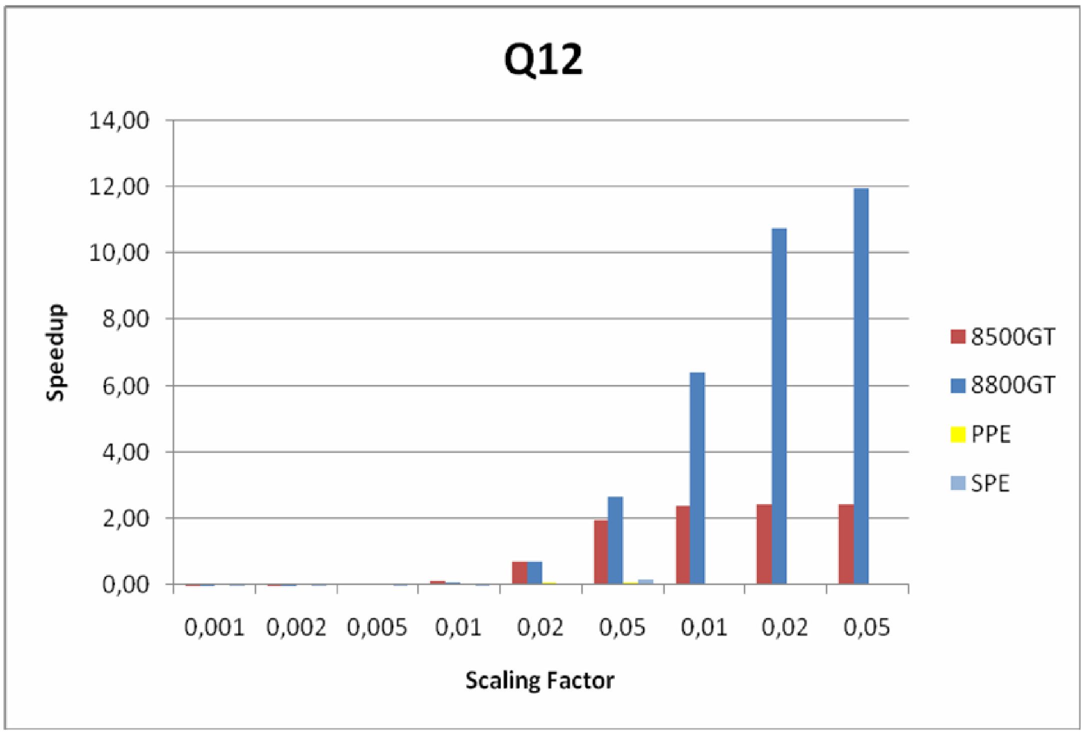


Figure 39(a): All speedups for Q12 query given from 8500GT GPU, 8800GT GPU, CELL PPE and CELL SPE executions, using as a baseline the CPU execution

There are a lot of factors that may have a role in the bad execution results. First of all, Rapidmind offers ease of programmability in return of abstraction and lack of control the hardware features. Thereby, we don’t know how and if Rapidmind utilizes the six available SPE cores of the CELL. Secondly, each SPE reads and writes only from and to its Local Store. The PPE is responsible to send data to the local store of each SPE. CELL does not have any cache to use. Again here, we cannot be very sure how Rapidmind creates the streams and if it splits them correctly and on time to each SPE’s local store. To this end, I think we need to do some more investigation for the CELL and see how other language platforms for parallel programming perform in it.

