Accelerating the MMD algorithm using Multi-core Environments

Michael Schlösser
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
Email: michael.schloesser@unb.ca

Rainer Herpers
Dept. of Computer Science
Hochschule Bonn-Rhein-Sieg
Sankt Augustin, Germany
Email: rainer.herpers@h-brs.de

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, NB, Canada
Email: ken@unb.ca

***Abstract*—This paper presents two approaches to accelerate the MMD algorithm in multi-core environments. The MMD algorithm is a transformation-based algorithm based in the field of reversible logic synthesis. It is used to synthesize and optimize reversible circuits which are an integral part of future technologies like quantum computers. However, the MMD algorithm is computationally intensive and the acceleration of the algorithm might not only produce faster but also better results. This paper focuses on two parallel hardware environments, the Cell Broadband Engine and the NVIDIA Tesla architecture. In the course of this project two different parallel algorithmic approaches have been implemented on both hardware architectures. These implementations have been compared in order to find the best combination of algorithmic approach and matching architecture. Additionally, the answer to the question if parallel hardware architectures are a means to improve algorithms in the field of reversible logic synthesis has been examined.**

***Keywords*-Parallel Processing; NVIDIA Tesla; Cell/B.E.; Reversible Logic Synthesis; Algorithms**

I. Introduction

Reversible Logic Synthesis is an emerging research topic. Application areas like low-power CMOS design benefit from it and for some areas, like quantum computing, it is absolutely necessary. Compared to conventional Logic Synthesis, design and synthesis methods for reversible functions are still in the beginning.

The MMD algorithm, named after its originators Miller, Maslov and Dueck, is one approach to solve the synthesis problem. It also attempts to further optimize the synthesized reversible network. Tests with the original implementation have shown that one of the computationally intensive optimization steps, the template matching approach, is very limited due to the time it takes to process especially large networks.

By utilizing modern parallel hardware architectures, like multi- or many core systems, it is possible to reduce the time it takes to process and optimize input networks. This results in a faster design and testing workflow, as well as the ability to process larger networks for possible real world applications.

II. Background

A. Reversible Logic Synthesis

The MMD algorithm, which will be introduced in the next section, is based in the field of reversible logic synthesis. This area of research, like classical logic synthesis, focusses on synthesizing logical networks with the characteristic that these networks fulfill the properties of reversible logic.

Reversible Logic Synthesis has its root and application in several fields like quantum computing, low-power CMOS, nanotechnology and optical computing [1]. The idea of reversible logic synthesis is based on a couple of simple assumptions and properties. While modern circuits become smaller and faster according to Moore's Law [2], one big problem, heat dissipation, has not been solved. Among other problems, this very issue is another motivation behind reversible logic synthesis. When looking at an irreversible operation, it becomes clear that it comes at a cost in terms of dissipated heat. This is due to the fact that during computation information is lost. This lost information is dissipated as heat which has been shown by [3], [4].

The most common logical operations, like AND, OR and XOR, are irreversible and therefore cannot be used for reversible logic synthesis. When looking at the respective function definitions this becomes clear. All of these operations are defined as a function $f : (x, y) \rightarrow z$, with $x, y, z \in \{0, 1\}$. This means two bits of information are provided but the result is only one bit of information. This single bit that is lost during computation produces the aforementioned heat.

In order for a function $f(x_0, x_1, x_2 \ldots, x_n)$, to avoid the aforementioned problem and to be reversible, two fundamental properties must be satisfied. First, the number of inputs and outputs must be equal. Secondly, a reversible function is a bijection. In other words, every output pattern has a unique preimage and therefore is a permutation of the input [5].

Many irreversible functions can be transformed to a reversible function by altering its definition in such a way that the aforementioned properties hold. Examples for this process can be found in [5], [6], [7].

978-1-4577-0253-2/11/$26.00 ©2011 Crown

B. The MMD Algorithm

The MMD algorithm, first introduced in [8], [7], is a transformation based algorithm that synthesizes a formal function specification, defined by a truth table, into a reversible network by using Toffoli gates of various sizes. Additionally, it is able to optimize the generated circuit C by applying a template matching approach that intends to reduce the number of gates of C.

