NTNU
Norwegian University of
Science and Technology

Throughput Computing on Future GPUs

Rune Johan Hovland

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The HPC community has devoted a great deal of attention to the general-purpose capabilities of the Graphics Processing Unit (GPU). More recently, the potential of the GPU as a computational device has also received attention in the Information Retrieval community. Large data volumes and a focus on throughput are characteristic for applications within this area. A search engine must handle large amounts of data, and most of this is fetched from disk, making data access a substantial cost for these applications. The lack of streaming capabilities between host and GPUs may limit the potential benefits of using GPUs in such applications.

This thesis should analyze GPU systems with respect to applications with large data volumes, and suggest any improvements which can benefit these types of applications. Developing a theoretical model for the improvements and if possible show gains for real-world applications, would be an essential part of this work.

Assignment given: 15. January 2009
Supervisor: Anne Cathrine Elster, IDI

Abstract

The general-purpose computing capabilities of the Graphics Processing Unit (GPU) have recently been given a great deal of attention by the High-Performance Computing (HPC) community. By allowing massively parallel applications to run efficiently on commodity graphics cards, "personal supercomputers" are now available in desktop versions at a low price. For some applications, speedups of 70 times that of a single CPU implementation have been achieved. Among the most popular GPUs are those based on the NVIDIA Tesla Architecture which allows relatively easy development of GPU applications using the NVIDIA CUDA programming environment.

While the GPU is gaining interest in the HPC community, others are more reluctant to embrace the GPU as a computational device. The focus on throughput and large data volumes separates Information Retrieval (IR) from HPC, since for IR it is critical to process large amounts of data efficiently, a task which the GPU currently does not excel at. Only recently has the IR community begun to explore the possibilities, and an implementation of a search engine for the GPU was published recently in April 2009.

This thesis analyzes how GPUs can be improved to better suit large data volume applications. Current graphics cards have a bottleneck regarding the transfer of data between the host and the GPU. One approach to resolve this bottleneck is to include the host memory as part of the GPUs' memory hierarchy. We develop a theoretical model, and based on this model, the expected performance improvement for high data volume applications are shown for both computationally-bound and data transfer-bound applications. The performance improvement for an existing search engine is also given based on the theoretical model. For this case, the improvements would result in a speedup between 1.389 and 1.874 for the various query-types supported by the search engine.

Preface

This master thesis was written at the HPC-group at Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). During the process of both the master-project, and during this thesis, I have experienced the benefits and limitations with programming on the GPU. Initially, I wanted to create a full-scale search engine on the GPU, and to some extent this has been done, but the pitfalls and difficulties with GPU programming made me focus more on what could be done to improve this process. This thesis is an attempt to give recommendations for improvements which would benefit those developing applications with large data volumes.

To be able to complete this thesis, there are many people who deserve my acknowledgement. First and foremost I would like to thank Associate Professor Anne C. Elster, who through her guidance and supervision made this thesis possible. Had it not been for her dedication towards building the HPC-lab at NTNU, this report would not have been realized. In that regard, I must also extend my gratitude to NVIDIA Corporation for donating most of the graphics cards used throughout this report, through Elster's membership in their Professor Affiliates Program. Associate Professor Magnus Lie Hetland and Øystein Torbjørnsen with Fast Search and Transfer have also been of valuable assistance in pointing out focus areas. Jan Christian Meyer has been a great assistance in the writing process of this thesis. I would also like to thank Rune Erlend Jensen for beating even Google in quickly answering my many small questions. He and the rest of the HPC-group at NTNU has been a valuable source of information and support during this period. Finally, I would like to thank Eirik Ola Aksnes for enduring my many hours of procrastination in his office.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last years, the High Performance Computation (HPC) community has devoted a great deal of attention to the Graphical Processing Unit (GPU). Its general-purpose capabilities has given developers access to cheap *supercomputers* in their own desktop computers. The GPU is capable of solving many computational challenges faster than modern CPUs due to its large number of processors. Most HPC applications are computationally intensive, which are tasks perfect for the GPU. This aspect has also been emphasized by NVIDIA with their release of their *personal supercomputer*[1] which consists of four NVIDIA Tesla c1060 GPUs. This supercomputer is able to deliver a performance close to 4 Teraflops. For comparison it can be mentioned that as of November 2008 the 500th fastest supercomputer delivers 12.59 Teraflops[2]. To exploit the GPUs fully requires problems with high computational density allowing the cores to be fully occupied. For problems less computationally intensive or bound by data bandwidths these *supercomputers* may not be the best fit.

The field of Information Retrieval is a field with large data volumes and computationally lighter applications than traditional HPC. For this reason the GPU has not yet gained the status as a suited computational platform. However, the WestLab[3] research group at the New York University has created a fully functional search engine [10] on the GPU with improved performance. While a search engine often contains complex ranking schemes and other calculations, it is still a application bound by the large data volumes stored in a search index.

Handling large data volumes on the GPU is not trivial on current GPUs, as there are certain limitations when developing such applications. All data

[1] http://www.nvidia.com/object/io_1227008280995.html
[2] http://www.top500.org
[3] Web Exploration and Search Technology Lab (http://cis.poly.edu/westlab/)

used on the GPU must be physically present on the GPU. This requires a data copy and must be performed before the application can start. Results from the GPU needed on the host must be copied back to the host after the completed calculation. When handling large data volumes this can cause critical delays which seriously impair the GPUs ability to compete with the CPU.By introducing full streaming capabilities the GPU might be able to remove most of these delays and efficiently handle large data volumes.

This thesis will suggest improvements that can be made to enable future GPUs to efficiently support large data volumes, and ease the development process of such applications. The NVIDIA Tesla Architecture and the NVIDIA CUDA programming extension will be used as the representative of current GPUs. The search engine created by WestLab is used as an example of a large data volume application.

1.1 Outline

Chapter 2 gives the necessary background to understand the GPU and its usage in general-purpose programming. This chapter also focuses on the performance characteristics of the NVIDIA Tesla Architecture which is the architecture that GPUs from NVIDIA are based on.

Chapter 3 starts with a suggestion to enlarge the memory hierarchy to also include host-memory to allow the GPU to access this while executing the GPU-program and thereby enables interleaving of data transfer and computations. This is then analyzed through a theoretical model and the improvement expected by this alteration is given. Finally, several other small changes are suggested to improve the usability of NVIDIA CUDA in large data volume applications.

Chapter 4 uses an existing CPU-GPU search engine [10] as a case study for the improvements suggested in this thesis. By extracting benchmarks given in WestLab's article [10], the expected performance improvement is calculated using the theoretical model.

Chapter 5 gives the concluding remarks of this thesis and summarizes the suggested improvements and their benefits for large data volume applications on GPUs.

Chapter 2

The Graphics Processing Unit

The Graphics Processing Unit (GPU) has over the last decade been used to show computer graphics and other graphical related tasks. In the last couple of years, the GPU has proven its capabilities of performing general-purpose calculations as well. This chapter will give a brief introduction to the GPU and its architecture, before the need for GPUs for general-purpose calculations is discussed. Finally, this chapter will give an overview of the performance characteristics of the NVIDIA Tesla Architecture for GPUs.

2.1 The Graphics Processing Unit

The GPU has recently been devoted interest in the HPC community due to its possibility to perform general-purpose calculations using the graphics pipeline. It is only recently that this has been possible to do efficiently. This achievement is primarily due to the introduction of the Unified Shader Model [6] which made the way for NVIDIAs Tesla Architecture [24, 23] and ATIs Close To Metal [1].

2.1.1 Historic Retrospect

The GPU started out in the 1960s as a processing unit dedicated to rendering graphics [7]. Graphically demanding applications such as Computer-Aided Design (CAD) and computer games required more computing power to draw their graphics than the computer could provide, and thus, dedicated graphics systems were introduced. These systems also included specialized hardware for converting 3-dimensional geometry into 2-dimensional images. When the raster displays later replaced the previously used vector-displays, the GPU had to include functionality to convert vector based graphics into

pixels. By the beginning of the 1980s, add-in graphics cards began to target personal computers and the video card was created. A decade later, the graphics cards had become more common in personal computers, making the need for standardization of the cards a necessity. The OpenGL[1] and Direct 3D[2] APIs came as a response to this need, allowing the developers to only design for a given API instead of a wide range of graphic devices. Also introduced in the 1990s was the specialized hardware to accelerate common multimedia applications such as video playback. This constant demand for new capabilities increased the graphics pipeline, making it a large and ineffective computational platform [7]. Even though the GPU at that time had a large throughput, the specialized pipeline made the programs suffer from bottlenecks as the pipeline could not efficiently adjust to the various needs of graphical applications.

The fixed pipeline in the GPU should soon prove to be a bottleneck for the graphics developers. Each application would have different need for the various shaders provided in the pipeline, leaving computing resources unused. As especially games requiring an ever increasing performance, this is not ideal.

2.1.2 Unified Shader Model

Microsoft introduced Direct3D 10 [6] in 2006, which attempted to free the GPU of its fixed-pipeline challenges. A key part of this specification was Shader Model 4.0, which defined how each type of shader in the graphics pipeline should be implemented. Instead of continuing the trend from earlier years with different hardware for each shader, it was decided to use a common core for all shaders. This was called the Unified Shader Model. Microsoft allowed the GPU-vendors to create a pipeline which used the same components for all the shader-stages by basing each shader on the same core. A GPU-architecture structured this way is called an Unifided Shader Architecture. By doing so, the computational power given to each stage in the pipeline could be balanced based on the computational requirements of the program, instead of being fixed by the GPU vendor.

2.1.3 GPU vendors

During the history of GPUs there has been many vendors providing GPUs. However in today's market, there are only two main competitors left; AMDs

[1] http://www.opengl.org/

[2] Part of DirectX (http://www.microsoft.com/directx)

subdivision ATI[3] and the NVIDIA Corporation[4]. In addition to these two, the Intel Corporation, Matrox, ARM and more are also manufacturing GPUs but within other marked segments than the one required for the tasks covered in this thesis. While using different architectures for their GPUs both ATI and NVIDIA mainly support the same features and both have general computation capabilities through proprietary languages. Performance wise comparison between the two GPU architectures is an impossible task, as the two architectures excel in different areas, and constantly outperform each other as newer models based on the architecture is launched. The fact that proprietary languages have to be used to perform general purpose calculation also complicates the comparison, leaving this a matter of personal preferences and outside the scope of this thesis.

2.1.4 NVIDIA Tesla Architecture

The NVIDIA Tesla Architecture [23] is the approach NVIDIA chose in order to implement the Unified Shader Architecture. It is an architecture which consists of a varying number of Streaming Multiprocessors (SM) which is a computational unit capable of performing all shader operations, as well as general-purpose calculations. When designing a new GPU, NVIDIA can duplicate the SMs in parallel to accommodate the computational needs of the specific GPU-model as shown in Figure 2.1. As of January 2009, the largest number of such SMs on a single GPU is 30, and can be found on the high-end cards on the new 200-series[5].

To differentiate more, the memory sizes on the card, clock frequencies, bus width and other features can also be adjusted, but common for all NVIDIA graphics cards as of the 80-series is the presence of one or more SMs. As will be explained in the later section, there are some minor additions which has been made to the SM in the 200-series, but the main architecture is the same.

Streaming Multiprocessors

The Streaming Multiprocessor (SM) is the computational unit in the Tesla Architecture [23], and is shown in Figure 2.1. It is designed in such a manner that it can be duplicated to support the desired degree of parallelism. It consists of:

[3]http://ati.amd.com/products/
[4]http://www.nvidia.com
[5]http://www.nvidia.com/object/geforce_family.html

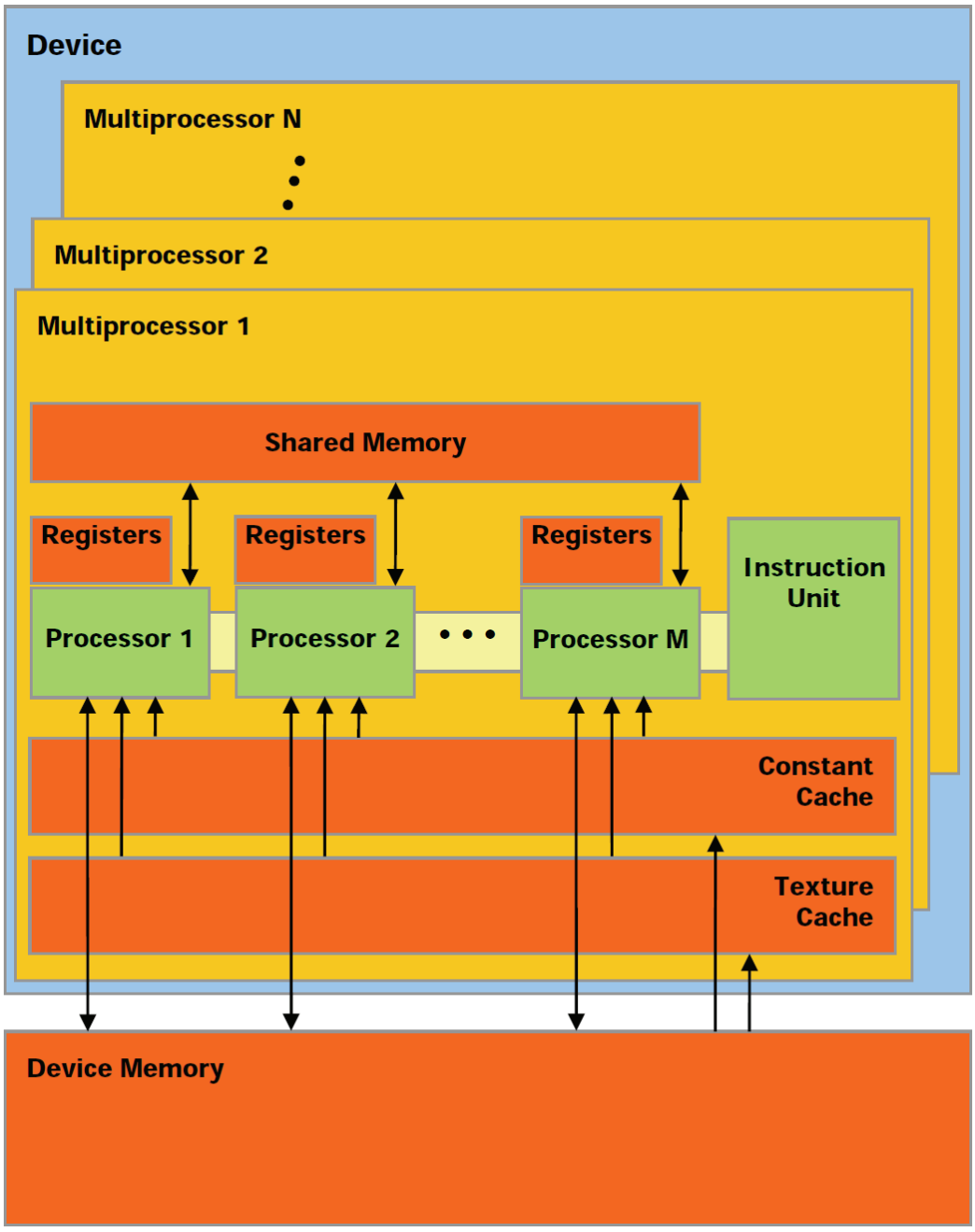


Figure 2.1: The Tesla Architecture (courtesy of NVIDIA).

Streaming Processors (SP) The eight Streaming Processors are the units executing most of the instructions. They all execute the same in-

struction synchronously in parallel, and can execute both integer and single-precision floating point operations.

Special-Function Units (SFU) The two Special-Function Units are designed to perform the more specialized functions which are often used and too demanding to perform using the simple instruction set available in the SPs.

Shared memory The shared memory is used by the SM to locally store the data required for the calculations. The Tesla Architecture does not implement a data-cache hierarchy, leaving the developer with the task of handling locality of data.

Multithreaded Instruction Fetch and Issue Unit (MT Issue) The instruction handling unit is responsible for fetching, decoding and issuing the instructions to be run on the group of SPs. Each SM runs 32 threads in parallel in what is called a warp. These 32 threads are run in groups of eight, and all perform the same instruction synchronously in parallel.

Double-Precision Floating Point Unit In the newest series of GPUs from NVIDIA, a single double-precision floating point unit to support higher precision for scientific applications has also been included.

Memory Hierachy

The memory hierarchy on the GPUs consists of two levels [24], which allows the developer to utilize data-locality to increase performance. Within each Streaming Multiprocessor, there is a shared memory area, which is shared between all the Streaming Processors. This memory offers a low latency data access, well suited for frequently accessed data. To accommodate parallel access by the SPs the shared memory is divided into memory banks [9, 15, 24] which can be accessed in parallel, but access to the same bank is sequential. Thus, if all threads executing in the SM access different banks, it will run optimally, but once two or more threads try to access the same bank, there will be a performance penalty.

The second memory level is the global, local and texture memory provided by external memory [23, 24]. Even though it may seem as three memory locales, it is the same physical memory, but when allocating it, one can choose which type to use it as. This choice determines the behavior of the allocated memory. For instance, the local memory is only accessible from the allocating thread, while the global memory is globally accessible.

Dedicated and Shared Global Memory

The Tesla Architecture defines the layout of the processing elements and how they interact. It also defines the size of shared memory and registers. The global memory, however, is not defined in the architecture [24]. The only thing specified is that the GPU can access the global memory through a set of memory controllers. This allows the global memory to be a dedicated memory on the graphics card, or a part of the local memory on the host-machine. The difference is the price and performance, as dedicated graphics memory is more expensive but also faster. On current graphics cards from NVIDIA, the shared memory approach is only used on low-end graphics cards typically found in media centers and laptops.

Program Execution

When executing a program on the GPU, the different threads the program is composed of are grouped together in thread blocks of up to 512 threads [24]. These thread blocks are distributed across the available streaming multiprocessors for execution. The SM then splits the thread block into warps of 32 threads, which are executed in four groups. The reason for this execution mode is to allow the instruction unit to fetch, decode and issue its instruction, a task which takes the equivalent of four instruction executions. All the 32 threads still have to perform the same instruction, making the SM a Single-Instruction-Multiple-Data processor (SIMD) [15, 24]. An effect of this requirement is that if two threads within one warp have divergent paths, the two paths must be executed sequential, giving a performance reduction [17]. The threads not following the executing path are disabled during the execution, leaving the integrity of the thread intact.

If a thread within a warp performs an IO-operation or for some other reason is suspended, the SM can change to another warp, choosing among all ready warps assigned to the SM.

2.1.5 AMD Stream Architecture

The AMD Stream Architecture is in many ways very similar to the Tesla Architecture, but with some small differences [1]. The main component is the SIMD Engine which can be compared to the NVIDIA Streaming Multiprocessor. The SIMD Engine contains a number of Thread Processors which all execute separate threads, and like the Tesla equivalent, all execute the same instruction. The number of Thread Processors in the SIMD Engine varies among the different ATI GPU models. Each Thread Processor has a

separate local set of registers, in addition to the global memory shared among the SIMD Engines. One key difference compared to the Tesla architecture is that all the Thread processors have double-precision support and a unit for handling special functions.

2.2 General-Purpose Computing on GPUs

Using the GPU for general-purpose computations can give the developers access to great computational power. Since the CPU currently struggles to improve the performance of sequential programs, parallelism is utilized to speed up programs. The GPU with its 300 parallel processors is among the largest parallel commodity processors. Utilizing this parallelism can give great speedups and help overcome the *brick wall*.

2.2.1 The Brick Wall

As pointed out by Patterson et. al. in *The Landscape of Parallel Computing Research: A view From Berkeley* [3], the sequential processor has reached a brick wall. Today's production techniques are incapable of maintaining the prediction of Moore's law and because of that CPU vendors have shifted their focus away from increasing the clock frequencies. To overcome the challenges, there are three problems which must be addressed, and together these form the Brick Wall.