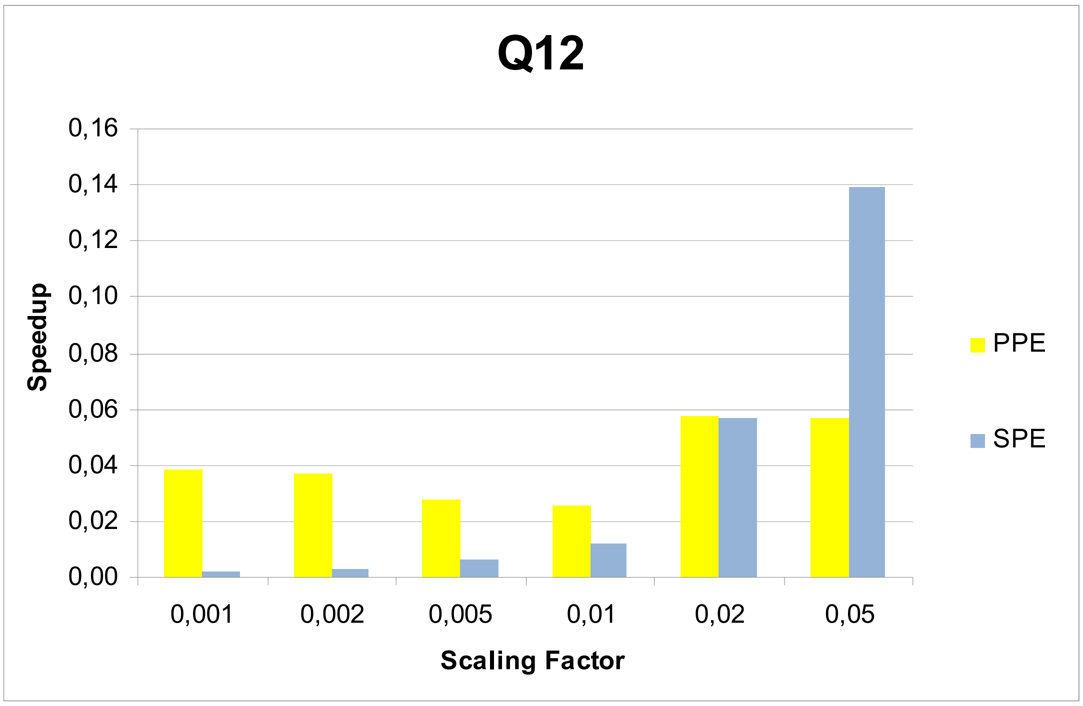


Figure 39(b): Speedups for Q12 query given CELL PPE and CELL SPE executions, using as a baseline the CPU execution

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Even though there were some programmable graphics processor units in the past, the whole procedure for creating and executing general purpose computations on the GPU was a very complex job. Also, the developer could use only a low-level language to do that, like OpenGL. There were no specialized languages or platforms to help you. Nowadays, the porting process is a lot easier task. The developer is now able to use high-level programming languages to do this job. These development platforms are used as a middleware between the high-level programming language and the actual low-level programming of the GPU. Rapidmind is an example of these development platforms.

We used Rapidmind in this thesis in order to exploit parallelization for three different decision support queries, on two different GPUs and on the CELL's SPE processors, using real TPC-H data ranging from a scale factor of 0,001 up to 5. After examining the results in the previous chapter we come to the conclusion that there is a large degree of parallelism offered by the GPUs, giving us very promising results in database query acceleration. Also, the more complex the query is, the more efficient the acceleration is and therefore, the more speedup we get by using GPU instead of CPU. Also larger scales and database sizes are in favor of the GPU. The 8800GT has shown four to six times better performance than the 8500GT. We can buy 8800GT for $260 and 8500GT for $60 in today's prices. So, we gain what we loose in cost, especially in cases with large size of data, where 8500GT cannot cope. In the case with the one join (Q12) the 8800GT shows more speedup difference from 8500GT than in the case with two joins (Q3). But, generally, the GPU gives more speedup, when comparing with the CPU, in the most difficult queries; Q3 and then Q12. We achieve speedup up to 21x with the use of Rapidmind on the 8800GT GPU.

The CELL did not show any really promising results when compared to the CPU. Though, we have to note some interesting observations. In regards to the single-core PPE execution, we get a quite good speedup from the multi-core SPE execution. If we study the fact that we are able to use the six of the eight SPE processors for development purposes and the speedup gained from Rapidmind execution on SPEs, versus C++ execution on PPE, is almost 3.0x, then we can say that we have a good degree of parallelism. But, with the results seen and in respect to the CPU execution times, we don't suggest the use of the $400 PS3 for database query executions. Although, we believe that before judging the Cell as "non gainful" in databases, we should do some more tests with it.

Based on this study, we can conclude that GPU seems to be the best decision to go. But again, it is not appropriate for every kind of query. It is beneficiary when having large data sizes and complex queries. The more complex, the best speedup we get. We could adjust applications to use the platform that suites better for them each time.