The basic synthesis algoritm is a greedy, naïve approach to identify Toffoli gates on the output side of the specification [7]. In order to achieve this, the Toffoli gates are chosen to progressively transform the output part of the function specification into the input part. Consider the reversible function as a mapping over $\{0, 1, ..., 2^n - 1\}$. The notation $f(i) = j$, describes the mapping of an input vector i to an output, where i and j are the binary expansion in the range of $0 \leq i, j \leq 2^n - 1$. In a sequential step the basic algorithm now tries to transform the output mapping by applying Toffoli gates in such a way that after the successful termination $f'(i) = i, \forall 0 \leq i \leq 2^n - 1$ holds. A correctness analysis of this approach is shown in [1] and [7]. Since this is a greedy approach, it results in a big network (circuit) with less than or equal to $(n - 1)2^n + 1$. This number of gates can be reduced by applying further approaches, like *Control Input Reduction*, *Bidirectional application* and *Asymptotically Optimally Modification*, also shown in [1] and [7].

To further reduce the number of gates in a network, in the second phase template matching is proposed. It is based upon the facts that reversible networks are structured in cascades and that adding a reversible gate to a reversible network will again yield a reversible network. The consequence of this is that adding, removing or replacing gates does not affect the reversible property of a circuit.

The template matching method considers a sequence of gates that realize a certain mapping of values. If it is possible to find another sequence with less gates that realizes the same function, the first can be replaced by the second. Let A be a reversible network with n gates realizing a function f and B a second network with m gates realizing a function g. Then A can be replaced by B, if $m \leq n$ and $f = g$.

Consider A to be the input circuit an B to be a template. By shifting B across A it is possible to reduce the number of gates of A and therefore to reduce its costs. The MMD algorithm works with a whole set of templates that are applied iteratively until no further template matches are found.

C. The Cell Broadband Engine

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor, jointly developed by Sony, Toshiba and IBM. It is an implementation of the Cell Broadband Engine Architecture (CBEA), a formal specification of the underlying hardware [9]. The processor has specifically been designed in order to solve computationally intensive tasks. To achieve this goal, it utilizes several CPUs to solve the problems in parallel. In contrast to a homogeneous multi-core CPU, the Cell/B.E. consists of two different types of processors. The first is based upon the 64-bit PowerPC Architecture and is called the PowerPC Processing Element (PPE) [10], [11]. The second type of processor is the Synergistic Processor Element (SPE) which is a highly specialized and lightweight CPU, compared to the PPE [9]. These two different types of processors are interconnected with the Element Interconnection Bus (EIB), a fast and coherent Direct Memory Access (DMA) based ring structure. Figure 1 illustrates the basic layout of a Cell Broadband Engine Architecture compliant processor.

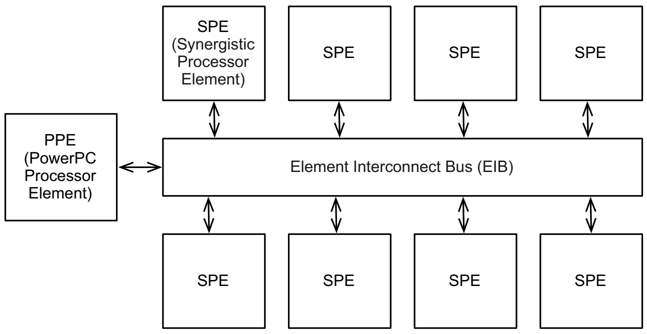
SPE (Synergistic Processor Element)
SPE
SPE
SPE
PPE (PowerPC Processor Element)
Element Interconnect Bus (EIB)
SPE
SPE
SPE
SPE

Figure 1: The Cell/B.E. consists of one PPE and eight SPUs that are interconnected with the Element Interconnect Bus (EIB).

In the CBEA design, the PPE serves as a general purpose processor that has been designed to execute control-intensive tasks such as running an operating system or to control application logic. In contrast, The SPE, has been designed to perform computationally-intensive tasks. This design creates a mutual dependency between the two types of CPU. On the one hand the PPE is dependent on the SPE in order to achieve high parallelism and to leverage high computational power. On the other hand, the SPEs are not capable of performing control-intensive tasks. They are dependent on the PPE that performs these tasks for the SPEs and delegates work to them.