Power Wall As the processors computational power increases, so does its power consumption. The CPU operates by switching electronic signals on and off over a set of transistors. To increase its computational power one can increase the number of transistors or increase the frequency of switching the signal[6].

$$Power_{dynamic} = \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched \quad (2.1)$$

$$Power_{static} = Current_{static} \times Voltage \quad (2.2)$$

As can be seen in Equation 2.1 and 2.2 [15], both increase in the number of transistors and clock frequency affects the power consumption of the CPU. This increased power consumption also increases the heat emission, thereby

[6]More commonly called the clock frequency

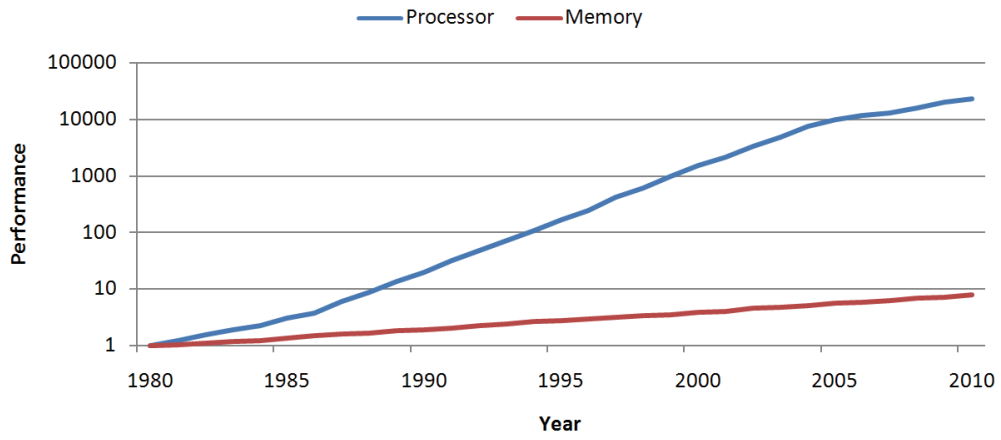


Figure 2.2: The development in CPU and Memory performance. [15].

increasing the need for cooling, which also is a power demanding operation. Today's processors have reached the point where the cost of increasing the power consumption outweighs the gains.

Memory Wall While the CPU until now have complied to Moore's Law and doubled its computational power every other year, the increase in memory access time has not kept up the pace, as can be seen in Figure 2.2.

To overcome this performance gap, memory hierarchies have been developed, utilizing several levels of cache to hide the high access times to larger storages. However, these memory hierarchies add more complexity to the CPU, and bring forth several new challenges, such as memory integrity.

ILP Wall Another approach heavily used to increase computational power of a CPU is Instruction Level Parallelism (ILP) [15]. By analyzing the instructions, the CPU can find and execute instructions which can be performed in parallel. The CPU can also speculatively execute branches and loops. All these techniques require that the instructions to be performed in parallel do not have data dependencies between them. Studies show that the number of instructions which can be parallelized is limited [15, 3], thus making ILP only a short-time solution to the need for more computational power.

2.2.2 Parallelism

To overcome the brick wall, CPU manufacturers have chosen to utilize parallelism, thereby increasing the number of processor cores instead of increasing the clock frequency. To double the theoretical computational power of a CPU, one only needs to double the number of cores. Managing multiple cores in parallel in reality requires additional hardware management which reduces the performance [15]. It also requires most programs to be rewritten to support the use of parallelism [35], or else they will continue to utilize one core and thus experience no increase in performance.

The use of parallelism is not a new concept within HPC, as most supercomputers use a large amount of CPUs spread across multiple nodes. While this can be compared to running several cores within the same processor, there are key differences. The overhead of communication between the internal cores is much lower than between nodes, and allows for more synchronization between the different threads and processes.

Still with lower communication cost between the different cores, the need for specially designed parallel algorithms is present. Due to the communication patterns, the best serial algorithms may not be suited for parallelization. Within most problems there is also a sequential section which is hard or impossible to parallelize [2, 35].

Amdahl's law [2] gives the correlation between the speedup of a parallel algorithm over the serial version, based on how large a portion of the algorithm which can be parallelized. In Equation 2.3 the speedup is given by the ratio of sequential code r_s in the algorithm, and the number of processors n. Also given is that $r_s + r_p = 1$ where r_p is the ratio of parallelizable code.

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}} \tag{2.3}$$

Amdahl's law states that the performance gain from adding more processors in parallel will eventually reach zero as more and more processors divide the work among themselves. Thus, at some point the workload will be so little for each processor that the serial part of the execution dominates the running time. While this has been used by some as an argument against parallel computing, there are many problems which benefit greatly from parallel processing even though they do not scale to several thousand processors.

2.2.3 Using GPUs for General-Purpose Computation

The idea to use the GPU for general purpose computations is not new. Since the first programmable shaders were made available, the GPU has in some

sense been able to perform such computations [7]. However, utilizing the computational power has not been easy since the GPU was not designed with such use in mind. A computation which was to be run on the GPU had to be transformed into a computation that would fit the fixed pipeline of the GPU [28]. For example, this could be done by storing all the data as textures which were applied to a single polygon covering the screen. The pixel-shader could then be programmed to alter the values of the pixels based on the neighboring pixels and by this for instance simulate the *Game Of Life* [12, 13]. Another problem with this approach was that these calculations had to be expressed using either OpenGL or Direct X and their respective shader languages. This mixture of two different fields of expertise would also prove to be a challenge, since the common HPC-programmer did not have the skills required to develop such a program, and the graphics developers may not have the understanding of HPC-problems and algorithms.

One approach was to remove the graphics aspect from the development process by developing languages which hides the graphical aspect of the computation [28]. By this, the language allows the developer to create normal programs which are then compiled into a graphical program. BrookGPU[7] and Sh[8] are examples of such languages.

The main problem with the approach of hiding the graphical computation was the fact that the computations were still converted to graphical computations, and thus required a well written compiler to successfully convert all programs while maintaining correctness and performance.

Another approach, which has become more popular, is to give direct access to the hardware through vendor specific languages [7, 28]. AMD Stream SDK[9] (successor of ATI Close To Metal) and NVIDIA CUDA[10] both compile directly to instructions which is run on the shader units. This approach eliminates the conversion between general program and graphical program, and thus removes some of the complexity and pitfalls. Another advantage is that the language can be simpler and it avoids some of the language artifacts which the convertion to graphics code may impose on the language.

2.2.4 NVIDIA CUDA

NVIDIA has launched CUDA [24], a programming language extension for C[11] alongside with its Tesla Architecture. This extension enables the developer to

[7] http://graphics.stanford.edu/projects/brookgpu/
[8] http://libsh.org/
[9] http://developer.amd.com/gpu/ATIStreamSDK/
[10] http://www.nvidia.com/cuda
[11] A general-purpose programming language.

access and use the shaders for general-purpose computation. The distinction between the architecture and CUDA is not clear, as NVIDIA often refers to possibilities and limitations as if they were properties of CUDA rather than the Tesla Architecture.

Kernel

The CUDA programming model builds around what NVIDIA calls kernels [24], which are methods executed on the GPU. These kernels are written under the Single-Instruction-Multiple-Threads paradigm, which allows the programmer to develop code which closer resembles sequential code which CUDA then runs in parallel. The level of parallelism is given as parameters to the kernel using a special syntax.

To declare a kernel one indicates this by using a new function type qualifier provided by CUDA. There are three such qualifiers.

`__global__` indicates that a method is to be compiled into a kernel and executed on the GPU. It also states that the kernel is callable from the host.

`__device__` also gives that the method is to be executed as a kernel on the GPU. The main difference from the `__global__` qualifier is that this kernel should only be callable from within other kernels.

`__host__` is a qualifier stating that the method should be executed on the host, and has thus no effect when used alone. However, if the `__host__` qualifier is combined with the `__device__` qualifier, the method is compiled into both a method on the host and a kernel on the GPU which in many situations may be beneficial.

A kernel may be called in the same manner as a method, except that there are some additional parameters that must be given. These parameters specify the level of parallelism and are supplied through a special syntax added to the method-call. As can be seen in Listing 2.1, the syntax uses `<<< Dg, Db, Ns, S >>>` to specify the parameters. The `Dg` and `Db` parameters respectively specify the number of blocks and number of the number of threads within each block. Both these are `x,y,z`-vectors allowing a 3-dimensional grid to be constructed for over the threads in the block, while the blocks may be laid out in a 2-dimensional grid. `Ns` and `S` are optional parameters, giving the size of the dynamically allocated shared memory for each block, and the stream which the execution should be assigned to.

```
// Kernel
__global__ void vecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main()
{
  // Kernel invocation
  vecAdd<<<1,N,0,0>>>(A,B,C);
}
```

Listing 2.1: CUDA kernel example. [24]

Memory

The memory on the Tesla Architecture [24] is, as explained in Section 2.1.4, a two layered hierarchy with a large global memory and a smaller shared memory on each Streaming Multiprocessor. Any data which is to be processed on the GPU must first be copied from host-memory onto the GPUs global memory before it can be further distributed to where it may be needed. The results must also reside in global memory to be accessible for retrieval from the host.

To allocate memory on the GPU, special functions provided by CUDA must be used. One of these is `cudaMalloc` which allocates a continuous part of memory for the application. There are other methods as well which allocate memory optimized for various access patterns such as 2-dimensional grid. Once the memory is allocated, data can be copied to and from the GPU using `cudaMemcpy`.

On the GPU the global memory can be accessed using normal C-syntax, but due to the high access cost for global memory, one wishes to use the shared memory as a sort of cache. To allocate shared memory one can initialize a variable with the `__shared__` qualifier. By doing so the variable is stored in the shared memory and is only accessible from within the thread-block.

Execution

The CUDA kernel [24] is executed in parallel on one or more Streaming Multiprocessors depending on the launch configuration. Once a kernel is started,

there may be need to synchronize the execution, and this can be done using `__syncthreads()` within a thread-block. It is not possible to synchronize various blocks, but this effect can be achieved by splitting the kernel in two and waiting for all blocks to finish the first kernel before initializing the second.

During execution there are also a number of variables accessible to the program which contain information about the execution. The variables `gridDim` and `blockDim` contain the size of the grid of blocks and threads within each block. To identify a thread, `blockIdx` and `threadIdx` contain the id of the block and the thread with respect to the block. By using these variables one is encouraged to create code which scales well over a various number of blocks and threads.

2.3 NVIDIA Tesla Architecture Performance Characteristics

While the GPU allows general-purpose calculations to be performed, it is not a fully general-purpose processor [7, 28], and thus has a bias towards graphics processing. This bias has made the architectural designers make certain tradeoffs with regard to performance to create the optimal GPU for what NVIDIA considers to be its main markets. This section will shed light on some of these performance characteristics.

2.3.1 Host to Device Transfers

When using a GPU for computations it usually requires data as input and in most cases produce output data. These data must be copied to and from the GPUs memory, since the GPU is unable to access the host memory while performing the calculations. This copy operation can be costly in many applications, especially for data intensive calculations.

Direct Memory Access

In modern operating systems there are great memory requirements, but the physical memory may not be sufficient. To allow programs to use more memory than physically available, paging has been introduced as a feature [15]. Paging allows memory to be swapped out to a hard-drive when it is not needed, and thereby freeing physical memory for other uses while still maintaining the integrity of the virtual memory. There is a major drawback with this technique, and that is that only the operating system knows the exact location of a memory segment since it may be moved around due to

memory swapping. In situations where an exact memory location is needed, page-locking can be used. This is mostly used when using Direct Memory Access (DMA) [33], since DMA allows memory copies to be handled by a DMA handler instead of the CPU.

Transfer to GPU

All copy operations between host memory and device memory requires the use of DMA [27]. Since most memory locations are not DMA accessible, there are two techniques which can be used. The first solution is to store all data which will be used on the GPU in page-locked memory locations. This is not always feasible since page-locked memory locations are a scarce resource. The other option is to copy the data to a page-locked memory location before copying it to the GPU. This approach requires an extra memory copy which can be costly. When to use the which of the two techniques depends on the usage of the data. For data which will be copied often to the GPU it may be best to use dedicated page-locked memory, while for a one-time copy one may just as well use the second technique [16].

In CUDA both techniques is supported automatically based on which type of memory location that is given to the copy-instruction [24]. By default, a pageable memory location is given when using the standard C/C++ command `malloc`, and if such a memory location is given to the copy instruction in CUDA, it will copy the data to a page-locked memory location before copying it to the GPU [27]. To allocate a page-locked memory location CUDA provides a method `cudaMallocHost`.

Which situations to use the two techniques may wary from application to application, but the measured bandwith when using the techniques in a simple test-case can be seen in Figure 2.3

Architectural Differences

The Tesla Architecture [24] allows the GPU to have its own dedicated memory or using a section of the host-memory as device memory. While these two approaches yield no difference for the developer, it may have an impact on performance, as the dedicated memory usually has higher performance. This can be seen in Figure 2.3 where there is a clear advantage to the NVIDIA GTX 280 card with dedicated memory over the NVIDIA 9300m and ION which uses local memory.

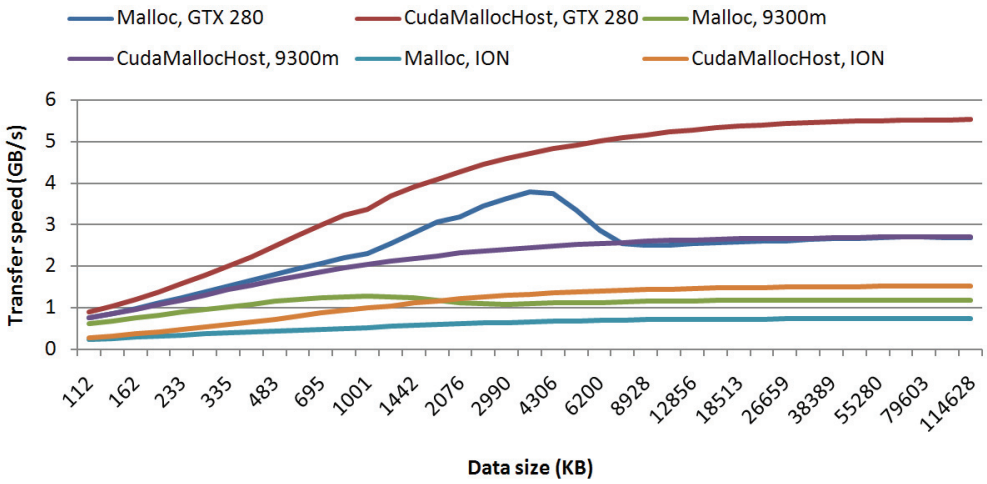


Figure 2.3: Bandwidth for transfers from host to device memory.

2.3.2 Data Access

In large data volume applications, the data may be to large to fit in memory and hard-drives must be used. This means that any data which should be used must be read into main memory before it can be used. If this should be done on the GPU, it is even more cumbersome, since it must first be read into memory, before being copied onto the memory of the GPU [33, 24]. The results calculated at the GPU must also be copied back to the host-memory if it is to be used further by the CPU.

Modern hard-drives are considered to be the bottleneck of any application operating over large datasets. The highest transfer rate[12] found for sustained-read for a hard-drive was 171MB/s[13] which is fairly low considering the memory bandwidth of the new Intel Core i7 processor[14] which is 25.6GB/s. To give a illustration of the speed difference, the transfer speeds between the various components in the computer is shown in Figure 2.4 where the bandwidth is given as the width of the connector. Also displayed in the figure is the bandwidth of data transfers to and from the GPU which was found to be close to 6GB/s [16].

[12]Solid-State Drives may achieve higher rates
[13]http://www.seagate.com/docs/pdf/datasheet/disc/dsi_cheetah_15k_6.pdf
[14]http://ark.intel.com/cpu.aspx?groupId=37147

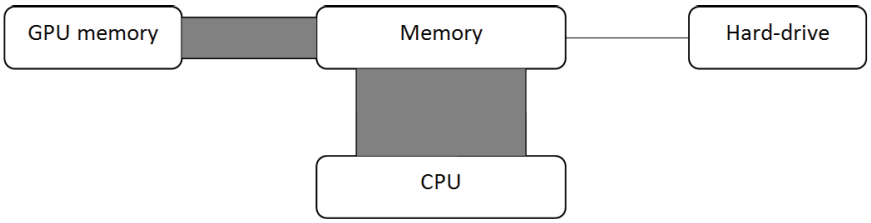


Figure 2.4: Bandwidth between computer components given by the size of the connection.

Determining the Transfer Time

This thesis focuses on possible approaches which can improve the performance of high data volume applications through accelerating the data access made to the data structures. We introduce T which is the time required to retrieve an entry in the data structure.

$$T = L_{HD} + \frac{S}{B_{HD}} \tag{2.4}$$

The time required can be expressed as a function of the hard-drive latency L_{HD}, size S, and hard-drive bandwidth B_{HD} of the data transfer as given in Equation 2.4. This equation describes the simplest form of data access where the data is stored uncompressed on a single disk. Once read, it is stored directly in the location where it will be used.

Introducing Compression

To reduce the impact of the hard-drive transfer rate, one can use compression thereby reducing the size of transferred data. Compression will be discussed in detail in Section 4.2. This compression reduces the time required to transfer the data, but introduces a computational step which decodes the data and copies it over to the final memory location. The new equation for the time required to access the entry which takes into account the time C_{comp} required by the added computational step and the compression ratio R_{comp} is given in Equation 2.5.

$$T = L_{HD} + S \cdot (\frac{R_{comp}}{B_{HD}} + C_{comp}) \tag{2.5}$$

The second case can be used to describe the first case by setting $R_{comp} = 1$ and $C_{comp} = 0$ which gives Equation 2.4. Because of allocation of an extra

memory location and initialization of the computation, there is an added component to the latency but this can be neglected.

Offloading to the GPU

The third step is to offload the decompression of the data to the GPU. By doing so, the CPU will be free to do other computations, and the GPU can utilize its many processors to decode the data. By introducing the GPU as an accelerator, there are a number of added complexities. First of all, the data has to be copied from memory and over to the GPUs memory. Second is the decompression stage in the same manner as with the compression on the CPU. After the decompression is completed, the uncompressed data must be copied back to the host-memory. The initialization of the GPU-kernel and allocation of memory on the GPU can be summarized into an GPU latency L_{GPU}, and the bandwidth to and from the GPU is given as B_{GPU}. The total amount of data copied to and from the GPU is $S * (1 + R_{comp})$. The equation for the time required to access the data when using the GPU for decompression is given in Equation 2.6.

$$T = L_{HD} + L_{GPU} + S \cdot (\frac{R_{comp}}{B_{HD}} + \frac{1 + R_{comp}}{B_{GPU}} + C_{comp}) \tag{2.6}$$

Focusing on Throughput

A search engine [5] can be considered to focus on High-Throughput Computation (HTC) rather than a High-Performance Computation, as long as the latency of a single query is below a certain threshold. This query-latency is the time from the query is given until it is answered. There are many components of this latency, and the retrieval of the index-entry is one of them. By offloading the decompression to the GPU, the CPU is free to perform other tasks, and the overall throughput of the system may increase even if the time required to fetch an index-entry may increase.

Optimizations

When using the GPU to accelerate the decompression of the data, the data-copy between memory-locations is a bottleneck which may be a problem when trying to achieve good performance. The first and most obvious approach to alter the process is to remove the need for a temporary place to store the data between the hard-drive access and the copying to the GPU. This is not possible yet, as the GPU is incapable of reading directly from the hard-drive, and the hard-drive is unable to write to the GPUs memory.

When memory is copied to the memory of modern GPUs, the CPU informs the GPU of where to find the data in the host-memory, and the GPU then uses DMA to access the memory location [27]. This is the same approach as the hard-drive uses when a file on the drive is accessed. In this case the hard-drive is informed that the CPU wants a certain file, and the location in which the content of that file is to be placed. The hard-drive reads the file and stores it to the given memory location using DMA. If the GPU were able to read directly from the disk, or the disk was able to write to the GPUs memory, then the time required to fetch the data would be as given in Equation 2.7. Due to the much higher bandwidth to the GPU than the bandwidth to the hard-drive, it can be reduced to Equation 2.8.

$$T = L_{HD} + L_{GPU} + S \cdot (\frac{R_{comp}}{min(B_{HD}, B_{GPU})} + \frac{1}{B_{GPU}} + C_{comp}) \tag{2.7}$$

$$T = L_{HD} + L_{GPU} + S \cdot (\frac{R_{comp}}{B_{HD}} + \frac{1}{B_{GPU}} + C_{comp}) \tag{2.8}$$

To mimic this effect, it is possible to perform most of the transfer to the GPU parallel with the transfer from the hard-drive. This is done by dividing the transfer into n parts, and start to copy a part to the GPU asynchronously as soon as it is read from the hard-drive. By choosing the right size to partition the transfer into, the extra time needed to copy data to the GPU would only be equal to the time needed to copy the final part, giving Equation 2.9. This approach would give the effect seen in Figure 2.5.

$$T = L_{HD} + L_{GPU} + S \cdot (\frac{R_{comp}}{B_{HD}} + \frac{1 + \frac{R_{comp}}{n}}{B_{GPU}} + C_{comp}) \tag{2.9}$$

The operation of copying of the result to the host-memory is more cumbersome to remove, since it will be used by other parts of the system, and its lifespan may not be known. It is therefore beneficial to free its memory location on the GPU and instead maintain it in host-memory which is more cost-efficient.