Even if Rapidmind shows interesting results in the GPU, we still feel that abstraction is not very good when it has to do with parallel programming. Rapidmind “hides” a lot of information from the developer. It does not give any control to the programmer on how the program will be shared and executed into the available processors. Abstraction seems to work well in the GPU but we cannot say the same for the CELL. There are studies that suggest some techniques for better performance when programming for the CELL, like overlapping data fetches, but there were not able to be implemented by using Rapidmind. The lack of control for some hardware features, is one trade-off for the easy programmability that Rapidmind offers.

7.2 Future Work

As we already know, the GPU hardware can only do simple operations like summation, subtraction, multiplication, etc, but he CELL SPE can do more complicate operations like root, power, sine, cosine, tangent, etc. We should try different kinds of complex operations in the CELL SPEs and compare them with the same operations on the CPU. Also, we could execute the same queries on the CELL using another language platform or pthreads, and compare the results with those of Rapidmind. We need to perform more research and tests for the CELL in order to understand the true reasons for the low performance results.

By using Rapidmind's latest release, the Development Platform Version 3.0, we could also execute the Rapidmind code on multi-core CPU, since it supports the CPU backend. With the version of Rapidmind that we used in this thesis, we didn't have this capability.

Also, we could use the NVIDIA proprietary programming model, CUDA, to implement the same queries and execute them on the same GPUs. Then we could compare the results with those of the Rapidmind on the same hardware. CUDA is specialized for general purpose applications on graphics processor units. It is more complicated in programming than Rapidmind but gives more control in the execution. I would expect CUDA to give better results with the use of the same GeForce GPUs that I used in my thesis. Rapidmind is very high-level and most of parallelization takes place behind the scenes. But, CUDA allows for lower-level control of the GPU and is more flexible. Also, CUDA can make use of multiple GPUs. Rapidmind can take advantage only on backend each time.

Another interesting issue we could investigate is the benefit of the “automatic” behavior of Rapidmind. With Rapidmind we have no control of the backend. If we implement these queries in CUDA (for GPU) and CELL SDK native language (for the CELL) we could compare the executions with those of Rapidmind, and see if the automatic behavior has any trade-offs in terms of performance.

REFERENCES

1. *AMD Completes ATI Acquisition and Creates Processing Powerhouse.* AMD, October 2006.

2. Artemiou, A. (2007). *Exploiting the Graphics Processing Unit for efficient execution of Decision Support System Queries. MSc Thesis.* Nicosia: University of Cyprus.

3. Blagojevic, F., Nikopoulos, D. S., Stamatakis, A., & Antonopoulos, C. D. (2006). *Dynamic Multigrain Parallelization on the Cell Broadband Engine.*

4. Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., & Bosilca, G. (May 7, 2007). *SCOP3: A Rough Guide to Scientific Computing On the PlayStation 3.* Innovative Computing Laboratory, University of Tennessee Knoxville.

5. Flachs, B., Asano, S., & Dhong, S. (February 2005). A streaming processor unit for cell processor. In *ISSCC Dig. Tech. Papers* (pp. 134-135).

6. Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., & Manocha, D. (2003). *Fast Computation of Database Operations using Graphics Processors.* University of North Carolina at Chapel Hill.

7. Hofstee, P. H. (2005). Power Efficient Processor Architecture and the Cell Processor. *11th Int'l Symposium on High-Performance Computer Architecture.*

8. *Exploratory Stream Processing System*. Retrieved December 8, 2008, from IBM: http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html. (2007).

9. IBM to show Stream Computing System. (2007, June 19). *New York Times* .

10. McCool, M. D. (2006). Data-Parallel Programming on the CELL/BE and the GPU using Rapidmind Development Platform. *GPSx Multicore Applications.* Santa Clara: Rapidmind Inc.

11. Monteyne, M. (February 2008). *RapidMind White Paper: RapidMind Multi-Core Development Platform.* RapidMind Inc.

12. *GeForce 8500 - Technical Specifications*. Retrieved January 20, 2009, from NVIDIA: http://www.nvidia.com/object/geforce_8500.html. (2007).

13. *GeForce 8800 - Technical Specifications*. Retrieved January 20, 2009, from NVIDIA: http://www.nvidia.com/object/geforce_8800.html. (2007).

14. Technical Briefs: *NVIDIA GeForce 8800 GPU Architecture Overview.* NVIDIA, November 2006.

15. *The CUDA Compiler Driver NVCC. NVIDIA, 2007.*

16. Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krugen, J., Lefohn, A. E., et al. (2007). A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum, Volume 26, number 1* , pp. 80-113.