The Cell/B.E. used for this work works in a Playstation 3, a next-generation gaming console designed and distributed by Sony [12]. This implementation of the CPU consists of one PPE and six freely programmable SPEs [13].

D. NVIDIA Tesla

Tesla is an architecture for Graphics Processing Units (GPU) designed by NVIDIA. One design goal was to target the usage of GPUs as a co-processor for scientific purposes like scientific simulations and calculations. Availability for computer graphics, and especially in the gaming industry for several years, latest developments in the field of so called General Purpose Graphics Processor Unit Programming (GPGPU) have made GPUs an interesting research area [14], [15], [16].

In order to provide a sufficient amount of processing power, the NVIDA Tesla GPU is divided into several compute units called Streaming Multiprocessors (SM). The number of these units can vary depending on the specific graphics card model. Generally, one can say that the more SMs on the GPU, the higher the possible parallelism. Each Streaming Multiprocessor consists of several parts from which the most important ones are the Streaming Processors (SP), and the local memory which is shared among the SM's. Independent of the graphics card model, each SM consists of exactly eight SPs. A shared memory between the SPs can store 16KB and delivers fast access to the data stored in it. Figure 2 illustrates the architecture.

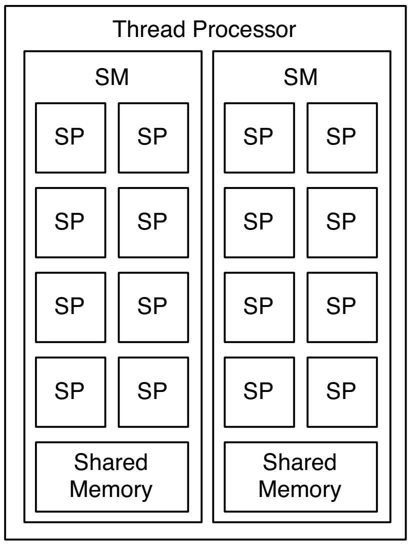

Figure 2: Structure of a Tesla Thread Processor. It consists of two Streaming Multiprocessors with each containing a Shared Memory (compare [17]).

Since the shared memory of each Streaming Multiprocessor is very limited, the Tesla architecture offers several layers of memory in order to store data. This is especially important for large data sets. Comparable to a memory pyramid, Tesla offers storage for small pieces of information in registers and a local thread memory. Inter-thread communication can be handled with the already mentioned shared memory. Larger data goes into constant or global memory, where constant memory is part of global memory but delivers a cached read-only access.

III. Parallel Concepts

A. Introduction

The sequential version of the MMD algorithm shows a high degree of locality which means that it performs its work only within a small area of the reversible circuit. In practice this means that the algorithm starts its template matching approach at the beginning of the circuit and sequentially shifts the templates from left to right. This implies that most parts of the circuit remain untouched. The parallel approaches described in this work aim to distribute the work over the whole circuit in order to have less unprocessed areas. By processing the templates at more than one position at the same time the aim is to reduce the execution time significantly.

Two different parallel approaches have been implemented on two different hardware architectures.

B. Approach One

In order to solve the aforementioned locality problem, this parallel approach accelerates the algorithm by partitioning the circuit C into t parts, minimizing these parts individually in parallel and finally merging the eventually minimized parts back together. Figure 3 illustrates this approach.

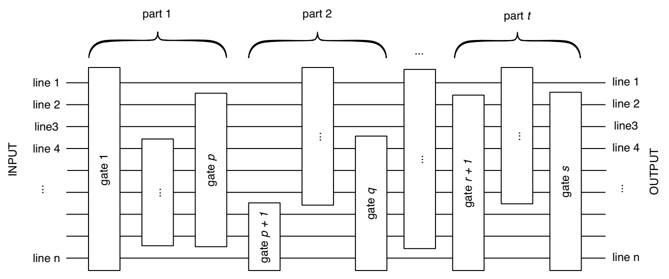

Figure 3: In order to process an input network in parallel it is partitioned into t parts which are processed independently. After successful termination the parts are merged to a single circuit.