Memory Mapped Files

The last approach which can be considered is to use Memory Mapped Files which is specified in POSIX [20] through `mmap()`. This is a technique which creates a memory pointer to a Virtual Memory location in which the file is mapped. In this way, the file can be accessed as if it were residing in the memory, and the task of actually supplying the data is left to the operating system. By choosing this approach, the programmer can utilize efficient

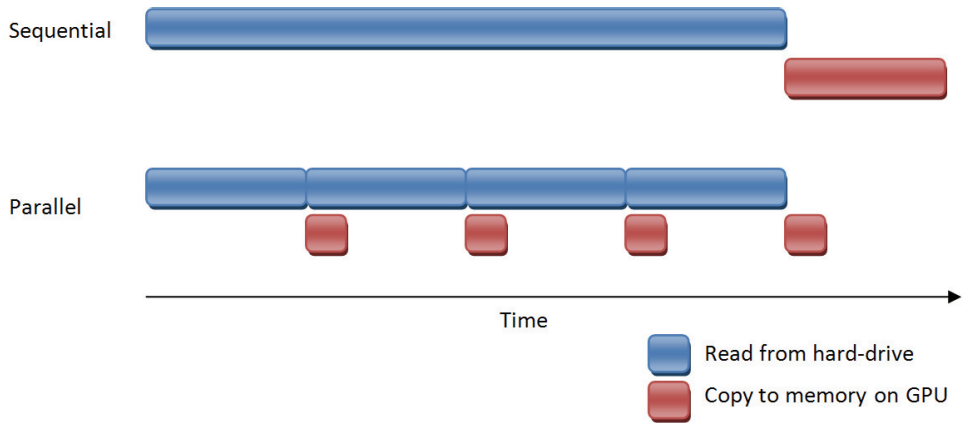


Figure 2.5: Mimicking GPU read from hard-drive by hiding data copy between host- and GPU-memory.

prefetching and caching on an operating system kernel level rather than managing the task itself.

Preliminary Performance Analysis

To determine if either of the two proposed improvements (data copy hiding and memory mapped files) would give improved performance over the basic approach, a test case was developed. By measuring the time each of the three approaches use to read data of various sizes from file and make it accessible on the GPU, an indication on the performance gain by the improved approaches could then be found. In Figure 2.6, the speedup of the two improvements with regard to the simple approach is given. As one can see, the memory mapped files obtains much better performance than the other two approaches. The hiding of data copy approach does not achieve noticeable speedups. This might be due to an added cost of initializing multiple file reads and data copies.

2.3.3 GPU Memory Access

Data stored on the GPU is kept in memory which must be accessed by the processors when the data is needed. The Tesla architecture [24] provides several alternative approaches to both storing and accessing the data in memory. These different approaches are tightly bound to the memory structure on the GPU, and also customized to benefit certain access patterns and uses. Some of these access patterns may be intended for graphical purposes only, but can

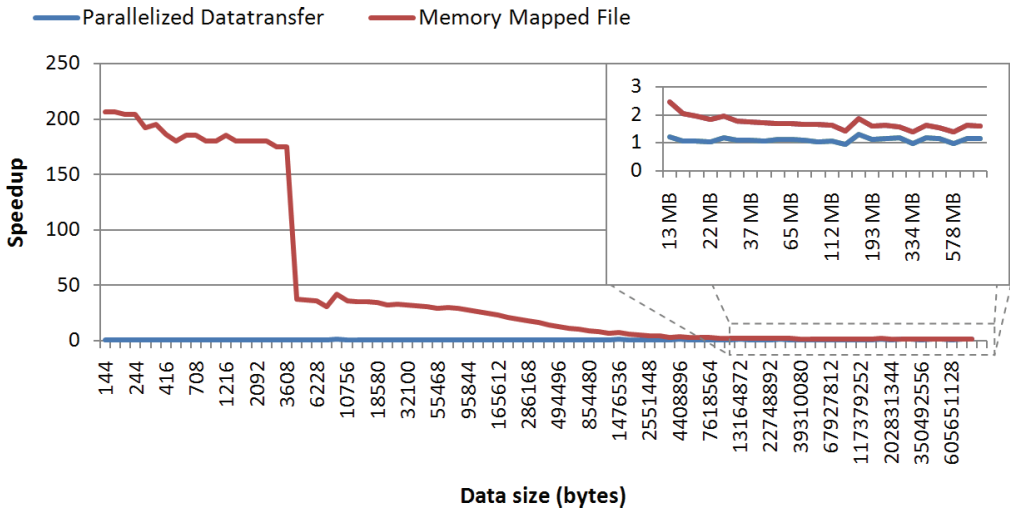


Figure 2.6: Speedup of data transfers to the GPU using Memory Mapped Files and parallelizing transfers to mimick GPU read from hard-drive.

be utilized to speed up memory access in general-purpose applications [24].

Memory Hierarchy

The memory on the GPU is, in the same manner as on the host [33, 24, 15], divided into a hierarchy of memory locations with different properties. This division is to better balance price versus performance of the GPU, since high speed memory is expensive and large quantities of such memory would make the GPUs too expensive for common use. To accommodate both high bandwidth and large memory storage capacity, several levels of memory are used. The lower levels provide the large storage space, while the higher levels provide high bandwidth. By carefully choosing which data is stored in the higher levels, most data access only needs to access the higher levels and thus achieves high performance. This technique is called caching. If data is not found in the higher levels, the lower levels must be accessed and what is called a cache miss occurs [15]. A high number of cache misses reduces the overall performance due to the longer access time.

Global Memory

On the GPU there are two levels in the memory hierarchy [24]. The large storage capacity is provided by the global memory which on the high-end GPUs reaches four gigabytes in size. When data is copied from the host

memory onto the GPU, it is copied into this memory. An access to this memory is slowed down by a latency between 400 and 600 clock cycles [24], and is thus not able to fulfill the role as high bandwidth memory. To improve performance, global memory allows memory access to adjacent addresses to be grouped together into one read or write operation. This approach is called coalesced read and write operations.

Coalescing A coalesced memory access [23, 24] is multiple memory accesses grouped into one access. This can be performed by the 16 threads in a half-warp if they at the same time wishes to access 16 32-bit memory locations residing adjacent in the global memory. In the first version the requirement was that thread n accessed the n'th memory address within in the segment. This initial requirement has been loosened and now the requirement is that all accesses are within the memory segment. It can also be coalesced access to 16 64-bit memory locations, but this is then divided into two coalesced memory accesses.

One of the first things to optimize in a CUDA program is the access to the global memory, since its large latency can drastically reduce the performance of an application [24, 23].

Texture Memory

When using the GPU for graphical applications it is common to do many computations on textures, and the GPU therefore has special functionality [24, 7] to improve the performance of such calculations. The textures are, as other data, stored in the global memory, and no distinction between the two types is made. However, defining data has one major benefit, and that is the added functionality the GPU provided for access to texture data.

Caching On a Streaming Multiprocessor there is also a memory location named Texture Cache [24]. This is used by the GPU to automatically cache texture data. One can leave the job of caching to the GPU by by defining the data as texture. The texture cache is eight kilobytes large, and by utilizing this cache as well as the shared memory and registers, there is more data which can be stored higher in the memory hierarchy. One major disadvantage with the texture memory is that it has no support for write-back [24, 15]. If a texture it altered during run-time, it is undefined if that change would be visible to all blocks. Data is already cached it must be thrown out and fetched again for the change to appear, which is not possible to force. The only way to be certain that a write to the texture memory is propagated through the system is by restarting the kernel.

Constant Memory

The constant memory [24] is cached in the same manner as the texture memory, and thus it is able to supply data fast to the kernels. The constant memory is, as the name implies, meant for constants needed during execution. It is therefore read only, and often too small for other uses with its total size of 64 kilobytes.

Registers and Shared Memory

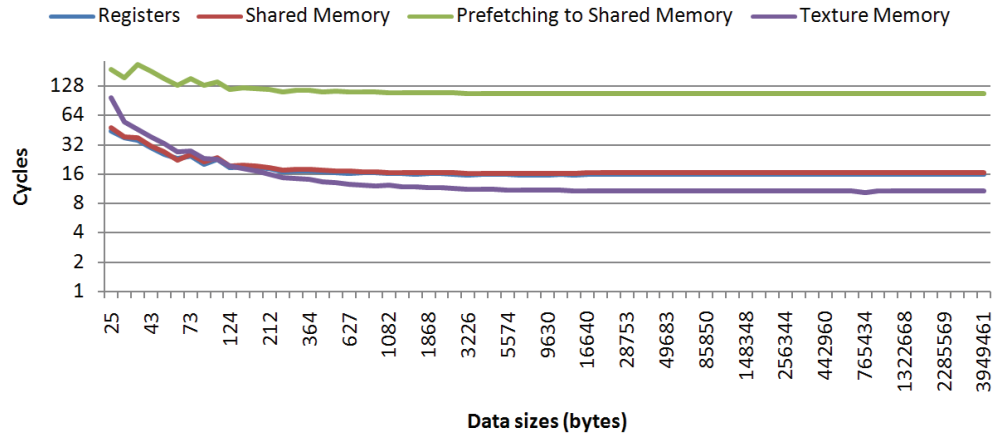
To accommodate the need for high performance memory, the registers and shared memory are included on the GPU [24]. These located on the Streaming Multiprocessor, and are only accessible from within the thread-block. The registers are intended for storing currently used variables, and there are either 8192 or 16384 32-bit registers available depending on the GPU model. The shared memory is, as the name implies, shared between all the threads in a block, making it ideal for storing data needed by multiple threads, or data exchange. The size of the shared memory is only 16 kilobytes in current versions of the Tesla architecture.

Banks Both the shared memory and the registers are divided into 16 segments called banks [24, 15]. This division allows parallel access to the individual banks. The number of banks matches the number of threads in a half-warp, allowing concurrent access by these threads if the access maps one-to-one between the threads and the banks. If two or more threads access the same bank, each access will be processed in sequence, causing a performance hit [9].

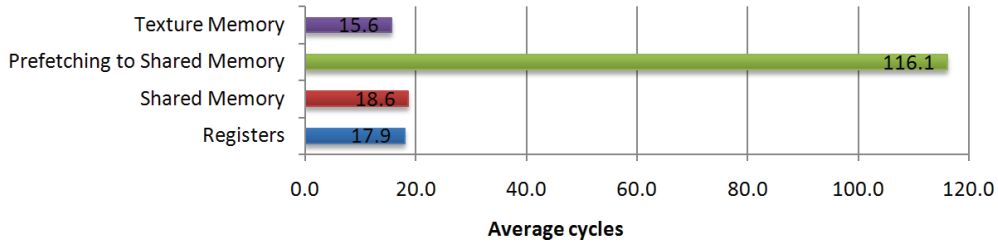
Correct use of shared memory and registers are essential for obtaining good performance, as these memory locations are intended for high speed access, and can be used by the developer to manually implement caching.

Sequential Read Performance

To show the most beneficial location to store data used by GPU application, a simple test has been created. It reads a data set sequentially and measures the average time needed to access each memory location. The test has some overhead due to instructions using the data and looping, but this overhead is equal to all tests, and does not affect the ranking of the different memory locations. Four different approaches are tested. The first is reading directly from global memory with coalesced reads into a register. The second and third use shared memory to read the data into before using it. The difference



(a) Various datasizes



(b) Average

Figure 2.7: Cycles needed for data access using different access approaches.

is that the third prefetches multiple data before using these data. The fourth approach is to use texture memory.

As can be seen in Figure 2.7, the use of texture memory is most suited for sequential reads. This is most likely due to the fact that the GPU handles the caching which reduces the number of explicit instructions. The reason why the prefetching into shared memory is slower is the large overhead of control instructions to handle situations when the buffer does not divide the data size.

Repeated Read Write Performance

While there is a large penalty for accessing memory in a lower level of the hierarchy [24], these accesses are more rare than accesses to the higher levels [15]. The GPU provides two high-level memory alternatives which can be read and written, the shared memory and the registers. These memory locations will be used for temporary variables and other often used data, and may be read and written often. Therefore, a test to determine the average cycles

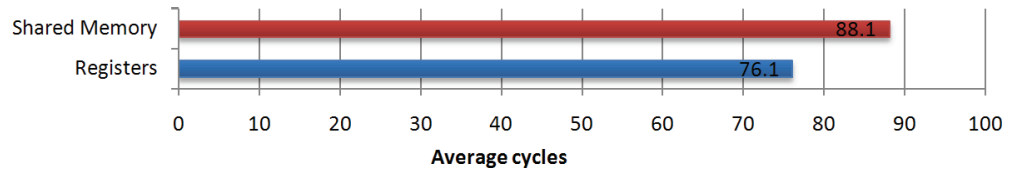


Figure 2.8: Average cycles needed for read and write data access to registers and shared memory.

needed to read and write to these areas is created. It performs some read and write instructions multiple times against the same location and measures the average cycle count. There is an overhead from looping included in this measurement, but it stills clearly shows which memory location give the best performance.

In Figure 2.8 one can see that the registers have lower access time than the shared memory. Since the shared memory is accessible by all the threads in the thread-block, the GPU must handle the cases when multiple threads access the same address, while the registers are private for each thread. This added complexity may be the reason for the higher access time.

2.3.4 Kernel Initialization

All computation on the GPU is performed using kernels. These are essentially methods which is compiled to device specific instructions [24]. Many problems can be solved using a single kernel which is initialized a single time, but some problems may require more complex calculations which do not fit in a single kernel or requires multiple kernels to complete [9].

Kernel Size Limitations

The GPU performs a number of instructions in the same manner as the CPU. The main difference is that the GPU does not have access to the file in which the program is stored. Therefore any program which runs a kernel on the GPU must first upload the instructions in the kernel to the GPU, then instruct the GPU to execute the kernel. This kernel must be stored on the GPU for the duration of the execution, and this storage space is of a finite size. Due to this limitation, each kernel may only be up to a certain number of instructions and large calculations may therefore need to be split into multiple kernels. When switching between two kernels too large to fit in instruction memory simultaneously, the old kernel must be removed before uploading the new kernel. If each kernel is run for a long time, this kernel

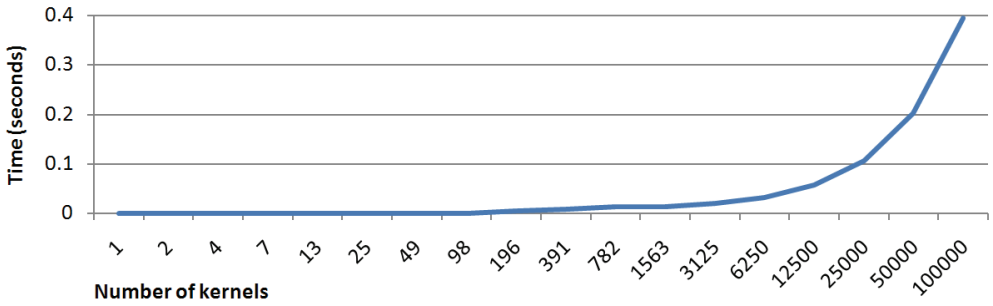


Figure 2.9: Cost of dividing a calculation into multiple kernels.

switching may not have a noticeable impact on the performance, but the case where each instruction in the kernel is executed only a few times, this may prove to be a noticeable factor in the performance.

Kernels and Streaming

For applications operating on large data volumes, it may not be possible to fit all the data on the GPU at one time. In this case, the data must be processed in batches by a kernel running once for each batch. This approach requires multiple initializations of the kernel which may contribute to a lower performance. The much impact this has on the performance can be seen in Figure 2.9, where a simple problem has been divided into different number of kernel calls.

Applications which handle streaming data such as audio stream filters may receive data on a regular basis and call the same kernel repeatedly with different data. For such applications, kernel initialization cost can influence performance.

2.3.5 Computational Characteristics

The streaming multiprocessor is capable of performing a lot of different instructions, but there are pitfalls which can reduce the performance [24, 9]. Most of these are related to the need for precision, as CUDA provides faster instruction with lower precision. Also, costly operations as modulo and division should be avoided, and replaced with cheaper instructions when possible [24].

Another important consideration to make is that the Tesla Architecture is a hybrid of Multiple-Instruction-Multiple-Data and Single-Instruction-Multiple-Data architecture. This affects the performance of the application when

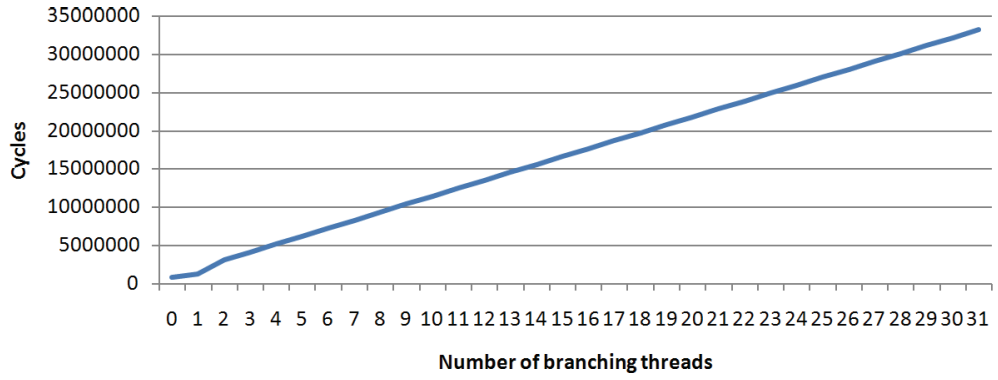


Figure 2.10: Average cycles needed for running a single instruction branching 10.000 times.

branching occurs within a warp. Since the Streaming Processors within a Streaming Multiprocessor must execute the same instruction, any branching must be handled by serializing the branches [17]. The effect of branches can be seen in Figure 2.10, and this effect is discussed thoroughly in the article included in Appendix B.

Chapter 3

Modelling Next Generation GPUs

When suggesting an improvement to an existing architecture, it is vital to point out the benefits of making the improvements. The approach chosen in this thesis is to create a theoretical model for both architectures and show mathematically how the performance would change with the suggested improvements

While it is easy to measure the performance on an existing architecture, measuring the performance on the suggested architecture would require creating a simulator for the architecture or actually manufacture the architecture.

In the first section, the main improvement suggested in this thesis is presented. It describes how the memory hierarchy of the Tesla architecture should be expanded. A theoretical model for the current and improved Tesla architecture is presented in the following sections, before the performance gain for the improvement is found in Section 3.3. Finally, a number of minor changes to the Tesla Architecture and CUDA which will improve the usability for developers of large data volume applications are given.

3.1 Expanding the Memory Hierarchy

The current Tesla Architecture allows the GPU to have its own dedicated memory or using a part of the host memory as device memory. Any data which is to be used by the GPU must be copied into device memory before a kernel is initialized. Any results from the kernel must be copied back to host memory after execution. These requirements forces any CUDA application to contain three steps; data-copy to device, execution and data-copy from device. While this approach may not seem like a problem in most cases, there

are cases in which this may create extra complexity for the developers. When transferring large amounts of data between the host and device memory, it may be beneficial to start computations before all the data is located on the GPU. The current Tesla Architecture allows this to be done by using streams, where both the data transfers and computations are divided into batches which are performed in an overlapping manner, thus hiding some of the data transfer cost. However, as seen in Section 2.3.4, initializing multiple kernels is a costly operation.

3.1.1 Accessing Host-memory

Allowing the kernel to directly access the host-memory removes the need for host to device transfers in most cases. Syntactically this would be similar to accessing any other memory location on the GPU, but it would have a higher cost in terms of lower bandwidth and higher latencies. It would therefore be up to the developer to reduce the number of accesses to host-memory to a minimum by pre-fetching data into device memory such as global or shared memory.

By allowing direct access to host-memory, the global memory would still maintain its role as a level in the memory hierarchy of the GPU, but it would be simpler to use it efficiently in a wide range of applications.

For an application with large volumes of data, this possibility to access the host-memory directly would remove the need to divide the calculation into several kernel executions when the data exceeds the size of device memory. With the direct-access approach, the kernel can simply fetch more data into the device memory, while discarding used data without the need to return control to the CPU.