17. Pham, D., Asano, S., & Bollier, M. (February 2005). The design and implementation of a first-generation cell processor. In *ISSCC Dig. Tech. Papers* (pp. 184-185).

18. *Rapidmind*. Retrieved May 24, 2008, from RapidMind Blog: http://blogs.rapidmind.com. (2008).

19. Rumpf, M., & Strzodka, R. (2005). Graphics Processor Units: New Prospects for Parallel Computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers.* A. M. Bruaset and A. Tveito.

20. Samuel, F., Berbay, J., & McCool, M. (2007). *Implementation of Parallel Set Intersection for Keyward Search using RapidMind.* Univercity of Waterloo, Canada.

21. Stokes, J. (2006, September 18). PeakStream unveils multicore and CPU/GPU programming solution.

22. Trancoso, P., & Charalambous, M. (2005). Exploring the Graphics Processor Performance for General Purpose Applications.

23. Williams, S., Shalf, J., & Oliker, L. (2006). *Scientific Computing Kernels on the Cell Processor.* Berkeley: Computing Research Division, Lawrence Berkeley National Laboratory.

24. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., & Yelick, K. (2006). *The Potential of the Cell Processor for Scientific Computing.* Berkeley: Computational Research Division, Lawrence Berkeley National Library.

25. McCool, M. D., Qin, Z., & Popa, T. S. Shader metaprogramming. *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware*, (pp. 57-68). Aire-la-Ville, Switzerland.

26. Programming Guide: *NVIDIA CUDA Compute Unified Device Architecture* (Version 1.1). NVIDIA, November 2007.

27. I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian. (2008) BrookGPU. Retrieved from http://graphics.stanford.edu/projects/brookgpu/

28. *Writing Applications for the GPU Using the RapidMind™ Development Platform*. Rapidmind, 2006.

29. *CELL/BE Porting and Tuning with RapidMind: A Case Study*. Rapidmind, 2006.

30. User Guide: RapidMind Multi-Core Software Platform. Rapidmind, 2007.

31. Transaction Processing Performance Council (TPC). [April 2006]. *TPC Benchmark H (Decision Support)*. Standard Specification revision 2.6.0.

32. McCool Michael, Wadleigh Kevin, Henderson Brent, Lin Hsin-Ying. [October 2006]. Performance Evaluation of GPUs Using the RapidMind Development Platform

33. K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, Dinesh Manocha. [2005]. A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. UNC Tech. Report.

34. Naga K. Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manocha. [2006] GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. ACM SIGMOD.

35. Nsort: Fast parallel sorting. Retrieved from http://www.ordinal.com/ in February 2009

36. Playstation 3. Retrieved from http://en.wikipedia.org/wiki/Playstation_3 in January 2009

37. "Sony PlayStation 3 Cell Processor". North Carolina State University. Retrieved from http://moss.csc.ncsu.edu/~mueller/cluster/ps3/ in January 2009.

38. Martin Linklater. "Optimizing Cell Code". Game Developer Magazine, April 2007: pp. 15–18.

39. Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In Michael Franklin, Bongki Moon, and Anastassia Ailamaki, editors, Proceedings of the ACM SIGMOD International Conference On Management of Data, June 3–6, 2002, Madison, WI, USA, pages 145–156, New York, NY 10036, USA, 2002. ACM Press. ACM order number 475020

40. Jingren Zhou and Kenneth A. Ross. Buffering database operations for enhanced instruction cache performance. In SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pages 191–202. ACM Press, 2004.

41. Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. [1999]. In Proceedings of the Twenty-fifth International Conference on Very Large Databases, pages 78–89.

42. Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings, pages 510–521, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

43. Samuel, F., Barbay, J., McCool, M. (2007). *Implementation of Parallel Set Intersection for Keyword Search using Rapidmind.* University of Waterloo, Canada: Technical Report CS-2007-12.

44. Beck, A. (1996, August). *Visual Parallel Programming May Come of Age with CODE.* Retrieved May 2009, from The University of Texas in Austin: http://www.cs.utexas.edu/users/code/CODE-HPCwire-article.html

45. Cattell, R. G. (1988). Object-oriented DBMS performance measurement. In S. B. Heidelberg, *Advances in Object-Oriented Database Systems* (pp. 364-367). Springer Berlin / Heidelberg.

46. Fraternali, P., & Paolini, P. (2000, October). Model-driven development of Web applications: the AutoWeb System. *ACM Transactions on Information Systems (TOIS) , Volume 18* (Issue 4), pp. 323-382.

47. Nah, F. F.-H. (2002). *Enterprise resource planning solutions and management.* Idea Group (IGI).

48. Roussopoulos, N., & Delis, A. (1991). Modern client-server DBMS architectures. *ACM SIGMOD Record* , pp. 52-61.

49. Wolfe, M. (2008, December 8). Compilers and More: A GPU and Accelerator.

50. Zukowski, M. (2005). Improving I/O Bandwidth for Data-Intensive.

51. C. Xavier, Sundararaja S. Iyengar (1998). *Introduction to Parallel Algorithms.* Wiley-IEEE.

52. M. Auguin, F. Boeri, J. P. Dalban, A. Vincent-Carrefour (1987). Experience using a SIMD/SPMD multiprocessor architecture. *Microprocessing and Microprogramming , Volume 21* (Issue 1-5), pages 171 - 177.

53. *AMD Completes ATI Acquisition and Creates Processing Powerhouse.* AMD Press Release, October 2006.

54. *Cell Broadband Engine Programming Handbook*, Version 1.0. IBM, April 2006.

55. *Writing Applications for the Cell BE Processsor Using the RapidMind™ Development Platform*. Rapidmind, 2006.

56. N. Goodnight, R. Wang, G. Humphreys. (2005). Computation on programmable graphics hardware. *Computer Graphics and Applications, IEEE , Volume 25* (Issue 5), pages 12-15.

57. H. Servat, C. Gonzalez, X. Aguilar, D. Cabrera, D. Jimenez (2007). Drug design on the cell broadband engine. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques,* pages 425.

58. B. Gedik, P. Yu, R. Bordawekar (2007). Executing stream joins on the cell processor. *VLDB '01: Proceedings of the 33rd International Conference on Very Large Data Bases,* pages 363-374.

59. T. Chen, R. Raghavan, J. Dale, E. Iwata (2005). Cell Broadband Engine Architecture and its First Implementation. IBM developerWorks.

60. A. E. Eichenberger et. al. (2005). Optimizing Compiler for a Cell Processor. *Parallel Architectures and Compilation Techniques.*

61. F. Petrini, G. Fossum, M. Kistler, M. Perrone. Multicore Suprises: Lesson Learned from Optimizing Sweep3D on the Cell Broadband Engine.

62. M. Livny, J. Basney, R. Raman, T. Tannenbaum (1997). Mechanisms for High Throughput Computing.

63. J. A. Hennessy, D. Goldberg (1996). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

64. Hruska, J. (2008, November 14). *AMD Fusion now pushed back to 2011*. Retrieved April 20, 2009, from Ars OpenForum 3.0b: http://arstechnica.com/old/content/2008/11/amd-fusion-now-pushed-back-to-2011.ars

65. David Tarditi, S. P. (December 2005). *Accelerator: simplified programming of graphics processing units for generalpurpose.* Microsoft Research Technical Report MSR-TR-2005-184.

66. *UltraSPARC Processors*. Retrieved April 20, 2009, from Sun Microsystems: http://www.sun.com/processors/UltraSPARC-T1/index.xml

67. *POWER4 System Microarchitecture*. Retrieved April 24, 2009, from IBM: http://www-03.ibm.com/systems/p/hardware/whitepapers/power4.html

68. *POWER5 Architecture*. Retrieved April 24, 2009, from IBM: http://www.ibm.com/developerworks/wikis/display/WikiPtype/POWER5+Architecture

69. John D. Owens (March 2005). *Streaming Architectures and Technology Trends*, in GPU Gems 2. Addison-Wesley.

70. *AltiVec Technology Programming Interface Manual.* Freescale Semiconductor, 1999.

71. Kleinrock, L., Fellow, IEEE, & Huang, J.-H. (1992). *On Parallel Processing Systems: Amdahl's Law Generalized and Some results on Optimal Design*. IEEE Transactions on Software Engineering , Volume 18 (No. 5).