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

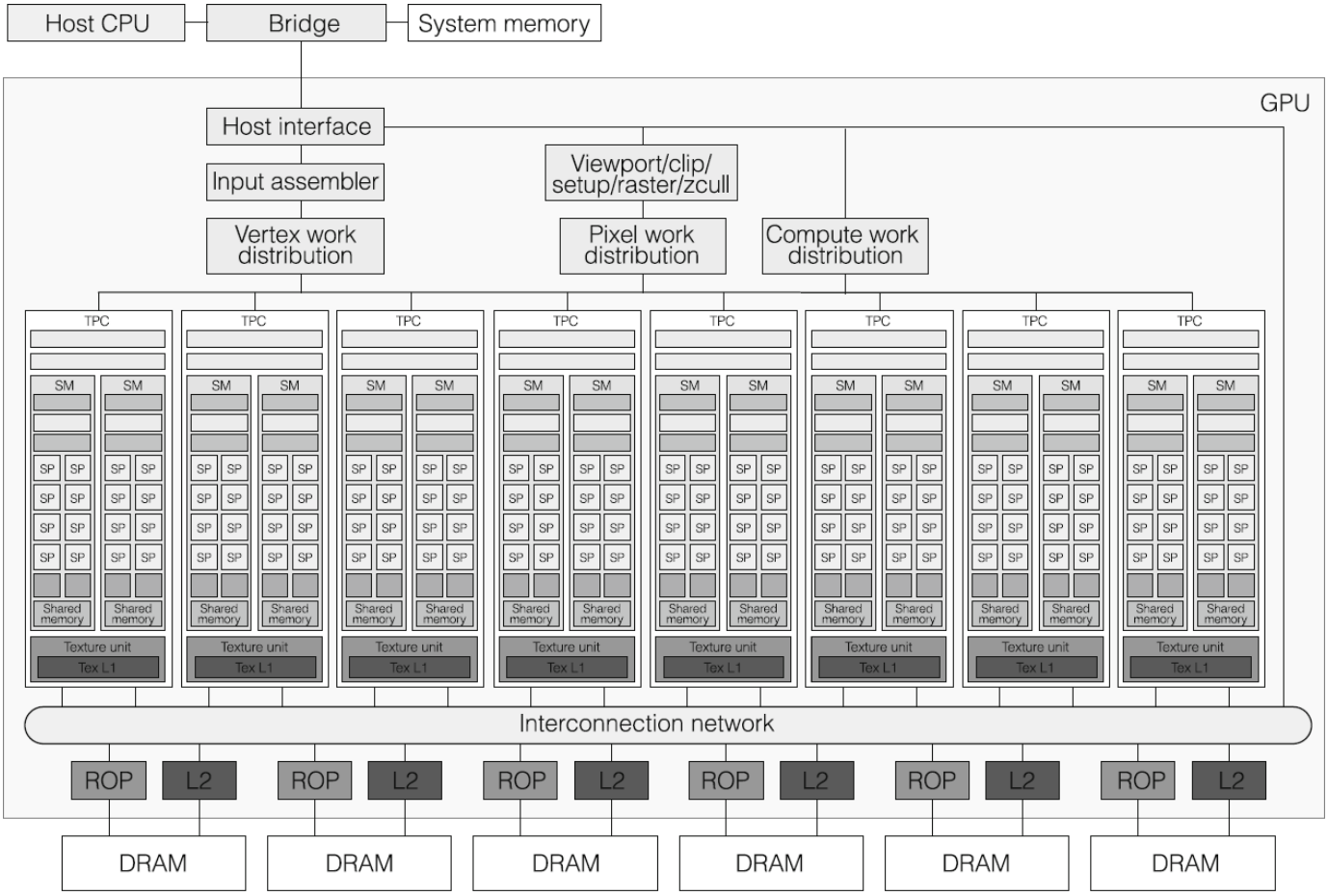

Fig. 1. The Tesla Architecture [3].

```
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

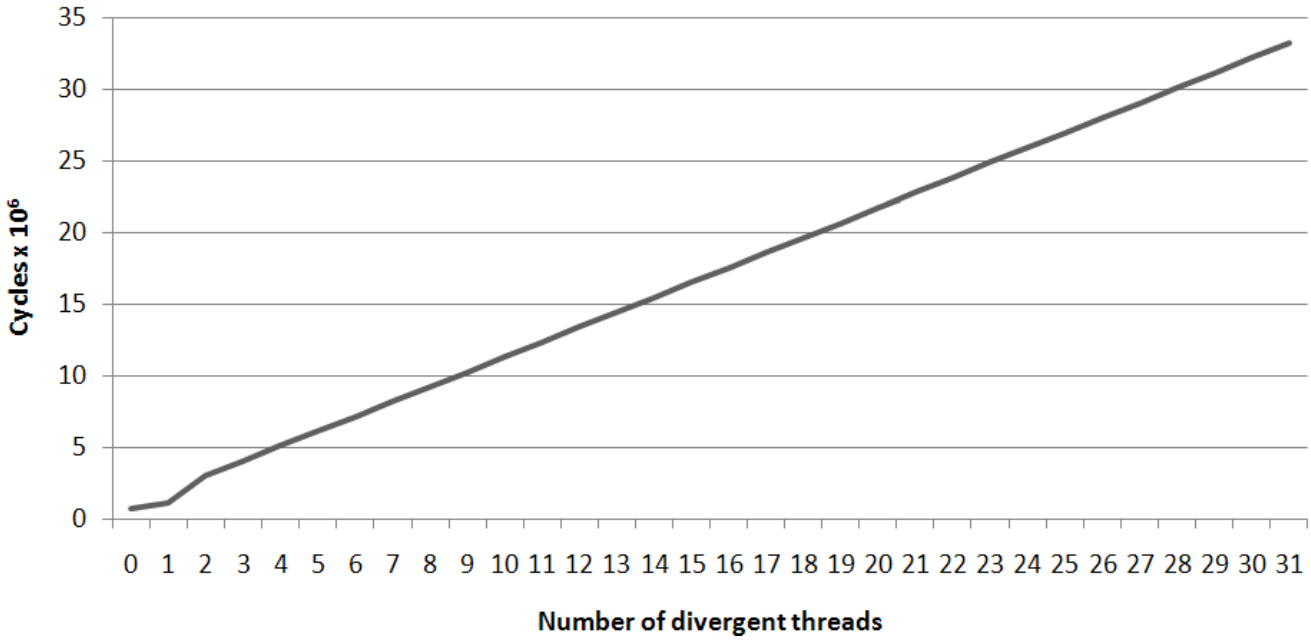

Fig. 3. Total cycles needed by testprogram for increasing number of divergent branches.

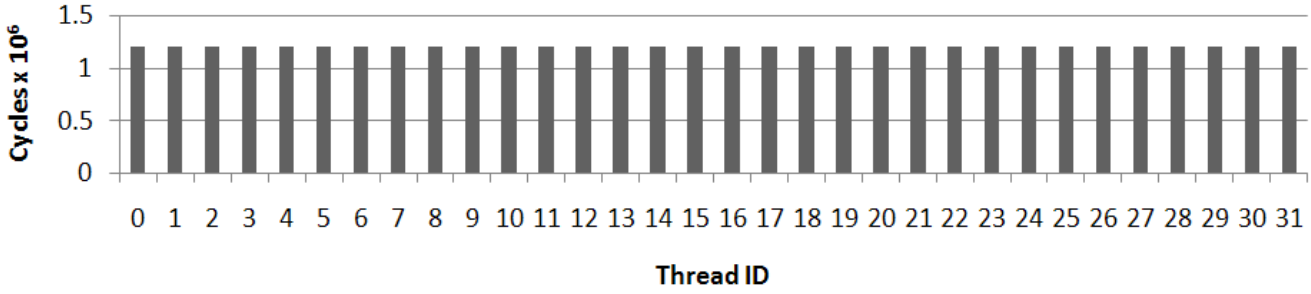

Fig. 4. Total cycles neaded by testprogram with a single divergent thread.

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

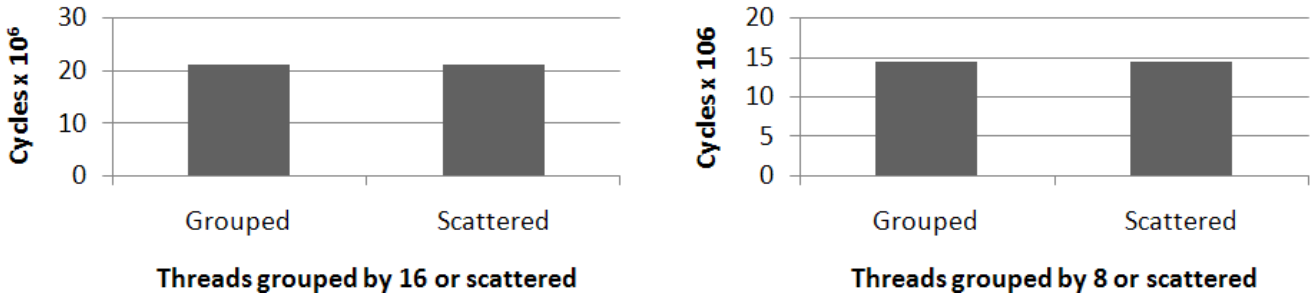

Fig. 5. Total cycles neaded by testprogram with a divergent paths grouped or scrabled across threads.

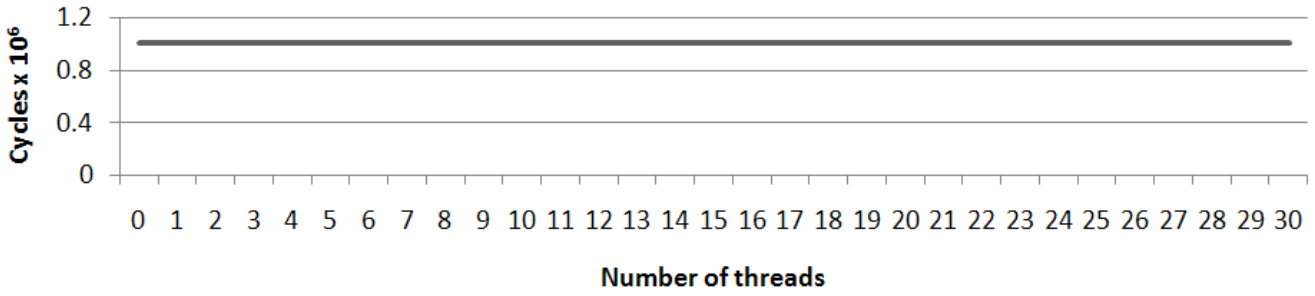

Fig. 6. Total cycles neaded by testprogram with an increasing number of threads on divergent path.

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.