3.1.2 Customizing Memory Hierarchy

When allowing direct access to the host-memory from the kernel, it practically adds another level to the GPUs memory hierarchy. This added layer will change the usage of the device-memory since it would allow for more data access patterns. Through the added layer of GPU memory, the need for large device memory locations will be more individual. One can thus envision a more customizable device memory on the GPU to tune the size to the individual needs. Since the GDDR memory used in GPUs is more expensive than DDR memory used for local memory, this customization of memory hierarchy would allow for more cost-efficient installations. It will also allow for the GPU to utilize larger memory when computing

3.1.3 Combining Dedicated and Shared Memory

If enabling customizations of the memory hierarchy is not feasible, there is another approach which seems easier to implement since most aspects needed exists in current GPUs. By allowing the GPU to use both dedicated and local memory as device memory, but dividing them into two distinct levels in the memory hierarchy, the same effect can almost be achieved. It would, however, require that the CPU also has write permissions to the host-memory used as device-memory, and that all data which the GPU will use is stored in this memory location.

3.1.4 Caching Problems

When using either of the two approaches for kernel access to host-memory, there will be issues with CPU caching that must be resolved, since the data stored in the host-memory may not be the valid version due to cache and delayed write-back. These problems are assumed solvable in this thesis.

3.1.5 Zero-Copy in CUDA 2.2

Zero-Copy is a feature which will be introduced in CUDA version 2.2 [25, 26]. Its main purpose is to allow the user to do, to some extent, what is suggested in this chapter. By allowing the user to access page-locked memory from the GPU, the need for copy prior to and after kernel execution can be eliminated. It is a major drawback that only page-locked memory can be used for the purpose, as page-locked memory is a scarce resource, and the developer may still be required to do extra copy operations if there is extensive memory usage on the host.

3.2 Theoretical Model

This section gives a theoretical model for the capabilities possible with the extended memory hierarchy suggested in the previous section. We will first present the model for the current architecture, and then introduce the extended memory hierarchy.

3.2.1 Variables and Assumptions

To be able to create a theoretical model describing the Tesla Architecture, there are several variables describing different bandwidths and latencies which must be defined. Some assumptions and simplifications must also be

made to make the task feasible, as it would be practically impossible to exactly describe the architecture mathematically, due to the large number of variables and characteristics involved in a actual architecture.

Latency

Every data transfer operation and computational task has a startup cost called latency, which is the time from the instruction to start the operation is given until it actually commences. When performing calculations on the GPU there are several such latencies which occur as data transfer is conducted, and these are given in Table 3.1. In addition there is also a latency l_{gpu} when starting a computation on the GPU.

Table 3.1: Latencies of data transfers to, from and on the GPU

Latency	From	To
$l_{m \to gm}$	Local memory	Global memory
$l_{gm \to m}$	Global memory	Local memory
$l_{gm \to r}$	Global memory	Register
$l_{r \to gm}$	Register	Global memory
$l_{r \to gpu}$	Register	GPU Streaming Processor
$l_{gpu \to r}$	GPU Streaming Processor	Register
$l_{m \to r}$	Local memory	Register
$l_{r \to m}$	Register	Local memory

To simplify the model slightly an assumption is made that the latencies of a data transfer are symmetric, which means that the same latency is assumed for data transfers in both directions. This is given in Equation 3.1.

$$l_{A \leftrightarrow B} = l_{A \to B} = l_{B \to A} \tag{3.1}$$

Bandwidth

There are several data transfers between host memory and device memory and also within the GPU. All these data transfers occur over various connections with different bandwidth. Since all of these can affect performance, each is taken into the theoretical model. These variables can be seen in Table 3.2.

In addition to the bandwidths between memory locations, there is also a speed with which the GPU can process the data it is provided. This will

Table 3.2: Bandwidths of data transfers to, from and on the GPU

Bandwidth	From	To
$b_{m \to gm}$	Local memory	Global memory
$b_{gm \to m}$	Global memory	Local memory
$b_{gm \to r}$	Global memory	Register
$b_{r \to gm}$	Register	Global memory
$b_{r \to gpu}$	Register	GPU Streaming Processor
$b_{gpu \to r}$	GPU Streaming Processor	Register
$b_{m \to r}$	Local memory	Register
$b_{r \to m}$	Register	Local memory

be considered as a bandwidth denoted b_{gpu}. It is not a direct property of the architecture, but rather a mixture of the architectural properties and the application executing on the GPU.

To simplify the model slightly, certain assumptions regarding the bandwidths are made. The main assumption seen in Equation 3.2 is that the bandwidth between two memory locations is the same in both directions.

$$b_{A \leftrightarrow B} = b_{A \to B} = b_{B \to A} \tag{3.2}$$

The other assumption is that the bandwidth decreases for each level in the memory hierarchy, as can be seen in Equation 3.3.

$$b_{m \to gm} < b_{gm \to r} < b_{r \to gpu} \tag{3.3}$$

The final assumption made to give an expected bandwidth of transfer between local memory on the host and the registers on the GPU. Since this can not be measured, it is assumed to be equal to the lowest bandwidth occurring on the normal path from local memory through global memory and over to the registers, as given in Equation 3.4

$$b_{m \to r} = min(b_{m \to gm}, b_{gm \to r}) = b_{m \to gm} \tag{3.4}$$

3.2.2 Current Tesla Architecture

The current approach to performing a computation on the GPU is to first copy the data to the global memory on the GPU, compute the result based on these data, and finally copy the results back to the local memory. These three steps must be performed sequentially, thus giving the sum of all the

operations as the total time needed for the operation. This is given in Equation 3.5.

$$t_{current} = t_{copy\ to\ device} + t_{computation} + t_{copy\ from\ device} \tag{3.5}$$

Copy to the Global Memory

The first step in the current procedure is to copy data to global memory. This requires the data to be fetched from local memory and transported over the interconnection between the local memory and GPU, which normally is a PCI Express 2.0 connection. Once it arrives on the GPU, it is stored in global memory. The time this operation would take is given by the size of the data transferred, the bandwidth achievable, and the latency of the transfer operation, as given in Equation 3.6.

$$t_{copy\ to\ device} = \frac{s_d}{b_{m \to gm}} + l_{m \to gm} \tag{3.6}$$

Using the assumptions stated in Section 3.2.1, Equation 3.6 is simplified to Equation 3.7.

$$t_{copy\ to\ device} = \frac{s_d}{b_{m \leftrightarrow gm}} + l_{m \leftrightarrow gm} \tag{3.7}$$

Performing Calculations

The second step is to perform the actual calculations. This is a complex operation requiring several data transfers. The first step is to bring the data from the global memory into the Streaming Multiprocessors registers. Second, the data must be copied from these registers over to the Streaming Processors registers to perform the actual calculations. The entire process is performed in reverse to return the result to the global memory. In addition the calculations are performed with a given bandwidth b_{gpu}.

The process of bringing data into the GPU and processing it is given in Equation 3.8 and the process of bringing the results back to global memory is given in Equation 3.9.

$$\begin{aligned} t_{data\ computation} = & \frac{s_d}{min(b_{gm \to r}, b_{r \to gpu}, b_{gpu})} \\ & + l_{gm \to r} + l_{r \to gpu} + l_{gpu} \end{aligned} \tag{3.8}$$

$$t_{result\ computation} = \frac{s_r}{min(b_{gpu}, b_{gpu\to r}, b_{r\to gm})} + l_{gpu} + l_{gpu\to r} + l_{r\to gm} \tag{3.9}$$

By simplifying these formulas in accordance with Section 3.2.1 and adding them together, the total time required to perform the calculation is given in Equation 3.10

$$t_{computation} = \frac{s_d + s_r}{min(b_{gm\leftrightarrow r}, b_{r\leftrightarrow gpu}, b_{gpu})} + 2(l_{gm\leftrightarrow r} + l_{r\leftrightarrow gpu} + l_{gpu}) \tag{3.10}$$

Copying Results back to Local Memory

The final step of the procedure is to copy the results of the data back to Local Memory. This process is the inverse of the first step given in Section 3.2.2, but with the size of the results s_r instead of s_d. The total cost is given in Equation 3.11 with the simplifications given in Equation 3.12.

$$t_{copy\ from\ device} = \frac{s_r}{b_{gm\to m}} + l_{gm\to m} \tag{3.11}$$

$$t_{copy\ from\ device} = \frac{s_r}{b_{gm\leftrightarrow m}} + l_{gm\leftrightarrow m} \tag{3.12}$$

Total Cost

The total cost of a computation on the GPU is the sum of the individual costs of the three steps given previously in Equation 3.5. By inserting the individual cost-formulas, the total cost can be seen in Formula 3.13, and simplified in Formula 3.14.

$$t_{current} = \frac{s_d}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm\leftrightarrow r}, b_{r\leftrightarrow gpu}, b_{gpu})} + \frac{s_r}{b_{m\leftrightarrow gm}} + 2(l_{m\leftrightarrow gm} + l_{gm\leftrightarrow r} + l_{r\leftrightarrow gpu} + l_{gpu}) \tag{3.13}$$

$$t_{current} = \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm\leftrightarrow r}, b_{gpu})} + l_{current} \tag{3.14}$$

3.2.3 Improved Tesla Architecture

The improved Tesla Architecture given in Section 3.1 opens for the possibility to copy data directly from the local memory to the registers on the GPU while the kernel is running. By doing, so the three steps used in the current Tesla Architecture can be interleaved. Thus, the bandwidth of the data through the entire process is determined by its weakest link. This is expressed in Equation 3.15.

$$\begin{aligned} t_{new} = & \frac{s_d + s_r}{min(b_{m\leftrightarrow r}, b_{r\leftrightarrow gpu}, b_{gpu})} \\ & + 2(l_{m\leftrightarrow r} + l_{r\leftrightarrow gpu} + l_{gpu}) \end{aligned} \tag{3.15}$$

By using the assumptions and simplifications given in Section 3.2.1, Equation 3.15 is reduced to Equation 3.16.

$$t_{new} = \frac{s_d + s_r}{min(b_{m\leftrightarrow r}, b_{gpu})} + l_{new} \tag{3.16}$$

3.3 Performance Improvement

An architectural change can only be justified if it provides some sort of benefit. In this case, it will be shown that the change will improve performance by hiding execution costs due to interleaving. To show this we start with defining a measure of saved time $t_{benefit}$, which is defined by Equation 3.17 and the more elaborate Equation 3.18.

$$t_{benefit} = t_{current} - t_{new} \tag{3.17}$$

$$t_{benefit} = \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm\leftrightarrow r}, b_{gpu})} - \frac{s_d + s_r}{min(b_{m\leftrightarrow r}, b_{gpu})} + l_{current} - l_{new} \tag{3.18}$$

By looking at the formula it is clear that there are two distinct cases which must be handled: when the bottleneck is either the memory bandwidth between local memory and the GPU, or that the GPU is unable to process the data fast enough.

3.3.1 Memory Bandwidth Bound

When the memory bandwidth is the limiting factor of the computing system, the problem becomes that the GPU would not receive data fast enough

to fully occupy the processing elements. This is typically the case in high data volume problems where each data element requires little processing. Examples of such problems include data decompression and list filtering.

In Equation 3.19, one can see the relation between GPU bandwidth and memory bandwidth when the application is bound by the memory bandwidth.

$$b_{m \leftrightarrow gm} < b_{gpu} \tag{3.19}$$

By using Equation 3.4 and 3.19 one can make the reduction given in Formula 3.20.

$$min(b_{m \leftrightarrow r}, b_{gpu}) = min(b_{m \leftrightarrow gm}, b_{gpu}) = b_{m \leftrightarrow gm} \tag{3.20}$$

By reducing Equation 3.18 with the reduction in Equation 3.20, one can get a formula for the reduction in computation time when the memory bandwidth is the bottleneck. This is given in Equation 3.21.

$$\begin{aligned} t_{benefit} &= \frac{s_d + s_r}{b_{m \leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm \leftrightarrow r}, b_{gpu})} - \frac{s_d + s_r}{b_{m \leftrightarrow gm}} + l_{current} - l_{new} \\ &= \frac{s_d + s_r}{min(b_{gm \leftrightarrow r}, b_{gpu})} + l_{current} - l_{new} \end{aligned} \tag{3.21}$$

By examining the resulting Equation 3.21, one can see that the time saved is equal to the time needed to perform the computations. This is because the computations can be performed while the data is copied as a stream to the GPU. Since the GPU processes the data at a faster rate than it is provided, the computation would be performed as a pipeline where the data copy of each element is the most time-consuming step, and thus hides the other costs.

3.3.2 Computational Bandwidth Bound

An application will be bound by computational bandwidth when the problem requires a large amount of computation compared to the data it is provided. An example would be to check if a number is a prime, or running *Game of Life* [12] for a large number of iterations. In these cases the, processing elements should be working at full load, and the memory bandwidth would not be fully occupied.

In Equation 3.22 one can see the relation between GPU bandwidth and memory bandwidth when the application is bound by the computational bandwidth.

$$b_{gpu} < b_{m \leftrightarrow gm} \tag{3.22}$$

By using Equation 3.4 and 3.22 one can make the reduction given in Equation 3.23 and 3.24.

$$min(b_{m\leftrightarrow r}, b_{gpu}) = min(b_{m\leftrightarrow gm}, b_{gpu}) = b_{gpu} \tag{3.23}$$

$$min(b_{gm\leftrightarrow r}, b_{gpu}) = b_{gpu} \tag{3.24}$$

By reducing Equation 3.18 using Equation 3.23 and 3.24 one gets Equation 3.25 which is the equation for reduction in execution time for a problem which is bound by the computational bandwidth.

$$\begin{aligned} t_{benefit} &= \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{b_{gpu}} - \frac{s_d + s_r}{b_{gpu}} + l_{current} - l_{new} \\ &= \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + l_{current} - l_{new} \end{aligned} \tag{3.25}$$

If we examine the formula, it is clear that the reduction in execution time is equal to the time required to transfer all the data to the GPU. As in the memory bound case, this is due to the pipelining of the task. In this case the computational task is the most time-consuming and thus hides the cost of data transfer.

3.4 Additional Improvements

Using CUDA to develop large data volume applications can be a cumbersome process. To increase the usability of CUDA for such applications, we present a number of improvements aimed at simplifying the development process and enable more compact and understandable code.

3.4.1 Automated Caching

When using data on the GPU there are two main levels in the memory hierarchy, global memory and registers[1] which must be used correctly to obtain best performance. Moving data between them can reduce the overall time spent on data access. However, this approach has its drawbacks, as the developer must explicitly handle this form of caching. This can in many ways be cumbersome when one has to strive to achieve coalesced reads and writes, and can be a a difficult task for the novice CUDA developer. While a hand-optimized pre-fetching can be more optimal than automated caching,

[1]The shared memory can be considered a level parallel to the registers.

it will in most cases be beneficial to have automated caching to ease the development process. By allowing both techniques to be used there is room for both the novice and the expert to utilize the GPU to the best of their knowledge.

While caching is not supported for normal data on current Tesla Architecture, there is still a way to have the GPU handle caching. By claiming that the data is a texture, the GPU seizes control of the data access and caches the data using the Texture-memory. One thing that must be noted is that by marking the data as a texture, it is read-only since there is no write-back on the caching. If cached data are altered the result when accessing the data is undefined. While this limitation is unacceptable in many situations, there are applications which this does not impose a problem.

To use data as a texture, it is only necessary to instruct the GPU to treat the data as a texture. Any data may be handled in this manner. However, the syntax for doing so does not resemble the normal way to handle data access, and it may be confusing to use. Therefore CUDA should include functionality to enable caching for data without referencing textures since this may easily be implemented as a syntactic sugar without any alterations to hardware.

3.4.2 Extended Host-Device Synchronization

The CPU and GPU have different objectives and will therefore continue to have different characteristics. To utilize the computational system optimally, calculations should be performed on the processor that gives the best overall performance of the system. This would in many cases require rapid changes between CPU and GPU calculations and data exchange between them. With CUDA as it is now, this can only be done by stopping the kernel each time the CPU should perform a calculation that the GPU depends on. While this is a solution that enables interaction between CPU and GPU, it is a cumbersome process which complicates the development process. A better solution would be to allow halting the CUDA kernels by synchronizing with the CPU. In this way, there would be a more intuitive interaction between host and device. This can be solved by either actually implementing synchronization in the architecture, or by adding the functionality as syntactic sugar which hides the process of dividing execution into multiple kernels. To efficiently implement the second approach, the cost of initializing a kernel must be reduced so that rapid control changes between GPU and CPU do not affect performance in to large extent. An possibility here is to implement it as syntactical sugar first to see if the developers will use it, and if so implement it in hardware. An approach like this will be less costly as hardware changes are more expensive.

3.4.3 Allow File Access

Both search engines and many other applications require large data volumes which are stored on disk. In the current CUDA environment, the GPU is incapable of accessing files. This means that the CPU must regain control and access the file, and copy the data to the GPU before restarting the kernel. This approach is cumbersome, and will complicate code as described in Section 3.4.2. Enabling file access from the GPU directly can be difficult, as it would require handling IO between the GPU and the disk. There is an easier approach which can be implemented if local memory access is enabled as described in Section 3.1. This approach is to use memory-mapped files, which enables file access from the GPU by masking the file as a memory location, and giving this memory location to the GPU. By doing so, the operating system ensures that the data in the file is accessible to the GPU trough the virtual memory address that the file is mapped to.

Chapter 4

Case Studies of Information Retrieval

The suggested Tesla Architecture improvements given in this thesis try to address performance problems for applications with high data volumes. A search engine is an application with high data volumes and not widely implemented on GPU. However, the research group WestLab at the Polytechnic Institute of the New York University has created a search engine on the GPU [10]. In their article they describe a search engine utilizing both the CPU and GPU to process and sustaining a high arrival rate of queries. In their description of the search engine, they give performance measurements for CPU and GPU versions of both decompressions of the search index, and the actual query evaluation. These two measurements will provide the necessary data to show the improvements made possible by the architectural improvements suggested in this thesis.

In the article by WestLab, they run the benchmarks on a NVIDIA GeForce 8800GTS, which is part of the G80 series from NVIDIA. Even though a GeForce 8800GTS was not tested in [16], it can based on the benchmarks of similar cards and from the information given on NVIDIAs product pages be estimated that the average bandwidth $b_{m \leftrightarrow gm}$ is 2.5 gigabytes per second.

First this chapter gives a general introduction to the search engine and decompression of search indexes, before the implementation of the search engine in [10] is used for two case studies.

4.1 Search Engines

Search engines have over the last decades evolved from simple applications which enabled the user to locate documents in a localized document collec-

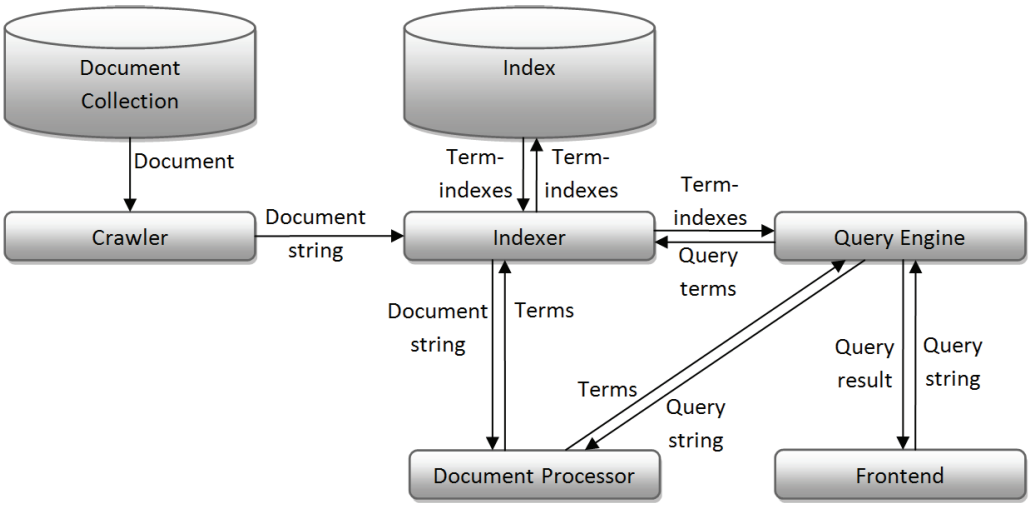


Figure 4.1: The abstract architecture of a search engine.

tions, into today's massive installations with document collections including the dynamic internet. Not only has the size of the document collection increased, but so has the user's expectations to the searching capabilities. Today a search engine is expected to understand what a user is searching for in order to only give results which are semantically relevant to the search. More advanced search engines may also try to interpret the context in which users make the search and then expand the query so that it includes more relevant documents, but with the context of the query and thus also are relevant to the user.