It is, of course, important that the processed circuit still computes the same function as the original input circuit. Let $G_1G_2G_3...G_s$ be a reversible network with s gates, implementing a function f. This network can be partitioned into t parts, as illustrated in Figure 3. Every part t_i computes a function f_i with $1 \leq i \leq t$.

$$f_1 = G_1G_2 \cdots G_p \quad (1)$$
$$f_2 = G_{p+1}G_{p+2} \cdots G_q \quad (2)$$
$$\vdots \quad (3)$$
$$f_t = G_{r+1}G_{r+2} \cdots G_s \quad (4)$$

Apparently, single parts of a reversible network can be merged and still be reversible. This means the concatenation of the single function results in the total function

$$f = f_1 \circ f_2 \circ f_3 \circ \cdots f_t \tag{5}$$

Each part i now computes a function f'_i with $1 \leq i \leq t$. Now, the following observation can be made after the successful termination of the template matching algorithm.

$$f_i = f'_i \quad \forall\, 1 \leq i \leq t \tag{6}$$

f' is the function that is computed after successful termination with

$$f' = f'_1 \circ f'_2 \circ f'_3 \circ \cdots f'_t \tag{7}$$

Equation 5 and 6 can now be concatenated. This leads to the following conclusion which shows that partitioning the circuit and processing the parts individually holds.

$$\begin{aligned} f &= f_1 \circ f_2 \circ f_3 \circ \cdots f_t && (8)\\ &= f'_1 \circ f'_2 \circ f'_3 \circ \cdots f'_t && (9)\\ &= f' && (10) \end{aligned}$$

For the implementation, a one-to-one mapping between the SPEs and the partitions of the circuit have been chosen. This mapping resulted in a partition with six parts. In a preprocessing step, the boundaries and the size of each part are determined on the PPE. The resulting chunks are distributed to the SPEs which perform the actual processing work. Each single result is sent back to the PPE and is merged back together.

The first version of this implementation already showed improved runtime results but lacked the precision of the original implementation. This loss of precision can be explained by the fact that potential template matches across the border of two parts will not be recognized. In order to solve this problem, the PPE initiates a second run. In order to save execution time, the area that is processed is represented by a small window that lies centered on top of every former partition boundary. This way any previously missed match can be discovered.

In a second attempt to improve the execution time, some parts of the SPE code have been vectorized. However, since the main algorithm has many dependencies and is very dynamic, vectorizing the whole algorithm is not possible.

C. Approach Two

The basic idea of this parallel approach is to utilize the NVIDIA Tesla many-core architecture by mapping each gate of the input circuit to a GPU thread. If a circuit C has n gates, this will result in n threads running in parallel on the GPU. The idea has been illustrated in Figure 4. The rectangles illustrated represent not only the threads but also templates that start matching at the encircled areas. This leads to n overlapping template matches running concurrently and producing a result set of size n. The main difference between the sequential and the aforementioned parallel approach is that the templates do not shift across the whole circuit. Instead each thread is responsible for matching gates in a fixed boundary, staring at an initial position. This leads to a result set of size n. Each item of this result set stores a position at which the thread has been executed if the template match is successful. While the first step, running on the GPU, only checks if a template can be replaced at a given position at all, the next step analyses the result set and performs the actual replacement on the host side.

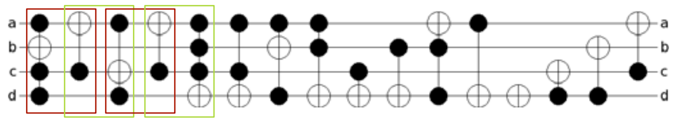

Figure 4: The parallel approach attempts to execute a thread on every single gate.

In a first version only one positive template match has been processed that leads to heavy data transfers between the host and GPU. In order to reduce these transfers an enhancement has been implemented that tries to replace as many templates on the CPU as possible. One major problem that had to be solved was that the positive matches in the result buffer can overlap. Imagine positive matches with a processing window of size n at positions p and q with $|p - q| < n$. This means it is possible that the results stored for p and q share a set of gates that potentially can be replaced which obviously is not allowed. This means that before a gate can be replaced the surrounding area has to be analyzed for potential overlaps. After this step the circuit is transferred back to the GPU and the process starts from the beginning.