4.1.1 Architecture

The architecture of a search engine can be abstracted into five major components, each with its own role in the system [8]. These five components as shown in Figure 4.1 can be designed independently as long as they share common interfaces among them.

Crawler

The crawler [22] is the component responsible for retrieving indexed the documents. Based on the type of documents and the locality, the crawler may vary a great deal in implementation.

A search engine allowing the users to search the internet such as Google[1] or Yahoo[2], has a complex crawler which traverses the internet and downloads pages which are then provided to the indexer. This crawler also has to make sure it does not index then same page several times, and should discover any updates of the pages. Another important aspect with this crawler is load balancing among the web servers it access. If all the pages the crawler tries to download are from the same server, it can flood the server with request, reducing the total bandwidth with which the crawler can download pages. The crawler must therefore try to balance downloads among servers, to maximize the bandwidth.

A search engine for use on local files such as a library or enterprise may not need such a complex crawler. For them, it may suffice with a crawler which traverses a file structure, or is provided a list of documents to crawl.

Indexer

The indexer is responsible for managing all access to the index. It must both provide access for the query engine so it can retrieve the term-indexes needed to answer the query, and access to the crawler, so it can insert its documents. If the index is distributed across several disks or nodes, it is also the indexer's job to distribute the index correctly.

Document Processor

The job of the document processors is to take a document or a query as a string, and process it to extract the terms, supplying these to either the indexer or the query engine [37]. The reason the document processor is used both by the query engine and the indexer is to ensure that the terms extracted are equal. The document processor is usually a series of sequential steps which manipulates the document.

Tokenizing is usually the first step in the document processor. It reads the entire string and splits it into terms. It may also remove unwanted sections such as HTML-tags and special characters.

Stop-word Removal is the process of removing terms from the list of terms which are too common to be needed in the index. These terms are removed since it would dramatically increase the size of the index if they were included, and a query with these words in it would not notice any contribution

[1] www.google.com
[2] www.yahoo.com

from the terms to the results due to the vast number of occurrences of the term in the document collection. Examples of stop-words are: *we, you, it, is, was* and *over*. There is a debate if stop-word removal is useful, as it makes the process of evaluating phrase queries more difficult.

Stemming is the process of trying to reduce similar words to their base or stem, so they would be indexed as the same term. This is done to reduce the number of terms in the collection, and to improve the quality of the search. For instance the three words *consort, consortedi* and *consorting* may all be reduced to the stem *consort*. The Porter stemmer [36] is an example of such a stemmer.

Lemmatization is the same process as stemming, except that the stemmer only operates on single terms, and thus has no knowledge of its context, while Lemmatization takes into account the context of the term, to avoid using the same term for different context.

Query engine

The query engine is the component which performs the actual search [5]. It accesses the index, retrieves the term-indexes needed to complete the query, and calculates a score for all the documents in the document collection. It then sorts the documents based on the score, and returns the desired number of documents to the frontend.

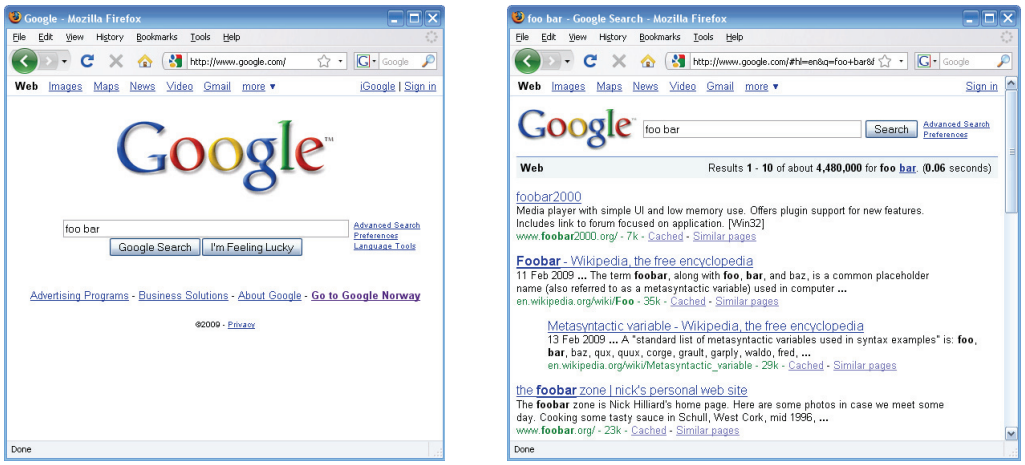
Frontend

The frontend is responsible for providing the user with an interface to the search engine. In the simplest form it must provide a textual input where the user can type in the query, and then present the result as a list of documents or links to the documents.

4.1.2 Inverted Index

The inverted index is a data structure which optimizes for lookup of documents in which a term occurs. This is done through the *inverted* approach where each occurrence of a term in the documents are listed grouped for each term [38].

The simplest form of inverted index stores the frequency f of each term in each document, while more advanced indexes stores the location of the terms in the documents as well as other relevant information.



(a) Query Input (b) Query Result

Figure 4.2: Google Web Search frontend

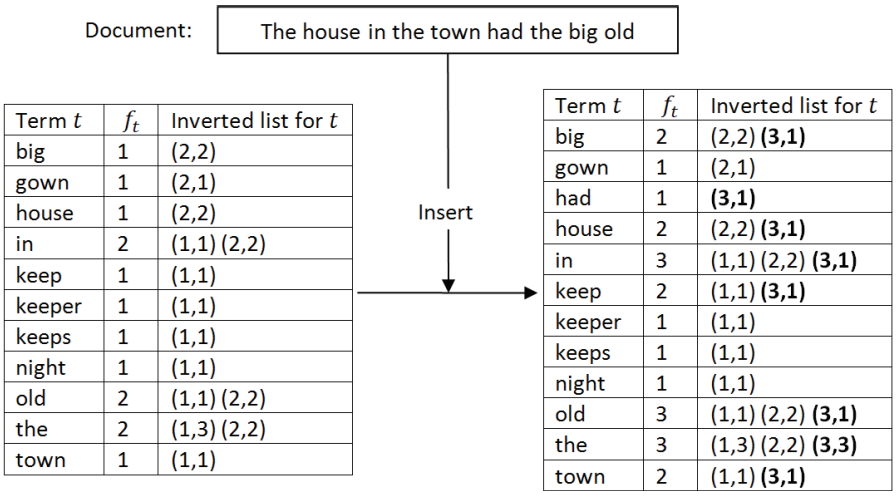
Document-level Inverted Index is the simplest form of inverted index. It only stores the number of occurrences of each term in the documents, and thus does not support proximity and phrase search.

Word-level Inverted Index stores the position of the terms in the document, and is thus able to support proximity and phrase search as well.

Block-level Inverted Index can be compared to a word-level index where the positions are not exact but instead point to the block of the document where the term occurs. This reduces the size of the index, but also removes information needed by proximity and phrase search. To still be able to support these two searches, the occurrences of terms within blocks can be used to identify blocks where phrases can occur, and analyze these parts of the documents online. To be able to do this efficiently, the blocks must be small enough, which increases the size of the index.

Creation

To create an inverted index, each document is processed independently. This processing passes over the document once, to find its terms and insert the relevant information into the index.



Term t	f_t	Inverted list for t
big	1	(2,2)
gown	1	(2,1)
house	1	(2,2)
in	2	(1,1) (2,2)
keep	1	(1,1)
keeper	1	(1,1)
keeps	1	(1,1)
night	1	(1,1)
old	2	(1,1) (2,2)
the	2	(1,3) (2,2)
town	1	(1,1)

Term t	f_t	Inverted list for t
big	2	(2,2) **(3,1)**
gown	1	(2,1)
had	1	**(3,1)**
house	2	(2,2) **(3,1)**
in	3	(1,1) (2,2) **(3,1)**
keep	2	(1,1) **(3,1)**
keeper	1	(1,1)
keeps	1	(1,1)
night	1	(1,1)
old	3	(1,1) (2,2) **(3,1)**
the	3	(1,3) (2,2) **(3,3)**
town	2	(1,1) **(3,1)**

Figure 4.3: Insertion in Document-Level inverted index

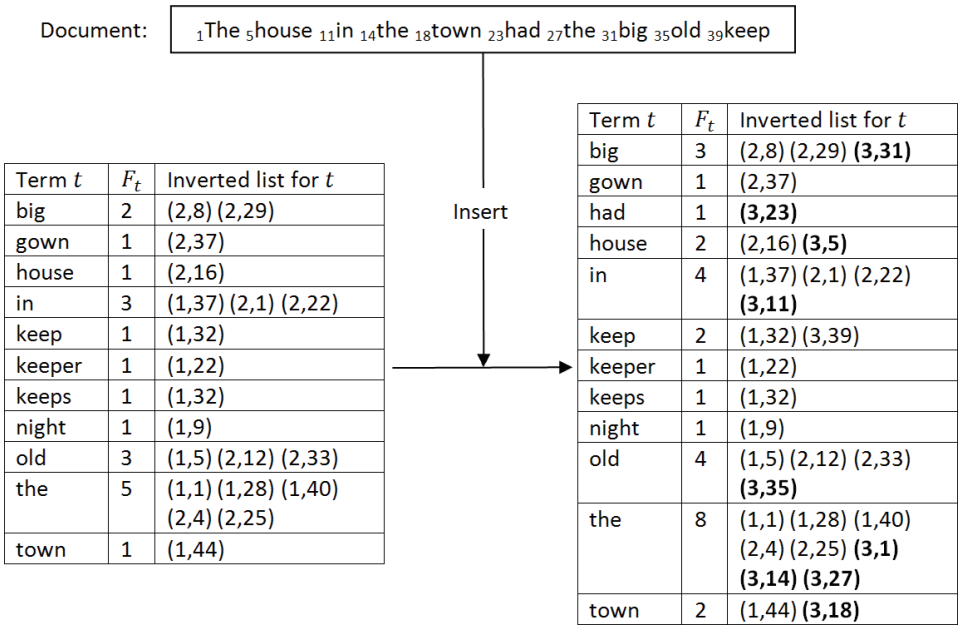
Document-level Inverted Index stores the frequency of each term in each document. To add such a document to the index is a trivial task. The document is read and the number of occurrences of each term is found. Then for each term, a pair (document-ID, frequency) is added to the index under its respective term. Such a process can be seen in Figure 4.3.

Word-level Inverted Index stores the position of the terms in the inverted index. This allows for proximity and phrase search. To add a document to this type of index, the document is read, and for each term, the position of the first character in the term is added to a list of position for each term. Then for each occurrence of each term, a pair (document-ID, position) is added to the index under its respective term. This process can be seen in Figure 4.4.

Block-level Inverted Index is created in the same way that the word-level index, except that instead of storing the exact position, the block number is used. Insertion of a document into this index can be seen in Figure 4.5.

Storing

The indexes may grow to such large sizes that it is not possible to store the entire index in memory. Thus, a file structure must be used to allow the index to be stored on disk while caching the required parts in memory. Due to the low performance of disks, file access must be kept to a minimum. One



Term t	F_t	Inverted list for t
big	2	(2,8) (2,29)
gown	1	(2,37)
house	1	(2,16)
in	3	(1,37) (2,1) (2,22)
keep	1	(1,32)
keeper	1	(1,22)
keeps	1	(1,32)
night	1	(1,9)
old	3	(1,5) (2,12) (2,33)
the	5	(1,1) (1,28) (1,40) (2,4) (2,25)
town	1	(1,44)

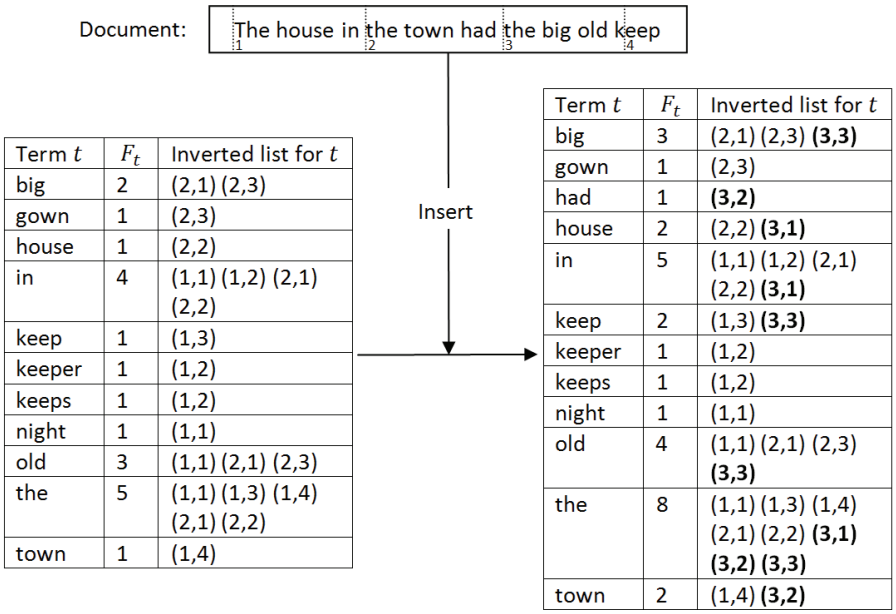
Term t	F_t	Inverted list for t
big	3	(2,8) (2,29) **(3,31)**
gown	1	(2,37)
had	1	**(3,23)**
house	2	(2,16) **(3,5)**
in	4	(1,37) (2,1) (2,22) **(3,11)**
keep	2	(1,32) (3,39)
keeper	1	(1,22)
keeps	1	(1,32)
night	1	(1,9)
old	4	(1,5) (2,12) (2,33) **(3,35)**
the	8	(1,1) (1,28) (1,40) (2,4) (2,25) **(3,1) (3,14) (3,27)**
town	2	(1,44) **(3,18)**

Figure 4.4: Insertion in Word-Level inverted index

way of doing this is through efficient use of caching. The other way is to have an efficient storage of the index on disk. Due to the size of the index, it is preferable to be able to directly find the place in the file where the required parts are stored. This can be achieved through a term-position list, where the position of each term's index in the file is stored. The size of the term-index is also stored to know how large the term-index is. To retrieve the index for a single term, one looks up the term in the term-position list to get the position in the file, and then directly find the correct position.

When updating the index, its size may change, and new terms or occurrences may be added or removed. This would require that the changes are updated in the file. Since the file is stored sequentially on the disk, this would require a complete rewrite since the updates may occur anywhere in the file. As the size of the file grows, this may be an expensive operation. A feature which may be added to reduce this cost, is to store the index in multiple files. By doing so, only the files which are altered must be rewritten. The index is usually spilt in such a way that each file only contains some of the term-indexes.

Another approach to increase disk access performance is to distribute the index across several disks. By distributing the index, parallel access to the disks is enabled, and thus an increase in bandwidth. One such distribution



Term t	F_t	Inverted list for t
big	2	(2,1) (2,3)
gown	1	(2,3)
house	1	(2,2)
in	4	(1,1) (1,2) (2,1) (2,2)
keep	1	(1,3)
keeper	1	(1,2)
keeps	1	(1,2)
night	1	(1,1)
old	3	(1,1) (2,1) (2,3)
the	5	(1,1) (1,3) (1,4) (2,1) (2,2)
town	1	(1,4)

Term t	F_t	Inverted list for t
big	3	(2,1) (2,3) **(3,3)**
gown	1	(2,3)
had	1	**(3,2)**
house	2	(2,2) **(3,1)**
in	5	(1,1) (1,2) (2,1) (2,2) **(3,1)**
keep	2	(1,3) **(3,3)**
keeper	1	(1,2)
keeps	1	(1,2)
night	1	(1,1)
old	4	(1,1) (2,1) (2,3) **(3,3)**
the	8	(1,1) (1,3) (1,4) (2,1) (2,2) **(3,1)** **(3,2)** **(3,3)**
town	2	(1,4) **(3,2)**

Figure 4.5: Insertion in Block-level inverted index

scheme is discussed in [21]

Distribution

As the size of the index grows, there is need for more computational power than can fit in one computer to process the query efficiently. To do so, the index can be distributed among several nodes. The inverted index is especially well suited for distribution, and there are two possible distribution schemes; term distribution and document distribution [38, 4].

Document Distributed Index is the simplest and most used scheme. This scheme assigns the documents in the collection to the various nodes so each node has a subset of the documents. When querying against this distributed index, all nodes must perform the query on their local index, and return the highest document scores back to the master node.

Term Distributed Index distributes its terms across the nodes. This means that a query may not need to access all nodes to be answered. Since one wants to group terms correctly on the nodes so the most frequent queries are answered using few nodes. The reason that this is important is that a single node can not answer a query and must transfer partial scores for all

documents to the master node for accumulation there, and this can be an expensive operation.

Term and Document Distributed Index is a mixture of the two schemes. These two schemes can be used together to tune the performance of the system. One can, for instance, create several clusters which distribute the document collection among themselves using document distribution, but internally using term distribution.

Compression

Another way to increase the performance of the search engine is to use compression on the inverted index [34, 30]. Since the search engine in most cases is limited by the poorly performing hard-disks, the use of compression can reduce the size of each individual transfer, and therefore the speed of the transfer. If this approach is to yield overall improvement for the search engine, it requires that the decompression consumes less time than the reduction of transfer time.

The use of compression in inverted indexes utilizes some properties in the index which enables the use of a certain group of compression schemes. The inverted index most often contains numbers which probability to occur matches well with a Bernoulli trial. Therefore algorithms such as Variable-Byte encoding and Golomb coding can be used with good compression rates. These algorithms will be discussed further in Section 4.2.

4.2 Compression

The use of compression varies from application to application, and so does the compression scheme used. While most think of compression as a method of saving space, compression is widely used to increase the speed of transfers by reducing the size of what is transferred.

The choice of which compression scheme to use must take into account the types of data that are to be compressed and the encoding and decoding performance requirements. The difference between a well suited and ill-suited scheme can have huge impact on application performance.

This section will discuss some compression schemes which are used to compress a series of integers where the probability of a number conforms to a Bernoulli trial.

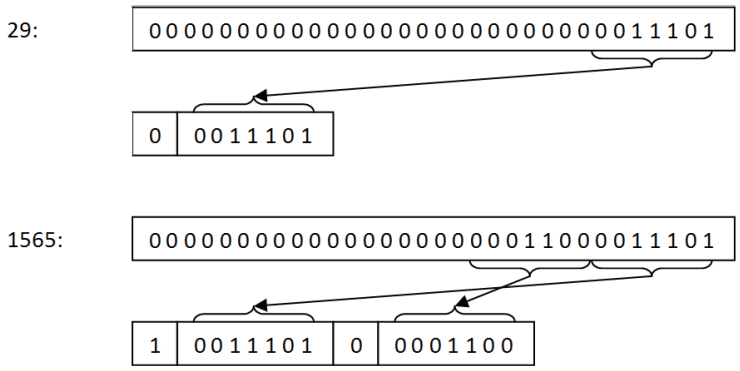


Figure 4.6: Encoding of the number 29 and 1565 in Variable-Byte encoding.

4.2.1 Variable-Byte Coding

Variable-Byte coding [30] is a byte-level encoding which is simple to use, and yields good compression results. By encoding numbers with a varying number of bytes, the encoding scheme allows certain numbers to be stored with a lower number of bytes than it would using no compression, but still maintaining an easy access to the numbers through a simple encoding and decoding process.

The variable-byte scheme uses seven out of the eight bits in a byte to store the number, while the remaining bit is used to indicate if another byte is needed to represent the number. By using this encoding scheme any number $[0, 128)$ can be represented using one byte, while 2^{32} must be represented using five bytes. This encoding scheme can be seen in Figure 4.6.

As one can see in Figure 4.7, Variable-Byte encoding favors small numbers. In fact, if there is equal probability for any number in the range $[0, 2^{32})$, it would use 4.93 bytes per number, which is more than the uncompressed number. If one however assumes that a number $n+1$ is half as likely to occur as n, then this encoding scheme would use 1.008 bytes on average.

The encoding and decoding using this scheme is a simple task. As one can see in Figure 4.6, there is a simple mapping between the decoded and coded byte sequence. To encode and decode a number, Algorithm 1 and Algorithm 2 are used, respectively.

Figure 4.6 shows how the encoding scheme is byte aligned, which is a great benefit for several reasons. It is for instance possible to start reading anywhere in the encoded sequence and easily identify a number to start decoding. When doing this, one does not know which number one is decoding. If decoding an encoded sequence in parallel, one can divide the sequenced into

Algorithm 1 Encode number using Variable-Byte encoding.

$v_{original} \leftarrow getNumber()$
$v_{part} \leftarrow v_{original}$ & 127
$v_{remaining} \leftarrow v_{original} \gg 7$
$b \leftarrow 0$
$i \leftarrow 0$
repeat
  if $v_{remaining} = 0$ **then**
    $m \leftarrow 0$
  else
    $m \leftarrow 1$
  end if
  $b \leftarrow b + (m \ll i + 7) + (v_{part} \ll i)$
  $i \leftarrow i + 8$
until $v_{remaining} = 0$
return v

Algorithm 2 Decode Variable-Byte encoded number.

$v \leftarrow 0$
$b \leftarrow getByte()$
$i \leftarrow 0$
repeat
  $v \leftarrow v + (b$ & $127) \ll i$
  $i \leftarrow i + 7$
until b & $128 = 0$
return v

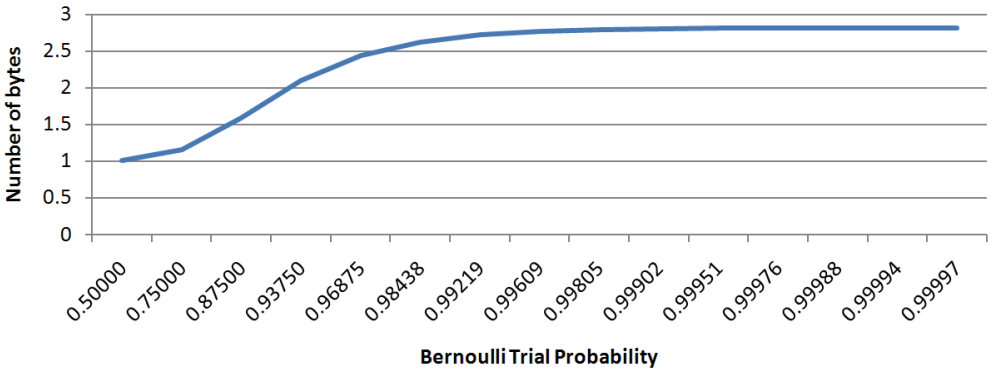


Figure 4.7: Average size for Variable-Byte encoded number for given Bernoulli trial probability.

several parts, and let each processor decode its segment. After the decoding process is completed, the processors must then align the decoded numbers correctly in a resulting table. This would of course require an additional overhead compared to a sequential decoding of the sequence, but may still give a good speedup.

4.2.2 Golomb Coding

Golomb coding is a bit-wise compression scheme which was first described by Solomon Golomb in 1966 [14]. By using a encoding similar to Huffman Coding, Golomb Coding manages to encode an infinite set of outcomes, breaking a limitation of Huffman Coding [19].

The Golomb code is built using two parts which can be combined to reconstruct the encoded number. The first part is a number $q + 1$ where $q = \lfloor \frac{x-1}{b} \rfloor$. The second part is the remainder $r = x - qb - 1$. The first part is encoded using unary code, which is a simple encoding scheme where a number n is encoded using n consecutive 1-bits, followed by a single 0-bit. The second part is encoded using standard binary encoding. It is here possible to some bits if the set of integers needed to be represented is less than the amount possible by the given number of bits. The parameter b is used to adjust the number of bits used in the second part, where the required number of bits in the second part is either $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$. How the parameter b affects the encoding can be seen in Table 4.1.

In 1975, Gallager and Van Voorhis [11] showed that Golomb Coding produces optimal compression for geometric distributions when b is chosen in accordance with Equation 4.1 where p is the probability of success for the

Table 4.1: Numbers encoded using Golomb encoding with various values for b [14]

n	$b = 1$	$b = 2$	$b = 3$	$b = 4$
0	0	00	00	000
1	10	01	010	001
2	110	100	011	010
3	1110	101	100	011
4	11110	1100	1010	1000
5	111110	1101	1011	1001
6	1111110	11100	1100	1010
7	11111110	11101	11010	1011
8	111111110	111100	11011	11000
9	1111111110	111101	11100	11001
10	11111111110	1111100	111010	11010

Bernoulli trial corresponding with the geometric series.

$$(1-p)^b + (1-p)^{b+1} \leq 1 \leq (1-p)^{b-1} + (1-p)^b \tag{4.1}$$

A special case of the Golomb Coding occurs when $b = 2^k$ for some integer k. In this special case, the coding scheme becomes much simpler, which was showed by Robert F. Rice in 1979 [31, 18]. In this case, the k least significant bits are used as the second part of the Golomb code, while the remaining bits are encoded using unary code and used as the first part of the Golomb code. The Rice coding is more widely used then the general Golomb Code due to the especially simple characteristics of the Rice code. As an example, is the Rice coding used in the Fast Enterprise Search Platform[3] [29], which is based on the FMS Search Engine Kernel [32].

4.2.3 PForDelta

Modern CPUs have a long pipeline, which can speed up the execution of a program if used properly. To achieve these speedups, the program strive to organize code in a manner thar efficiently can be executed in a pipeline. Branching is among the instructions which is difficult to pipeline, and this is often used in decompression. PForDelta [39] is a compression scheme

[3]http://fastsearch.com/l3a.aspx?m=1031

which can be implemented without branching in its inner loops allowing the compiler and CPU to optimize for the pipeline.

To remove the branching from the inner loops, PForDelta processes numbers in groups of 128. By doing so, all 128 numbers can be decoded without branching, making the inner loop faster. To allow this no-branching decoding to occur, PForDelta compresses all 128 numbers with the same number of bits, gaining the benefits of a fixed width compression within the 128 numbers. While compressing all numbers with a fixed width would in many cases lead to large bit widths, PForDelta avoids this by using exceptions. The bit width is chosen so that 90 percent of the numbers fit, and the rest are stored as full integers using exceptions at the end of the list. To identify which numbers are handled by an exception, a number which gives the offset to the first exception is stored first in the compressed segment. In that position, the offset to the next exception is stored. In this way, the exceptions are stored as a linked list within the list itself.

To decode this scheme without branching, two loops are used. The first loop decodes all the numbers as if there were no exceptions. This leads to errors for the numbers handled by exception. The second loop loops over the linked list giving the exceptions, and inserts the correct numbers from the exception list into the resulting uncompressed list of numbers.

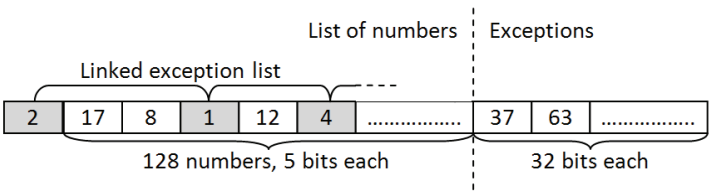


Figure 4.8: Exception handling in PForDelta compression scheme.

Figure 4.8 gives an example of the PForDelta compression scheme. In this example the bit width is set to 5 bits, and the numbers 37 and 63 are encoded using exceptions. The first number in the list gives the location of these two exceptions in the list. The actual exceptions are stored at the end of the compressed data.

4.3 Case Study: Decompression of Inverted Index

One of the tasks often performed in a search engine is to decode the inverted index. The index is compressed to reduce the bandwidth requirement for the hard-disks. By offloading this task to the GPU, the CPU would be free to perform other tasks, thus increasing the throughput of the search engine.

In the article [10], both Rice coding and PForDelta have been used for compression in the search engine, and the article provides enough data to give an estimated performance gain using the architecture suggested in this thesis. It is, however, worth noticing that the GPU versions perform more work than the CPU version, since the CPU version finds the gaps between docIDs while the GPU version finds the actual docIDs. Therefore, any comparison between them will be slightly biased towards the CPU.

In WestLabs article [10], they perform the decompression either on the GPU or the CPU based on which processor is used for the intersection of the search indexes. The GPU and CPU benchmarks assume that the data is present in global memory and local memory respectively. The decompression case tested here assumes that the data resides in local memory and thus has to be copied to the GPU before any decompression can be performed. Therefore, the cost of the data transfer must be added to the total cost for the GPU version.

By using the compression ratio c_{Rice} and $c_{PForDelta}$ the total data transfer can be found based on the chosen number of integers n in the search index. This is given in Equation 4.2 and 4.3.

$$s_d = 4nc \tag{4.2}$$

$$s_r = 4n \tag{4.3}$$

The measured compression values as given in the article are 0.61 bytes per integer for Rice coding and 0.64 bytes per integer for PForDelta coding.

The measured performance the actual decompression is given in Table 4.2 as millions of integers per second. For the GPU, this result $b_{gpucard}$ is the measured throughput on the GPU including data transfers and computation and is equal to $min(b_{gm\leftrightarrow r}, b_{gpu})$.

For the CPU version the total time needed to perform a decompression is given in Equation 4.4.

$$t_{cpu} = \frac{n}{b_{cpu}} + l_{cpu} \tag{4.4}$$

Table 4.2: Millions of integers per second decompressed by the GPU and CPU.

Algorithm	b_{cpu}	$b_{gpucard}$
Rice	310.63	305.27
PForDelta	1165.13	1237.57

By exchanging values in Equation 3.14 and 3.16 formulas for expected time to perform decompression on the GPU using the current and suggested Tesla Architecture can be obtained.

$$\begin{aligned} t_{current} &= \frac{s_d + s_r}{b_{m \leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm \leftrightarrow r}, b_{gpu})} + l_{current} \\ &= \frac{4nc + 4n}{b_{m \leftrightarrow gm}} + \frac{4nc + 4n}{b_{gpucard}} + l_{current} \end{aligned} \tag{4.5}$$

$$\begin{aligned} t_{new} &= \frac{s_d + s_r}{min(b_{m \leftrightarrow r}, b_{gpu})} + l_{new} \\ &= \frac{4nc + 4n}{b_{gpucard}} + l_{new} \end{aligned} \tag{4.6}$$

By evaluating Equation 4.5 and 4.6 for both Rice and PForDelta, and assuming no latency a comparison of the different approaches can be made. The assumption of no latency can be made due to large data volumes hidign the latency.

In Table 4.3 the speedups of the GPU versions on both architectures is given against the CPU versions and the speedup of the suggested architecture against the current architecture is also given.

As one can see from Table 4.3, the suggested architecture gives a increased performance. On the original GPU architecture, offloading the decompression to the GPU would not be beneficial, as performance would decrease. However on the suggested architecture, the performance of decompression on the GPU would be as fast as on the CPU, thus allowing offloading without performance reduction.

4.4 Case Study: Query Evaluation

Evaluating a query is a process with several steps. Firstly, the search index must be decompressed, allowing the data to be read. Thereafter, the search

Table 4.3: Speedups of the suggested architecture for Rice and PForDelta decoding

	Rice coding	PForDelta coding
Current GPU vs CPU	0.639	0.333
Sugested GPU vs CPU	0.983	1.062
Suggested GPU vs current GPU	1.537	3.191

indices for each term must be combined to find the relevant documents. Finally the top k documents are returned. In the search engine [10] WestLab chose to return the top 10 documents. These steps can be done sequentially, it is beneficial to interleave the steps by handling one document at the time. See the article [10] for details on how this is done.

In the search engine a search index which contains a docIDs and a frequency for each posting is used. The index is built over the TREC GOV2 dataset which contains 25.2 million web pages. A random set of queries associated with the dataset, and on average the search indexes associated with the query contains 3.74 million postings denoted n.

In Equation 4.7 and 4.8, the sizes of data needed by each query evaluation is given.

$$s_d = 8nc \tag{4.7}$$

$$s_r = 4k \tag{4.8}$$

The timings of the query evaluation on both GPU (t_{gpu}) and CPU (t_{cpu}) are given in Table 4.4, along with the bandwidth b_{cpu} and $b_{gpucard}$ of the operation based on the datasize s_d and s_r. Three different scoring schemes AND, OR and AND+OR are used, which are explained in the article. In the benchmarks it is assumed that the data is present on the local memory for the CPU version and in global memory for the GPU version. The cost of data transfers must therefore be added in the calculated timings for the current and suggested Tesla Architecture. Timings for the CPU version can be used directly.

By exchanging values in Equation 3.14 and 3.16, formulas for expected time to perform the query evaluation on the GPU using the current and suggested Tesla Architecture can be obtained.

Table 4.4: Timings and bandwidth for scoring schemes on CPU and GPU.

	AND	OR	AND+OR
CPU timings (ms)	8.71	212.72	23.85
GPU timings (ms)	7.66	29.31	9.98
CPU bandwidth (B/s)	3435136.62	140654.57	1254509.01
GPU bandwidth (B/s)	3906010.44	1020813.37	2998000.00

$$\begin{aligned} t_{current} &= \frac{s_d + s_r}{b_{m \leftrightarrow gm}} + \frac{s_d + s_r}{min(b_{gm \leftrightarrow r}, b_{gpu})} + l_{current} \\ &= \frac{8nc + 4k}{b_{m \leftrightarrow gm}} + \frac{8nc + 4k}{b_{gpucard}} + l_{current} \end{aligned} \tag{4.9}$$

$$\begin{aligned} t_{new} &= \frac{s_d + s_r}{min(b_{m \leftrightarrow r}, b_{gpu})} + l_{new} \\ &= \frac{8nc + 4k}{b_{gpucard}} + l_{new} \end{aligned} \tag{4.10}$$

By inserting the values for the respective scoring schemes, the speedups of the query evaluation on the suggested architecture compared to the CPU and current architecture versions can be determined. These are presented in Table 4.5.

Table 4.5: Speedups of the suggested architecture for AND, OR and AND+OR scoring schemes

	AND	OR	AND+OR
Current GPU vs CPU	1.055	5.504	1.648
Sugested GPU vs CPU	1.763	7.647	3.090
Suggested GPU vs current GPU	1.671	1.389	1.874

As in the previous case, the performance gain by using the improved Tesla Architecture is clearly visible.

The query evaluations performed on the GPU so far assume that the data is not available on the GPU, and that they are transferred to the GPU for

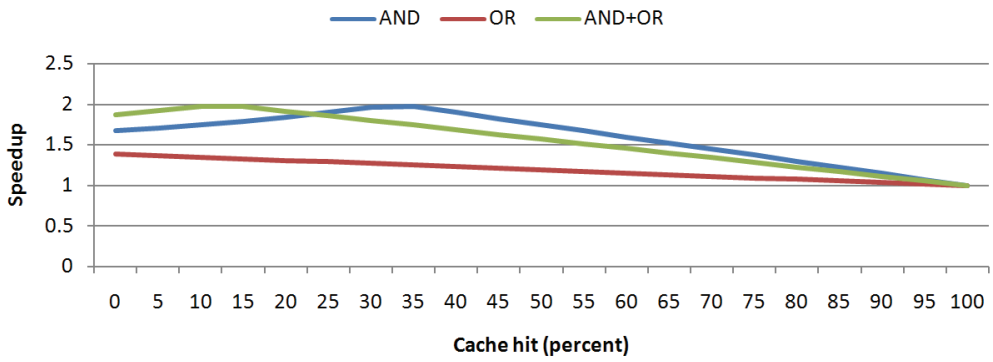


Figure 4.9: Speedup of the scoring schemes on the suggested architecture compared to the current architecture for different levels of cache hits.

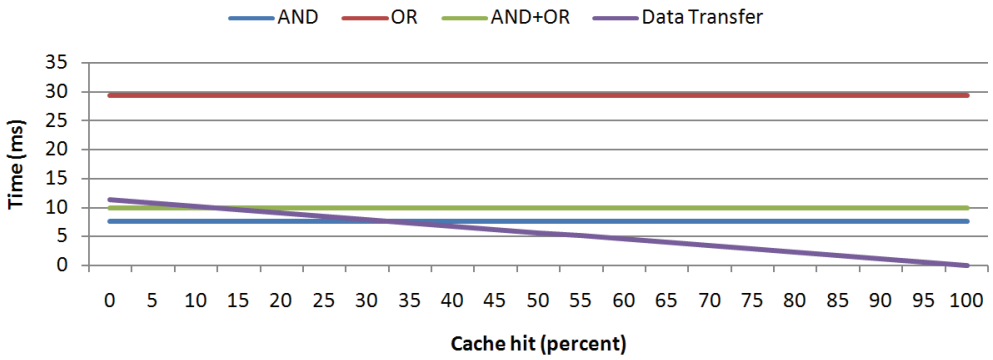


Figure 4.10: Time required by the different scoring schemes and the data transfer for different levels of cache hits.

each query evaluation. In practice, the memory on the GPU is used as a cache drastically reducing the amount of data needed to be copied. How often the cache will contain the data needed depends on both the cache replacement scheme and the queries made to the search engine. In Figure 4.9, the speedup of the suggested architecture versus the current architecture is given when the cache hit ratio varies between 0% and 100%.

In Figure 4.9 one can see a reduction in speedup after a certain point. These points are shown in Figure 4.10 to be when the problem changes from bandwidth bound to computational bound.

As one can see from this case, the suggested architecture gives an improvement here as well, but it is also shown that the benefits of the suggested architecture lies in data transfers. The less data transfer needed, the

less beneficial are the improvements. The case also shows that the improvements would allways provide at least the same performance as the current architecture, but in most cases will yield an increase in performance.

Chapter 5

Conclusion and Future Work

This thesis has pointed out the need for certain features on the GPU, which would improve performance for large data volume applications, through analysis and a theoretical model. By including the host-memory in the memory hierarchy of the GPU, new ways to access data during calculations can be developed. Other benefits which reduce the complex code of large data volume applications have also been suggested. This chapter concludes on how likely these features are to be realized, and what work lies ahead in the process of providing these features.

5.1 Realizing the Improvements

Adding support for the GPU to access host-memory may seem to be a difficult task, but NVIDIA supports a variant of this in hardware on its newest GPU series. The concept is called Zero-Copy, and it allows the user to access pinned host-memory. CUDA does not yet support this, but it is included in the beta version of CUDA 2.2 [25]. While this method to implement host-memory access partially, there is a limitation since it only allows access to pinned memory. The amount of pinned memory is usually restricted, and would thus not allow full use of host-memory efficiently in a combined CPU and GPU application. It will, however, provide the same benefits as described in this thesis in terms of streaming capabilities which will improve performance. If the developers see the potential of Zero-Copy and start to utilize it in their applications, it may be the first step towards allowing more and more interaction and resource sharing between CPU and GPU. While Zero-Copy will resolve many of the issues addressed in this thesis, there is still a limitation since only pinned memory can be used. By allowing the GPU access to the entire host-memory, GPU developers will have greater

freedom to utilize the large amount of available memory on modern systems.

Among the other improvements suggested in this thesis, many should be possible to implement. Synchronization can be solved using syntactic sugar. Even though using syntactic sugar would enable the improvement, it would not be the ideal approach to solve it. If synchronization is handled through syntactic sugar, the developer loses the ability to hand-optimize the code. This would be done by the compiler, which in some cases may do the program less efficient. If the syntactic sugar approach is chosen, it should therefore not exclude the manual approach. The most desirable solution would be to support it efficiently through hardware flags or some other mechanism, but this can be cumbersome and require redesign of the architecture. We therefore suggest that the changes which could be provided through syntactic sugar are first added in this manner. In this way there is a low-cost opportunity to see the potential in the improvement, and if it is a success it can be fully supported in hardware.

Caching on the GPU is enabled in current versions to some extent, as textures and constant memory is cached. This memory is read-only and will therefore never be a fully acceptable replacement for global memory and manual caching. While some applications can benefit greatly from the cached textures, many applications requires the ability to alter the data during execution. Implementing cache with write-back or write-through can be a difficult task but would be a great benefit to the developers. As was shown in Figure 2.7, allowing the GPU to handle data access through hardware caching outperforms the explicit approach.

Theis thesis points out that if the GPU gains access to the entire host-memory, it would also gain access to files on hard-disks through the use of memory mapped files. While Zero-Copy grants access to pinned memory on the host, it would not enable file access, since memory mapped files are paged. To allow file access by the GPU, it must either be developed new access methods specific for the GPU or allowing the GPU to access the entire host-memory. As stated previously in this thesis, large data volume applications use files frequently, and enabling file access for the GPU would be a great benefit.

5.2 Benefits of the Improvements

While the benefits of the improvements suggested in Section 3.4 are hard to quantify, they would provide the developer with a larger set of options when developing applications on the GPU. These improvements were suggested with large data volume applications in mind, but would be beneficial to

many other types of applications as well. It is easier to discuss the benefits from expanding the memory hierarchy to include the host-memory as an improvement. Through a mathematical model, it has been shown that by allowing the GPU to access the host-memory, data transfer and calculations can be interleaved much tighter than what is possible in current versions of CUDA. This would allow the developer to hide the cost of certain operations since these can be done in parallel in the created pipeline. Through the mathematical model described in Section 3.2 it is shown that there are two major costs in a GPU program: data transfer and calculations. By allowing these operations to be interleaved on a fine grained scale, the cost of the cheapest one can be hidden by the other operation.

To show how beneficial this ability could be, a search engine [10] by West-Lab, NYU, has been studied. Based on the benchmarks made by WestLab it was possible to calculate the performance gain by extending the memory hierarchy. For the search engine with the improved GPU it was shown a speedup between 1.389 and 1.874 by the various query types over the current GPU. When comparing the improved GPU versus the CPU this gives a speedup between between 1.763 and 7.647. In the case with using the GPU as an accelerator for search index decompression, the increased memory hierarchy would make the GPU handle decompression at the same rate as the CPU. With the current GPU this was not possible, as the data transfer would be too expensive. This result shows that with the increased memory hierarchy, the GPU would be able to function as an accelerator card for large data volume applications since it would offload the CPU without reducing performance.

5.3 Future Work

This thesis focused on suggesting improvements to the Tesla architecture and CUDA programming environment which would benefit large data volume applications, and the improvements and approaches chosen to implement them is biased by this viewpoint. The GPU is used in a wide range of general purpose applications and improvements must be analyzed with respect to a these applications to see if there are modifications to the approaches which will be more beneficial on a global scale. A poorly designed feature may not be used correctly, or not used at all. It is therefore vital to make sure the features are easy to use and suited for most areas which the GPU may be used within.

When version 2.2 of CUDA is released, the theoretical model should be validated for CUDA with Zero-Copy, and possible adjustments should be

made to the model.

5.4 Final Thoughts

This thesis has suggested several features that could be added to CUDA and the Tesla Architecture to increase performance of large data volume applications. The suggested improvement will ease the development process of such applications. As general-purpose calculations on GPUs enters new areas of computer science, these benefits would prove to be valuable. If one or more of these features are implemented in actual GPUs it would be a step towards tighter integration and cooperation between the CPU and GPU, a step we consider to be in the right direction. This will allow both processors to be used for their intended tasks, and together form an efficient computational system.

Bibliography

[1] Advanced Micro Devices, Inc., "Technical Overview, Stream Computing," [Cited: January 17, 2009]. [Online]. Available: http://ati.amd.com/technology/streamcomputing/Stream_Computing_Overview.pdf

[2] G. M. Amdahl, "Validity of the Single Processor Apporach to Achieving Large Scale Computing Capabilities," in *Proceedings of the 30th AFIPS Spring Joint Computer Conference*, Atlantic City, N.J., April 1967, pp. 483–485.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yellick., "The Landscape of Parallel Computing Research: A view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep., 2006.

[4] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani, "Distributed Query Processing Using Partitioned Inverted Files," in *Symposium on String Processing and Information Retrieval*, Laguna De San Rafael, Chile, November 2001, pp. 10–20.

[5] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 1st ed. Addison Wesley Longman Limited, 1999.

[6] D. Blythe, "The Direct3D 10 System," *Proceedings of the ACM SIGGRAPH 2006*, vol. 25, no. 3, pp. 724–734, July 2006.

[7] ——, "Rise of the Graphics Processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–777, May 2008.

[8] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems*, vol. 30, pp. 107–117, April 1998.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, October 2008.

[10] S. Ding, J. He, H. Yan, and T. Suel, "Using Graphics Processors for High Performance IR Query Processing," in *Proceedings of the World Wide Web Conference 2009*, Madrid, Spain, April 2009, pp. 421–430.

[11] R. G. Gallager and D. C. V. Voorhis, "Optimal Source Code for Geometrically Distributed Integer Apphabets," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 228–230, March 1975.

[12] M. Gardner, "The Fantastic Combinations of John Conways's New Solitare Game "Life"," *Scientific American*, no. 223, pp. 120–123, October 1970.

[13] S. Gobron, H. Bonafos, and D. Mestre, "GPU Accelerated Computation and Visualization of Hexagonal Cellular Automata," in *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, vol. 5191, pp. 512–521.

[14] S. W. Golomb, "Run-Length Encodings," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, July 1966.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann Publishers, 2007.

[16] R. J. Hovland, "Latency and Bandwidth Impact on GPU-systems," December 2008, report in course "TDT4590 Complex Computer Systems, Specialization Project", Department of Computer and Information Science, Norwegian University of Science and Technology. [Online]. Available: http://wo.uio.no/as/WebObjects/frida.woa/wo/26.Profil.29.25.2.3.15.1.2.3

[17] ——, "Branch Performance on the Tesla Architecture," April 2009, report in course "TDT4260 Computer Architecture", Department of Computer and Information Science, Norwegian University of Science and Technology. [Online]. Available: http://wo.uio.no/as/WebObjects/frida.woa/wo/3.Profil.29.25.2.3.15.1.0.3

[18] P. G. Howard and J. S. Vitter, "Fast and Efficient Lossless Image Compression," in *Data Compression Conference*, Snowbird, U.T., USA, March/April 1993, pp. 351–360.

[19] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[20] IEEE Computer Society, "IEEE Std 1003.1-2008, Standard for Information Technology. Portable Operating System Interface (POSIX). Base Specifications, Issue 7," 2008.

[21] B.-S. Jeong and E. Omiecinski, "Inverted File Partitioning Schemes in Multiple Disk Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 142–153, February 1995.

[22] M. Kobayashi and K. Takeda, "Information Retrieval on the Web," *ACM Computing Surveys*, vol. 32, no. 2, pp. 144–173, 2000.

[23] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March-April 2008.

[24] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.0," [Cited: January 17, 2009]. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[25] ——, "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.2," [Cited: May 25, 2009]. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf

[26] NVIDIA Forums Register Users, "Cuda 2.2 / Zero-copy access," 2009, [Cited: May 5, 2009.]. [Online]. Available: http://forums.nvidia.com/index.php?showtopic=92290

[27] nwilt, "Page-locked memory," February 1, 2008, [Cited: November 26, 2008.]. [Online]. Available: http://forums.nvidia.com/index.php?showtopic=58505\&st=0\&p=318592\&#entry318592

[28] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[29] Øystein Torbjørnsen, "Personal Communication," FAST, a Microsoft Subsidiary, February 2009.

[30] M. Porter, "An Algorithm for Suffix Stripping," *Program*, vol. 14, no. 3, pp. 130–137, July 1980.

[31] R. F. Rice, "Some Practical Universal Noiseless Coding Techniques," Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep., 1979.

[32] K. M. Risvik and T. Egge, "The FMS Search Engine Kernel and its Performance Characteristics," Norwegian University of Science and Technology, Department of Computer and Information Science, Tech. Rep., 2002.

[33] A. S. Tanenbaum, *Structured Computer Organization*, 5th ed. Prentice Hall, 2005.

[34] A. Trotman, "Compressing Inverted Files," *Information Retrieval*, vol. 6, no. 1, pp. 5–19, January 2003.

[35] B. Wilkinson and M. Allen, *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd ed. Pearson Education, Inc., 2005.

[36] H. E. Williams and J. Zobel, "Compressing Integers for Fast File Access," *The Computer Journal*, vol. 32, no. 3, pp. 193–201, 1999.

[37] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes - Compressing and Indexing Documents and Images*, 2nd ed. Morgan Kaufmann Publishers, 1999.

[38] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," *ACM Computing Surveys*, vol. 38, no. 2, July 2006.

[39] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Super-Scalar RAM-CPU Cache Compression," in *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, Atlanta, GA, USA, April 2006.

Appendix A

Annotated Reference

A.1 GPUs and CUDA

[24, 25] The CUDA Programming Guide released by NVIDIA with the CUDA programming language extension is a valuable source of information on how to create CUDA programs which can run on the GPU. It contains short descriptions of the features of CUDA and a large example in the form of matrix multiplication. The most important contribution by the manual is the performance chapter which outlines strategies to improve the performance of a CUDA program.

[7, 28] The two articles *Rise of the Graphics Processor* and *GPU Computing* gives a soft introduction to the Tesla Architecture and the history of the GPU. They also give a valuable insight in how the Tesla Architecture both allows for graphical rendering and general-purpose computations.

[23] The article *NVIDIA Tesla: A Unified Graphics and Computing Architecture* gives a detailed view on the Tesla Architecture.

[9] The article *A performance study of general-purpose applications on graphics processors using CUDA* contains an evaluation of CUDA for several different problem types and gives a clear image of many of the challenges faced by CUDA developers when trying to map a problem so that is efficiently solved on the Tesla Architecture.

A.2 Search Engines and Compression

[38] The tutorial article *Inverted Files for Text Search Engines* gives a good introduction to the various aspects of a inverted index in a search engine. It lists many alternative ways to create, store and maintain the index, and gives a discussion of the possibilities and limitations with each approach. In the way the material is presented, the article gives the reader a clear and concise introduction to the field of inverted indexes and leaves the reader with a good basis for understanding more detailed and specific literature.

[5] For an overall view of the field of Information Retrieval, the book *Modern Information Retrieval* is recommended. It contains information on all aspects concerning a search engine, and provides the reader with a good foundation to clearly understand the inner workings such an application.

[30] The article *Compressing Integers for Fast File Access* looks into several compression schemes which can be used to compress inverted indexes.

[37] For an in depth coverage of indexing in search engines, the book *Managing Gigabytes* is a great source of information. It contains in detail description of various compression schemes used in compressing inverted indexes, and thus gives a good foundation of assess such schemes in terms of usage on a GPU.

Appendix B

Articles and Posters

B.1 Branch Performance on the Tesla Architecture

The article *Branch Performance on the Tesla Architecture*[1] was written as an assignment in the cource *TDT4260 Computer Architecture* given each spring-semester at the Norwegian University of Science and Technology. As one of the graded evaluation during this course, a report must be delivered which focuses on an aspect of one or more computer architectures. Given the interest and focus on GPU in the HPC community, an article about the Tesla Architecture was a natural choice. The Tesla Architecture has a lot of performance characteristics which are different from CPUs due to the special massively parallel architecture. One of this is a penalty when performing branches within a thread-warp, and the article investigates this. Branching was chosen as it is one of the less discussed penalties as it may not have an equally large impact on performance as other characteristics of the architecture.

[1] Available: http://wo.uio.no/as/WebObjects/frida.woa/wo/29.Profil.29.25.2.3.15.1.0.3

Branch Performance on the Tesla Architecture

Rune Johan Hovland

Abstract—The use of CUDA for GPGPU applications has been a tremendous success. Many applications and algorithms have been reimplemented to run on the Tesla Architecture. However this architecture has other performance characteristics than regular CPUs and CPU clusters. The use of a Singe-Instruction-Multiple-Thread (SIMT) architecture forces the developers to consider new pitfalls such as wrong use of branching. This paper will show how the Tesla Architecture handles branching trough defining a theoretical model, and discussing the validity of this. The performance characteristics which can be expected from a program using branches will also be showed as part of validating the model.

Index Terms—Tesla Architecture, CUDA, branching, performance.

1 Introduction

With the introduction of the Tesla Architecture and CUDA, the High Performance Computing (HPC) community has been given the tools necessary to easily do General-Purpose Computing on GPUs (GPGPU). The use of NIVIDAs GPUs has thus been given much attention, and many papers has been released outlining how to perform various algorithms and applications using CUDA. What is also pointed out by the same articles is the difference in paradigm under which one develops. As normal supercomputers are either Single-Instruction-Multiple-Data (SIMD) or Multiple-Instruction-Multiple-Data (MIMD), the change to the Single-Instruction-Multiple-Thread (SIMT) encouraged through CUDA proves to be difficult. One of many pitfalls is wrong use of branching which may lead to reduction in performance. This paper seeks to clarify the behavior of branching on the Tesla architecture, and which performance characteristics can be expected when using branching.

The paper will start with a outline of the GPU and the Tesla Architecture in Section 2, and then later in the section focus more on threading in the Tesla Architecture, and how this correlates to branching. The section then finishes with a small part about GPGPU and CUDA. In Section 3 a theoretical model for instruction execution on the Tesla Architecture is given, along with a discussion on how this affects branching. Section 4 gives an overview of the test environment used in the verification of the model. Here both the hardware and software used is described. The tests used to verify the model are given in Section 5 along with the results of the tests. The paper concludes on the validity of the model in Section 6.

• *R. J. Hovland is a graduate student with the Department of Computer and Information Science at the Norwegian University of Science and Technology, Trondheim, Norway.*
E-mail: runejoho@stud.ntnu.no

2 Background

From the first simple dedicated graphics systems in 1960 to the massively parallel computational platforms today.s graphics cards are, there has been a tremendous evolution [1]. The first systems merely acted as specialized hardware for drawing graphics on vector displays and later raster displays. As 3-dimentional drawing became more sought after, the graphics systems included hardware for transforming 3-dimentional objects into 2-dimentional drawings. By the 1980's personal computers started including specialized extension cards for displaying graphics and gave birth to the graphics card. As more and more features were added to the graphics pipeline, the cost of supporting the various graphics cards increased unacceptably, and the need for standardization emerged, and in the 1990's both OpenGL[1] and Direct 3D[2] application programming interface (API) were released. Still with the standards in place, the pipeline increased due to new requirements such as multimedia acceleration and specialized shaders. To overcome this increase in complexity, Direct 3D introduced its Unified Shader Model in 2006.

The Unified Shader Model introduced as a part of Shader Model 4.0 in the Direct 3D 10 specification [2] marked a turning point in the development of graphics processing units (GPU). By expressing all its shaders on a single shader core, it allowed for reuse of shader units for different types of shaders. The intention with this choice was to overcome the problems with the earlier pipelines where specialized shaders were not fully utilized due to mismatch between the pipelines ratio between shader types and the applications needs. By basing the shaders on the same shader core, a shader could be used in all shader steps of the pipeline and thereby allowing the GPU to adjust the pipeline to applications needs. Another effect caused by the Unified Shader Model was that the use of a single shader core throughout the pipeline made the GPU more suited for general-purpose

1. www.opengl.org
2. www.microsoft.com/windows/directx/

computation.

2.1 Tesla Architecture

The Tesla architecture [3] is NVIDIA's Unified Shader Architecture, and first appeared in the G80 series of its GPUs. The architecture is designed in such a way that it is highly scalable, allowing it to be used in a wide range for GPUs. This scalability is achieved through the use of Streaming Multiprocessors (SM). These processors can be duplicated any number of times to give the GPU its desired performance. Since each SM is an independent unit without possibility to communicate directly with other SMs, this scalability is easily achieved.

These processors form the backbone of the architecture, and give the Tesla architecture its ability to scale. To give the GPU its desired performance and parallelism, the SM can be duplicated any number of times. As an example, GeForce GTX 285[3] which is the new high-end GPU has a total of 30 SMs, while the low-end GeForce 9400GT[4] 9400GT[5] only has two SMs. An example layout of the Tesla Architecture can be viewed in Figure 1.

The Streaming Multiprocessor is a Single-Instruction-Multiple-Thread (SIMT) processor, and this is emphasized by NVIDIA [4]. While not part of Flynn's taxonomy [5], there is a key difference between SIMT and Single-Instruction-Mulitple-Data (SIMD). Both types allow the same instruction to be executed on multiple data in parallel. The key difference as pointed out by NVIDIA is that while SIMD processors exposes the data parallelism, the Tesla Architecture hides this by allowing the developers to program multiple threads and running them in parallel when their instructions are equal. This thread parallelism is achieved by the eight Streaming Processors (SP) inside the SM. These SPs all perform the same instruction which is given by the SMs *Instruction Fetch and Issue Unit* (MT Issue). If one or more of the threads does not contain the instruction, the corresponding SP is deactivated during the instruction execution and thus maintaining the correct program execution for all threads. This effect will be discussed further in Section 2.2. In addition to the SPs and MT Issue, the SM contains two Special Function Units (SFU) and a shared memory. The SFUs are specialized hardware capable of performing more complex calculations such as square root. Also include as of the NVIDIA G200 series, is a double-precision floating-point unit which can do double-precision calculation. This is required as the SPs are only capable of performing calculations using single-precision floating-point and integers.

The shared memory located on the SM is part of a two level memory hierarchy in the Tesla Architecture. Since the Tesla Architecture does not implement cache for data memory, a good utilization of the shared memory by the developers is crucial to achieve high performance. Since a memory access instruction takes four cycles, and the latency to the global memory is 4-600 cycles, there is great performance gain by prefetching data to the shared memory. Another vital thing to consider is the access pattern to the shared memory, as the shared memory is divided into 16 memory banks which can be accessed in parallel. If two or more threads try to access the same memory bank, the access is serialized. The NVIDIA CUDA Programming Guide [4] is a good source of information on this effect, and some more optimization techniques.

3. http://www.nvidia.com/object/product_geforce_gtx_285_us.html
4. http://www.nvidia.com/object/product_geforce_9400gt_us.html
5. http://www.nvidia.com/object/product_geforce_9400gt_us.html

2.2 Thread Branching on Tesla Architecture

As pointed out earlier the Streaming Multiprocessor is a SIMT processor. This enables the developers to write a massively multithreaded program, and the Tesla Architecture manages and parallelizes the threads. To organize the large number of threads, the threads are grouped together in thread blocks of up to 512 threads. On or more thread blocks are executed on a SM interleaved. To mask IO operations, a stalled block may be replaced by another block to allow high utilization of the SM. Each thread block operates as a independent unit without other possibilities to communicated with other blocks than through the global memory. Within a thread block the threads are grouped together in Warps of 32 threads. These threads are run in parallel on the SM, and all execute the same instruction. To allow for instruction decoding, the SM runs the 32 threads in four iterations on the eight SPs. In the documentation the Warp is divided into half-warps of 16 threads, but as far as it can be found this grouping refers more to memory access than execution patterns.

When a warp is executed, it must all perform the same instruction. If branching occurs among the threads, and one or more threads follows another branch, the SM executes the different branches in serial and disables the SPs handling the threads not following the current branch. An effect of this is that any branching within a warp will lower the utilization of the SPs and thus also reduce the performance.

In the code given in Figure 2.2 the if-statement will create a divergent path for the threads in the warp. The first 11 threads will execute line 5, while the remaining threads will execute line 9. To handle this situation, the SM will then execute line 1-3 as normal, and then execute line 5 with the SPs handling thread 11 to 31 disabled. Further it will then execute line 9 with SPs assigned to thread 0 to 10 disabled, and then finally execute line 11 for all SPs. Even though each thread only executes 5 instructions, the SM have to use 6 steps to complete, since the threads takes divergent paths, and this reduces the performance.

2.3 General-Purpose Computing on GPUs

The ability to perform general-purpose computing on GPUs (GPGPU) have been possible since the appearance

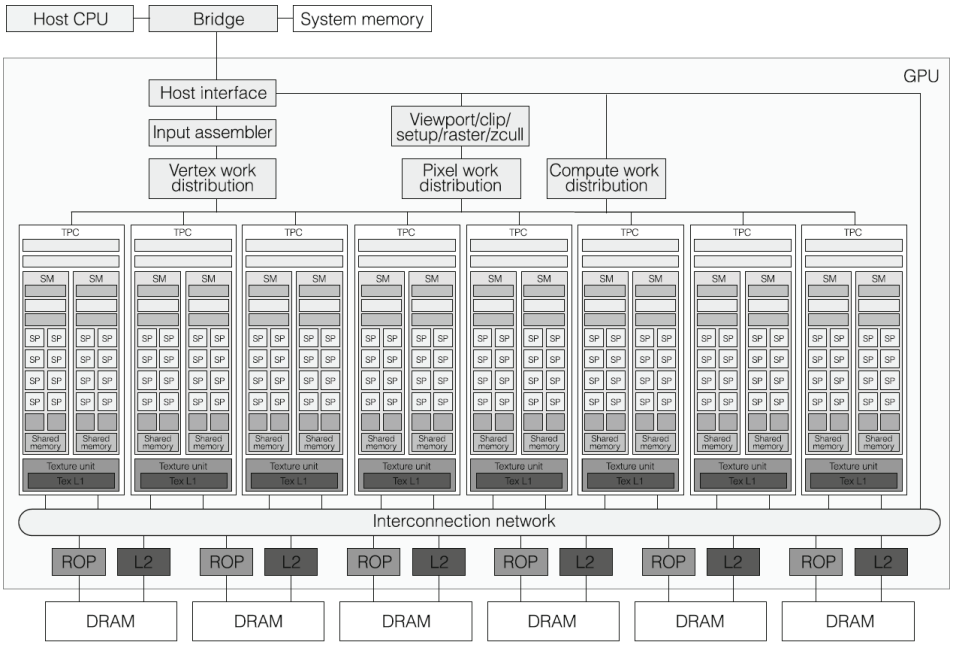


Fig. 1. The Tesla Architecture [3].

```
1  int threadID = threadIdx.x;
2  int k = 8;
3  if ( threadID < 11 )
4  {
5    k = k + 5;
6  }
7  else
8  {
9    k = k - 2;
10 }
11 k = k + 3;
```

Fig. 2. Branching code. The first 11 threads execute the if-section while the remaining 21 threads executes the else-section.

of programmable shaders. However, the task has not always been as easy as today. The pioneers of GPGPU had to camouflage their computations as graphical rendering, a necessity to adapt the computation to the graphics pipeline [6]. This transformation between a given problem and a graphical rendering was not a trivial task and set the bar to high for common usage. Many attempts have been made to hide this transform from the user through languages such as Brook and Sh.

Parallel to the introduction of the Tesla Architecture, NVIDIA released CUDA which is an extension to the C programming language. It allows the developer to program directly towards the GPU without having to consider the graphics pipeline. The extension includes the method modifier `__global__` which makes the method execute on the GPU. These kind of methods are called kernels. When calling the kernel, the program specifies how many thread blocks and threads to spawn like this; `foobar<<<number of blocks, number of threads per block >>>(arguments).` Any data required for the calculations must explicitly be copied to the GPU, and the result copied back. The graphics card and surrounding system may both affect the performance of this operation [7], and thus affect the performance of several bandwidth-bound GPGPU-applications.

For more information regarding CUDA, see [4]

3 Methodology and Models

Based on the description of how the Tesla Architecture operates in [3][4], there has be created a simple model describing the expected behavior of the system. The model describes the two steps needed by the Tesla Architecture to execute a single instruction. The MT

Issue unit fetches and decodes the instruction which is to be executed in the first step. It also determines which SPs are going to be active during the execution. The deactivation of SPs will ensure that threads do not execute unwanted instructions. After this step is completed, the second step commences. Here the MT Issue unit oversees the execution of the instruction. This step is divided into four sub-steps where eight threads at the time are executing the instruction. At the beginning of each step, the MT Issue unit activates the SPs which is assigned threads who are to execute the instruction. The remaining SPs are deactivated. Both steps take the same amount of time, allowing them to be pipelined, giving a higher throughput. The outline of the model can be seen below.

1) Instruction decoding stage
2) Exectuion stage
    a Execute thread 0-7
    b Execute thread 8-15
    c Execute thread 16-23
    d Execute thread 24-31

When executing a branch under these conditions, the SM would see to instructions needed to be executed, and would then serialize it since the SPs are only capable of performing the same instruction. A branch with two paths would therefore take the total execution time as if the two paths were executed after one another, which is exactly what is done.

4 Test Environment

The hardware used for these tests are a common personal computer with high-end components. The hardware can be seen in Table 4. It has been configured with the 64 bit version of Ubuntu 'Hardy Heron' 8.04, and the NVIDIA driver is of version 180.22.

TABLE 1
Test hardware

Processor		
	Intel Core 2 Quad Q9550, 64 bit	
	Clock frequency	2.83 GHz
	L2 Cache	12 MB
	Bus speed	1333 MHz
Motherboard		
	EVGA nForce 790i Ultra SLI	
	Chipset	nForce 790i Ultra SLI
Memory		
	OCZ DDR3 4 GB Platinum EB XTC Dual Channel	
	Frequency	1600 MHz
	Size	2x 2048 MB
GPU		
	NVIDIA GeForce GTX 280	
	Processor Cores	240
	Graphics Clock	602 MHz
	Processor Clock	1296 MHz
	Memory Clock	1107 MHz
	Memory Size	1 GB
	Memory Bandwidth	141.7 GB/sec

The software is a simple test program which enables the user to run different types of branches multiple times and count the number of cycles. The program consists of two parts; a CUDA kernel which runs a for loop 10 000 times, and within that loop does the wanted branching. Before and after the loop, there is functionality to start and stop the cycle counter. The other part of the program is the CPU application, which starts the CUDA kernel, and supplies it with dummy data used in the kernel.

5 Results

To exactly determine the behavior of the Tesla Architecture and its conformance with the model given in Section 3 would detailed descriptions of the architecture and all its optimizations. This is however not accessible, so another approach has to be taken. By devising small tests to expose performance details about the Tesla Architecture, it can be showed beyond reasonable doubt the correctness of the model. The four tests performed are given in the following subsections.

5.1 Number of Branches

It is pointed out by NVIDIA in their CUDA Programming Guide [4] that using branches within a warp can seriously affect the performance. This is due to the SIMT architecture, which only allows one instruction at the time to be performed by the warp. Any divergent paths must be handled in serial. The more divergent branches, the longer it would require to complete all branches. Based on this description, one would expect a linear increase in cycles needed to complete an increasing number of branches. This effect can be seen in Figure 3 where the test program create a number of divergent branches. This is done using a switch with thread id as argument. Threads who do not diverge are handled by the default-clause, ensuring equal computational load on all threads. What is worth noticing is the abnormality with one divergent thread, where the additional thread does not cause the expected increase in cycles. This may be an optimization made by NVIDIA to allow one branch to act as a control branch without the full branch penalty.

5.2 Location of Branches

Since the test in Section 5.1 uses thread zero to branch out one thread, an extra test is required to test if the reduced cost of branching for a single branch is thread location dependent. To test this, a divergent branch will be created which only one thread will follow. Then this branching scheme is tested for all 32 threads. As can be seen in Figure 4, there is no difference in the cost of branching out a single thread regardless of which thread follows the branch.

5.3 Grouping of Branches

The SMs are composed of eight SPs, while the warp is divided into half-warps of 16 threads. To determine if

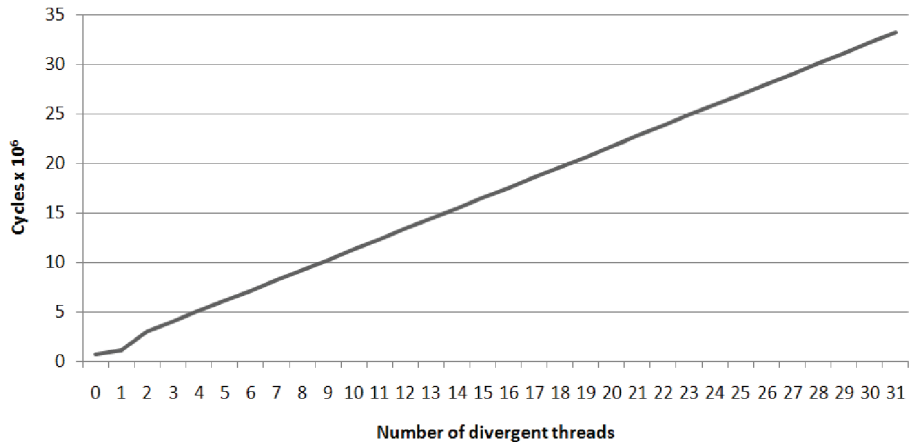


Fig. 3. Total cycles needed by testprogram for increasing number of divergent branches.

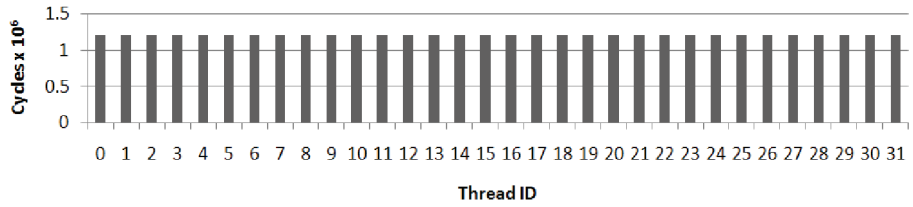


Fig. 4. Total cycles neaded by testprogram with a single divergent thread.

these groupings may have an effect on the performance of branching, two simple tests have been devised. The first test determines if threads can diverge as long as all threads executed simultaneously on the SPs does not diverge. This is done by creating four branches, and running groups of eight threads through the four branches. The second test is created in the same manner but with two branches and threads grouped 16 together. The result of these runs can be seen compared with the runs of the threads scrambled across the branches in Figure 5. As can be seen there is no difference in the required number of cycles.

5.4 Size of Branch

It has been showed that the location of a branch does not have an impact on the performance, but what remains to be showed is the impact of the number of threads following a branch. To show this a branch is created and a increasing number of threads are instructed to follow this branch. As can be seen in Figure 6, there is no difference in the required number of cycles for the different number of threads following the branch.

6 Conclusion

Through this paper, there has been showed a theoretical model which describes the execution of instructions on the Tesla Architecture. This model has then been examined and attempted verified by four tests designed to expose the performance characteristics of the architecture. During these tests the model was for most parts verified, with one exception. The test which should show how the required number of cycles needed to perform an increasing number of branches did not appear to comply completely with the model. When there is only one diverging branch, the performance does not decrease to the expected level. This may indicate that the Tesla architecture has some built-in optimization to handle one diverging branch. A reason for this may be to increase the performance for applications where warps have master-threads which execute different code

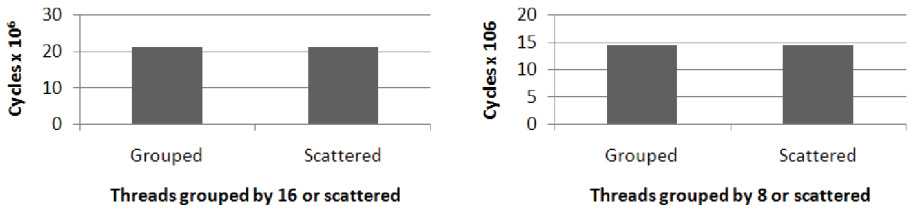


Fig. 5. Total cycles neaded by testprogram with a divergent paths grouped or scrabled across threads.

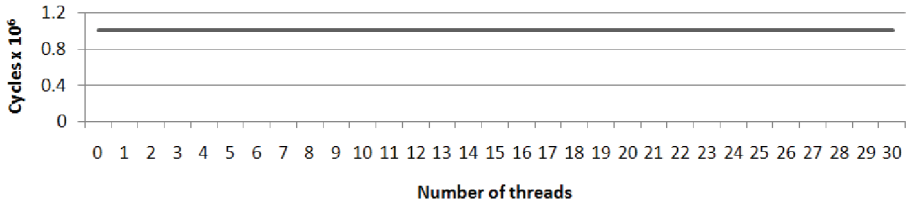


Fig. 6. Total cycles neaded by testprogram with an increasing number of threads on divergent path.

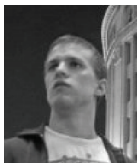
to control the other threads. Besides this one case, the model describes the instruction execution on the Tesla Architecture well. Although not the main reason for low performance on CUDA applications, this paper has shown which impact poor use of branching could cause on the performance, and that it is a aspect developers must pay attention to.

ACKNOWLEDGMENTS

The author would like to thank the HPC-group at the Department of Computer and Information Science at the Norwegian University of Science and Technology for use of the HPC-lab. He would also like to thank the NVIDIA Corporation for donating the graphics cards used in this paper.

REFERENCES

[1] D. Blythe, "Rise of the Graphics Processor," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 761–777, May 2008.
[2] ——, "The Direct3D 10 System," *Proceedings of the ACM SIGGRAPH 2006*, vol. 25, no. 3, pp. 724–734, July 2006.
[3] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March-April 2008.
[4] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide," [Cited: January 17, 2009]. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf
[5] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 52, no. 12, pp. 1901–1909, December 1966.
[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[7] R. J. Hovland, "Latency and Bandwidth Impact on GPU-systems," December 2008, report in course "TDT4590 Complex Computer Systems, Specialization Project", Department of Computer and Information Science, Norwegian University of Science and Technology. [Cited: February 2, 2009]. [Online]. Available: http://publications.runejoho.net/gpgpu_latency_bandwidth.pdf



Rune Johan Hovland pursuits his Master of Technology at the Norwegian University of Science and Technology. The focus of the master is High Performance Computing, and the master-thesis focuses on using GPUs to accelerate search methods.

B.2 NOTUR 2009 Poster

NOTUR[2] is an organization responsible for the national infrastructure for computational science in Norway. Each year they host the conference *High Performance Computing and Infrastructure in Norway* on different universities in Norway. In May 2009, this conference was hosted in Trondheim at the Norwegian University of Science and Technology, and this thesis was presented as a poster[3]. While not all aspects covered in this thesis could be included, the poster focused on the expansion of the memory hierarchy, and presented the theoretical model with the calculated performance improvements. Also included was a short summary of the case study and the other suggested improvements.

[2]http://www.notur.no/

[3]Available: http://wo.uio.no/as/WebObjects/frida.woa/wo/32.Profil.29.25.2.3.15.1.1.3

High Data Volumes and Streaming on Future GPU Systems

Rune Johan Hovland
Master Student, IDI, NTNU

Anne C. Elster
Advisor, IDI, NTNU

Magnus Lie Hetland
Co-advisor, IDI, NTNU

Introduction

The HPC community has devoted a great deal of attention to the general purpose capabilities of the Graphics Processing Unit. More recently, the potential of the GPU as a computational device has also received attention in the Information Retrieval community. Large data volumes and a focus on throughput are characteristic for applications within this area. A search engine must handle large amounts of data, and most of this is fetched from disk, making data access a substantial cost for these applications. The lack of streaming capabilities between host and device for the GPUs may limit the potential benefits of using GPUs in such applications.

To allow streaming capabilities to the GPU, this poster proposes to allow the GPU to access host-memory during execution. This will allow data transfers and computations to be interleaved by pipelining. It has been developed a theoretical model for the existing Tesla Architecture, and a model for the suggested improvements.

Theoretical Model

For variables, b denotes bandwidth, s is datasize and l is latency. The bandwidth between host and device $b_{m\leftrightarrow gm}$ is assumed to be the lowest, and the bandwidth increases for transfers closer to the GPU processors.

$$b_{m\leftrightarrow gm} < b_{gm\leftrightarrow r} < b_{r\leftrightarrow gpu}$$

Bandwidths and latencies are assumed to be symmetric.

Current

The current NVIDIA CUDA version requires that data is copied to the GPU before it can be used by the kernel and the results must be copied back to the host after the computation is complete.

$$t_{current} = \underbrace{\frac{s_d}{b_{m\leftrightarrow gm}}}_{t_{copy\ to\ device}} + \underbrace{\frac{s_d + s_r}{\min(b_{gm\leftrightarrow r}, b_{gpu})}}_{t_{computation}} + \underbrace{\frac{s_r}{b_{m\leftrightarrow gm}}}_{t_{copy\ from\ device}} + l_{current}$$

Improved

By adding streaming capabilities between host and device, the time is dependent on the data size and the lowest bandwidth in the stream.

$$t_{new} = \frac{s_d + s_r}{\min(b_{m\leftrightarrow r}, b_{gpu})} + l_{new}$$

Performance Improvements

There is two outcomes on performance improvements based on which bandwidth that limits the application.

Data Bound

$$t_{benefit} = \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{\min(b_{gm\leftrightarrow r}, b_{gpu})} - \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + l_{current} - l_{new} = \frac{s_d + s_r}{\min(b_{gm\leftrightarrow r}, b_{gpu})} + l_{current} - l_{new}$$

The cost of computations is hidden in this case.

Computational Bound

$$t_{benefit} = \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + \frac{s_d + s_r}{b_{gpu}} - \frac{s_d + s_r}{b_{gpu}} + l_{current} - l_{new} = \frac{s_d + s_r}{b_{m\leftrightarrow gm}} + l_{current} - l_{new}$$

The cost of data transfers is hidden in this case.

Case Studies

The cases are based on the article "Using Graphics Processors for High Performance IR Query Processing" by Ding, He, Tan and Suel at NYU which was published at the 18th Int. World Wide Web Conference. Benchmarks are taken from the article and used to show performance improvement for the suggested architectural changes.

Compression

This case uses the GPU to decompress the search index. The compressed data is assumed to be in host-memory, and the uncompressed data is copied back to host-memory.

	Rice coding	PForDelta coding
Current GPU vs CPU	0.639	0.333
Sugested GPU vs CPU	0.983	1.062
Suggested GPU vs current GPU	1.537	3.191

Search Engine

This case uses the GPU to run a complete query engine. There is no caching of data on the GPU. Data is assumed to be available in host-memory

	AND	OR	AND+OR
Current GPU vs CPU	1.055	5.504	1.648
Sugested GPU vs CPU	1.763	7.647	3.090
Suggested GPU vs current GPU	1.671	1.389	1.874

Cached Search Engine

If caching is used on the GPU, the speedups would be affected as the amount of data needed to be transferred for each query is reduced. This is shown in Figure 1

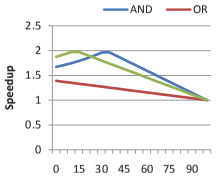


Figure 1: Speedup for Suggested GPU Architecture

Figure 2: Computational and data transfer time

In Figure 2, the time each query operation requires is shown together with the time used for data transfers. Observe how the intersection between data transfer time and computational time relates to the speedup.

Conclusion

The suggested architectural improvements will reduce the overall cost of performing calculations on the GPU by pipelining operations. This will cause the cost of either the data transfer or computations to be hidden, giving a increased performance. Currently NVIDIA is developing some of this functionality in CUDA 2.2, which is still in the Beta stage.

Improvements

Memory Hierarchy

A limitation with current GPUs is the lack of possibility to access host-memory while the kernel is running. Any data that is needed for the computation must be present on the GPU before the computation can commence. By allowing the kernel to access host-memory, it can perform calculations while prefetching data needed later in the computation. This will reduce the total time needed for the computation due to efficient pipelining of data copy and computation.

Automated Caching

Currently the GPUs memory hierarchy is not cached for normal data. Data locality is a key factor in performance, and handling this then becomes an important task. Manual handling of locality can lead to improved performance, but also complicates development. Automated caching should therefore be introduced as an optional feature when allocating memory to reduce the complexity for novice developers. Caching is already implemented for texture data, and normal data can be masked as textures, so this can easily be solved using syntactic sugar.

Extended Host-Device Synchronization

Allowing synchronization between host and GPU would allow complex program to be less complex. Currently synchronization is solved by ending the kernel, thus returning control to the CPU.

Allow File Access

Many high data volume applications has extensive use of files. Porting these applications can be cumbersome when CUDA does not support file access. Allowing file access from the GPU can be difficult, but by allowing the GPU access to host-memory, Memory-Mapped-Files can be used to grant the GPU file access without physical access.

Acknowledgements

We would like to thank NVIDIA for providing several of the graphics cards used in this project through Dr. Elsters membership in their Professor Affiliates Program.

NTNU
Norwegian University of Science and Technology

Appendix C

Test Systems

The systems used in this thesis are referred to by their graphics card. In this appendix, the rest of the specifications for the three systems are given.

Table C.1: GeForce GTX 280 machine specifications

Processor		
	Intel Core 2 Quad Q9550, 64 bit	
	Clock frequency	2.83 GHz
	L2 Cache	12 MB
	Bus speed	1333 MHz
Motherboard		
	EVGA nForce 790i Ultra SLI	
	Chipset	NVIDIA nForce 790i Ultra SLI
Memory		
	OCZ DDR3 4 GB Platinum EB XTC Dual Channel	
	Frequency	1600 MHz
	Size	2x 2048 MB
Hard-disk		
	Samsung Spinpoint S166	
	Rotational Speed	7200 RPM
	Size	160 GB

Table C.2: GeForce 9300m machine specifications

Processor		
	Intel Pentium Dual Core E5200, 64 bit	
	Clock frequency	2.50 GHz
	L2 Cache	2 MB
	Bus speed	800 MHz
Motherboard		
	Asus S-775 Gf7300	
	Chipset	NVIDIA GeForce 9300/nForce 730i
Memory		
	CORSAIR TWIN2X 6400 DDR2	
	Frequency	800 MHz
	Size	2x 2048 MB
Hard-disk		
	Western Digital Caviar GP	
	Rotational Speed	7200 RPM
	Size	1 TB

Table C.3: ION machine specifications

Processor		
	Intel Atom 330, 64 bit	
	Clock frequency	1.60 GHz
	L2 Cache	1 MB
	Bus speed	533 MHz
Motherboard		
	Zotac ION	
	Chipset	NVIDIA ION
Memory		
	OCZ DDR3 4 GB Platinum EB XTC Dual Channel	
	Frequency	1600 MHz
	Size	2x 2048 MB
Hard-disk		
	Western Digital Caviar GP	
	Rotational Speed	7200 RPM
	Size	1 TB