IV. Cross Implementations

In order to examine behaviors and properties of both, parallel implementations and parallel hardware architectures, two additional cross-implementations have been realized. This means that approach one has been ported to the CUDA architecture and, in turn, approach two has been implemented on the Cell/B.E.. Another intent of this step was to examine and verify which kind of hardware architecture, and implicitly which kind of algorithmic approach might be better suited for applications in the field of reversible logic synthesis.

The first cross-implementation is an approach to port the parallel version discussed in the previous Section to a NVIDIA GPU. When recalling the very basic idea of the first parallel approach, this means splitting up the input circuit and distributing the parts to an equally sized number of threads. In order to function, each GPU thread must have the same properties and perform the same tasks as a SPE. The difference here is that the template replacement has to be performed on the compute device and the whole template library is needed on the compute device. Each CUDA thread

	Cell/B.E. to GPU		GPU to Cell/B.E.	
function	Cell/B.E.	GPU	GPU	Cell/B.E.
plus63mod8192	5.45	42.28	1.79	18.86
plus127mod8192	8.99	84.09	3.06	35.20
urf2	44.33	528.46	36.51	947.83
urf3	25.16	512.01	13.69	271.98

Table I: Results of the first cross-implementation: Cell/B.E. on GPU.

will act as an independent compute unit and compute its own part of the circuit.

For the second cross-implementation this means that each Synergistic Processing Element is responsible for performing exactly one template match instead of computing a whole part. The responsibility of choosing the templates and executing the main loop is externalized to the PPE which is now in charge of managing more tasks. The result is a communication overhead between PPE and SPEs.

The results of these approaches showed that neither cross-implementation is suited for any further discussion, development or future work. Already small circuits like *plus63mod8192* or *plus127mod8192* showed a significantly higher execution time than the originally implemented versions. Table I shortly illustrates these results for both cross-implementations by choosing four example circuits with various sizes.

However, one important thing learned from these results is that the right combination of hardware architecture and parallel concept is important when developing parallel algorithms.

V. Results

Table II shows the results performed on the Cell Broadband Engine. The table shows two sets of results. Columns two, three and four compare the minimization result, where column two denotes the original size of the circuit before the template matching and columns three and four the result after the processing is done. The last two columns compare the actual runtimes of the sequential and the parallel version. All benchmarks have been completely executed on the Cell/B.E. with PPE for sequential execution and PPE/SPE for parallel. The parallel benchmark has been tested with six SPEs.

It can be seen that the minimization result shows equally good results as the original approach. The execution time could be improved significantly with an approximately linear speedup.

Table III has the same structure as Table II. With this implementation it is clearly visible that the minimization result lacks the precision of the sequential and the Cell/B.E. version. The speedup, however, shows even better results which is interesting, since the used GPU (NVIDIA GeForce 9400M) is a low-end version. Further tests with high end GPUs could result in even better execution times.

	Number of gates			Elapsed time	
function	orig	seq	Cell/B.E.	seq	Cell/B.E.
plus63mod4096	429	429	429	22.11	4.89
plus63mod8192	492	492	492	25.95	5.45
plus127mod8192	910	910	910	48.27	8.99
hwb7	289	284	282	13.91	3.49
hwb8	637	637	637	31.14	5.93
hwb9	1544	1541	1541	78.91	13.32
urf1	11554	7225	7225	681.13	105.82
urf2	5030	3250	3250	286.54	44.33
urf3	2732	2674	2674	147.10	25.16
urf5	10276	5582	5576	642.15	108.19
urf6	10740	5455	5455	810.13	128.26

Table II: Results of Cell/B.E. implementation. The runtime could be reduced in comparison to the sequential version.

	Number of gates			Elapsed time	
function	orig	seq	GPU	seq	GPU
plus63mod4096	429	429	429	22.11	1.65
plus63mod8192	492	492	492	25.95	1.79
plus127mod8192	910	910	910	48.27	3.06
hwb7	289	284	284	13.91	2.87
hwb8	637	637	637	31.14	1.71
hwb9	1544	1541	1543	78.91	8.19
urf1	11554	7225	7240	681.13	87.70
urf2	5030	3250	3252	286.54	36.51
urf3	2732	2674	2680	147.10	13.69
urf5	10276	5582	5583	642.15	76.37
urf6	10740	5455	5456	810.13	105.77

Table III: Results of the second approach running on the GPU.

VI. Conclusion

The results of this work show that the MMD algorithm could be successfully improved. The execution time of the template matching approach could be reduced significantly. This makes template matching as an additional optimization technique more interesting for future developments in the field of reversible logic synthesis. By exploiting the gained computational power and by optimizing the way templates are matched, e.g. by expanding the size of the look-ahead window, it might even be possible to further optimize the minimization result.

Acknowledgment

This work is supported in part by CMC Microsystems, the Natural Sciences and Engineering Research Council of Canada. We acknowledge financial support by the German Academic Exchange Service (DAAD) under the PPP project “Development Framework for Computer Vision Applications” grant No 50750255.

References

[1] D. Maslov, G. W. Dueck, and D. M. Miller, "Toffoli network synthesis with templates," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 6, pp. 807–817, 2005. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2005.847911

[2] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965. [Online]. Available: http://www.intel.com/research/silicon/moorespaper.pdf

[3] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 44, no. 1, pp. 261–269, 2000. [Online]. Available: http://www.research.ibm.com/journal/rd/441/landauer.pdf

[4] R. W. Keyes and R. Landauer, "Minimal energy dissipation in logic," *IBM Journal of Research and Development*, vol. 14, no. 2, pp. 152–157, Mar. 1970.

[5] D. Maslov, "Reversible Logic Synthesis," Ph.D. dissertation, University of New Brunswick, September 2003.

[6] D. Maslov, D. M. Miller, and G. W. Dueck, "Techniques for the synthesis of reversible Toffoli networks. eprint arxiv:quant-ph/0607166," *ACM Trans. Design Autom. Electr. Syst*, vol. 12, 2006.

[7] D. Miller, D. Maslov, and G. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Design Automation Conference, 2003. Proceedings*, June 2003, pp. 318–323.

[8] D. Maslov, G. W. Dueck, and D. M. Miller, "Fredkin/Toffoli templates for reversible logic synthesis," in *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 256–261. [Online]. Available: http://dx.doi.org/10.1109/ICCAD.2003.73

[9] IBM. Software Development Kit for Multicore Acceleration Version 3.1 - Programming Tutorial. Accessed: 2010-05-04. [Online]. Available: http://public.dhe.ibm.com/software/dw/cell/CBE_Programming_Tutorial_v3.1.pdf

[10] ——. (2007, October) Cell Broadband Engine Architecture. Accessed: 2010-05-04. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1AEEE1270EA2776387257060006E61BA/$file/CBEA_v1.02_11Oct2007_pub.pdf

[11] H. P. Hofstee. (2005, May) Introduction to the Cell Broadband Engine. White Paper. Accessed: 2010-05-04. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D/$file/2053_IBM_CellIntro.pdf

[12] Sony Computer Entertainment Inc. (2005, May) Sony computer entertainment inc. to launch its next generation computer entertainment system, in spring 2006. Press Release. Accessed: 2010-05-04. [Online]. Available: http://www.scei.co.jp/corporate/release/pdf/050517e.pdf

[13] M. Linklater, "Optimizing cell code," *Game Developers Magazine*, vol. 14, no. 4, pp. 15–18, April 2007.

[14] NVIDIA Corporation. What is CUDA? Accessed: 2010-05-04. [Online]. Available: http://www.nvidia.com/object/what_is_cuda_new.html

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Mircro*, vol. 28, no. 2, pp. 39–55, March–April 2008.

[16] J. Cohen and M. Garland, "Solving computational problems with gpu computing," *Computing in Science and Engineering*, vol. 11, pp. 58–63, 2009.

[17] T. R. Halfhill. (2009, January) Parallel Processing With CUDA. Accessed: 2010-05-04. [Online]. Available: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf