Enhancing the MMD Algorithm in Multi-core Environments

by

Michael Schlösser

B.Sc. in Computer Science, Bonn-Rhein-Sieg University of Applied Sciences, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth Kent, Ph.D., Computer Science
Rainer Herpers, Dr-Ing., Computer Science

Examinining Board: Gerhard Dueck, Ph.D., Computer Science, Chair
Andre Hinkenjann, Dr. rer. nat., Computer Science

External Examiner: Mary Kaye, Ph.D., Electrical and Computer Engineering, UNB

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

January, 2011

©Michael Schlösser, 2011
Abstract

The work done in this thesis enhances the MMD algorithm in multi-core environments. The MMD algorithm, a transformation based algorithm for reversible logic synthesis, is based on the works introduced by Maslov, Miller and Dueck and their original, sequential implementation. It synthesises a formal function specification, provided by a truth table, into a reversible network and is able to perform several optimization steps after the synthesis. This work concentrates on one of these optimization steps, the template matching. This approach is used to reduce the size of the reversible circuit by replacing a number of gates that match a template which implements the same function and uses less gates. Smaller circuits have several benefits since they need less area and are not as costly.

The template matching approach introduced in the original works is computationally expensive since it tries to match a library of templates against the given circuit. For each template at each position in the circuit, a number of different combinations have to be calculated during runtime resulting in high execution times, especially for large circuits.
In order to make the template matching approach more efficient and usable, it has been reimplemented in order to take advantage of modern multi-core architectures such as the Cell Broadband Engine or a Graphics Processing Unit. For this work, two algorithmically different approaches that try to consider each multi-core architecture’s strengths, have been analyzed and improved. For the analysis these approaches have been cross-implemented on the two target hardware architectures and compared to the original parallel versions. Important metrics for this analysis are the execution time of the algorithm and the result of the minimization with the template matching approach. It could be shown that the algorithmically different approaches produce the same minimization results, independent of the used hardware architecture. However, both cross-implementations also show a significantly higher execution time which makes them practically irrelevant.

The results of the first analysis and comparison lead to the decision to enhance only the original parallel approaches. Using the same metrics for successful enhancements as mentioned above, it could be shown that improving the algorithmic concepts and exploiting the capabilities of the hardware lead to better results for the execution time and the minimization results compared to their original implementations.
Acknowledgements

First of all, I would like to thank my supervisors Prof. Rainer Herpers (Bonn-Rhein-Sieg University of Applied Sciences, Germany) and Prof. Kenneth Kent (University of New Brunswick, Canada) for their constant support and valuable feedback.

But in particular, I’d like to mention my fiancée Judith Dittrich who supported me not only while writing this thesis but throughout my whole studies. Without her I wouldn’t sit here and write these lines: Thank you!!!
Table of Contents

Abstract i
Acknowledgments iii
Table of Contents vi
List of Tables viii
List of Figures x

1 Introduction 1
  1.1 Motivation ........................................ 1
  1.2 Structure ........................................ 3

2 Background 6
  2.1 Reversible Logic Synthesis .......................... 6
    2.1.1 Introduction .................................. 6
    2.1.2 Toffoli Gates ................................ 9
    2.1.3 Reversible Networks .......................... 11
  2.2 The MMD Algorithm ................................. 12
    2.2.1 Introduction .................................. 12
    2.2.2 Synthesis .................................... 13
      2.2.2.1 The Basic Algorithm ...................... 13
      2.2.2.2 The Bidirectional Algorithm .............. 15
    2.2.3 Template Matching ............................ 17
  2.3 The Cell Broadband Engine (Cell/B.E.) ............ 21
    2.3.1 Introduction .................................. 21
    2.3.2 The PowerPC Processing Element .............. 23
    2.3.3 The Synergistic Processing Element ........... 25
3 Related and Previous Work

3.1 Sequential approach
3.1.1 Data Structures
3.1.2 The Matching Algorithm

3.2 Parallel Approach using the Cell/B.E.
3.2.1 Prerequisites
    3.2.1.1 Hardware: Playstation 3
    3.2.1.2 Software
3.2.2 Implementation
    3.2.2.1 The parallelization concept
    3.2.2.2 The PPE implementation
    3.2.2.3 Implementation on the SPE
3.2.3 Results
3.2.4 Conclusion

3.3 Parallel Approach using NVIDIA Tesla
3.3.1 Implementation
3.3.2 Results
3.3.3 Conclusion

4 The Cross-Implementations

4.1 Cell/B.E. on GPU
    4.1.1 Design and Implementation
    4.1.2 Results
4.2 GPU on Cell/B.E.
    4.2.1 Design and Implementation
    4.2.2 Results
4.3 Analysis
4.4 Conclusion
5 Implementation Improvements

5.1 Cell/B.E. Improvement I: Optimizing Minimization Results
5.1.1 Motivation
5.1.2 Design and Implementation
5.1.3 Results

5.2 Cell/B.E. Improvement II: Vectorizing Execution by using SIMD
5.2.1 Motivation
5.2.2 Single Instruction Multiple Data on the SPE
5.2.3 Design and Implementation
5.2.4 Results

5.3 GPU Improvement I: Improving Speed
5.3.1 Motivation
5.3.2 Design and Implementation
5.3.3 Results

5.4 GPU Improvement II: Improving Minimization Results
5.4.1 Motivation
5.4.2 Design and Implementation
5.4.3 Results

6 Critique

7 Future Work

8 Conclusion

Vita
## List of Tables

2.1 Truth table defining the reversible AND operation. .......................... 9  
2.2 Example of the basic MMD algorithm. ................................. 14  
2.3 All necessary steps in order to create a circuit with the bi-directional algorithm. ................................. 16  

3.1 The runtime and minimization results of the unstructured reversible function (urf) benchmarks show that the acceleration of the algorithm was successful but at the cost of the resulting circuit size. ................................. 56  
3.2 Speedup comparison of functions with almost equal minimization behavior ................................. 56  
3.3 Comparison between the sequential and the two parallel versions. (i) sequential version(ii) parallel version ................................. 67  

4.1 Differences between original GPU- and Cross-Implementation ................................. 71  
4.2 Cross-implementation results of the URF functions on GPU in comparison to the original results on a Cell/B.E. ................................. 76  
4.3 Cross-implementation results running on a GPU showing functions with no or few template matches ................................. 77  
4.4 Cell/B.E. cross-implementation results in comparison to the sequential and the original OpenCL version ................................. 81  

5.1 The runtime and minimization results of the window based approach show that the minimization result could be optimized with a small time penalty. ................................. 94  
5.2 Results of vectorized Cell/B.E. implementation. The runtime could be reduced in comparison to the previous version. ................................. 110  
5.3 Results of the first GPU improvement. A speed-up could be achieved in comparison to the first GPU implementation. ................................. 115
5.4 Results of the second GPU improvement. The minimization result could be improved in comparison to the first GPU implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118

6.1 Comparison matrix of all implementations and architectures. . 126
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Graphical representation of Toffoli gates. NOT and CNOT are special cases of the generalized $x \times n$ Toffoli gate.</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The function $4 \times 49$ has four in- and four outputs. It consists of 16 Toffoli gates of size one to four.</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Resulting circuit for the function specified and synthesized in tables 2.2.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Circuit A shows the result of the bidirectional application. Circuit B is the naïve result of the basic algorithm shown in Section 2.2.2.1.</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>Template library defined in [1].</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>The Cell/B.E. consists of one PPE and eight SPUs that are interconnected with the Element Interconnect Bus (EIB).</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Block diagrams of both Cell/B.E. processor types.</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Structure of a CUDA Thread Processor. It consists of two Streaming Multiprocessors with each Streaming Processors each and a Shared Memory (compare [2]).</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>In order to provide code portability and scalability, CUDA source code gets compiled to a generic format. In a second step a specialized version is compiled that fits to a specific hardware configuration (Source: [3]).</td>
<td>28</td>
</tr>
<tr>
<td>2.10</td>
<td>The lifecycle of a SIMT application in five simple steps.</td>
<td>30</td>
</tr>
<tr>
<td>2.11</td>
<td>The OpenCL architecture (Source: [4]).</td>
<td>33</td>
</tr>
<tr>
<td>2.12</td>
<td>OpenCL is able to manage and communicate with more than one compute device. This way it is possible to use different hardware architectures or at least one or more of the same (e.g. NVIDIA CUDA graphics cards) (Source: [5]).</td>
<td>35</td>
</tr>
</tbody>
</table>
2.13 The OpenCL Memory Model defines an abstract view onto a potential memory hierarchy. It is the hardware vendors task to suit a sufficient mapping between abstract model and physical hardware (Source: [6]).

3.1 General processing phases of the original MMD algorithm

3.2 Mapping of reversible gates to memory

3.3 In order to process an input network in parallel it is partitioned into \( t \) parts which are processed independently. After successful termination the parts is merged to a single circuit again.

3.4 The PPE prepares the input circuit for the parallel execution and initializes the SPEs. After all SPEs have finished computation the PPE gathers the computed results and merges the circuit.

3.5 Both graphs show the growth behavior of the functions tested in Table 3.1 and 3.2. It can be seen that the parallel version decreases the runtime significantly.

3.6 An increasing number of threads leads to a better runtime behavior.

3.7 The parallel approach attempts to execute a thread on every single gate.

3.8 Figure 3.8(a) shows the values for the parallel version. The GPU is nearly twice as slow as the CPU. Figure 3.8(b) shows the current problem with large circuits with many template matches.

5.1 Different data types used by the SPE, their register layout and preferred scalar slot [7].
Chapter 1

Introduction

1.1 Motivation

The last few decades of computer hardware architecture have produced many new and powerful systems. Fast and efficient development of new chips and integrated circuits have resulted in a huge variety of application areas, like servers and home computers but also mobile devices such as cell phones or lately tablet computers. However, in the near future the race to produce even smaller and faster chips will come to a halt due to physical limitations, like transistor sizes or power dissipation.

Reversible logic is an emerging research topic that tries to overcome several problems of classical computer architecture as it is known and used today. Reversible logic has many application areas such as low-power CMOS design, quantum or optical computing. The aforementioned power dissipation
problem, for example, could be solved efficiently with the new approach. With the use of integrated circuits based on reversible logic it is theoretically possible to reduce the power or heat dissipation during computation to zero [8, 9, 10, 11].

One step in the design flow of creating a reversible circuit is the logic synthesis, or more specifically the reversible logic synthesis. Compared to classical circuits, reversible circuits have several restrictions and special properties that result in larger and more complex circuits. The MMD algorithm, originally developed by Maslov, Miller and Dueck [1], is a transformation based approach with the aim to create a reversible circuit from a formal function specification, defined by a truth table. One part of the synthesis performed by the MMD algorithm is an optimization step that aims to reduce the number of gates with a technique called template matching.

Benchmarks of the original MMD algorithm’s implementation have shown that the template matching produces good results and can contribute to minimize a given reversible circuit significantly. The advantages of smaller circuits are obvious and already apply to classical integrated circuits, like less area or lower costs, just to name two. But the aforementioned benchmarks also showed that applying the template matching approach to a circuit is computationally intensive and results in large runtimes which makes this approach impractical for a design flow or prototyping of reversible functions. With the use of modern, parallel hardware architectures, the runtime complexity of the template matching approach could be reduced and made more
efficient [12]. However, the parallel approaches showed worse minimization results than the non-parallelized, original version. In order to be more useful for future design flows it is essential to optimize the minimization result. Furthermore, improving the execution time by exploiting the underlying hardware architectures could contribute to make the template matching algorithm even more usable. Comparing the hardware architectures and parallel algorithmic approaches with each other could give a hint which of the two is more useful and efficient for future developments in this field, especially, because the template matching approach could be used for other synthesis methods other than the MMD algorithm.

Although various template matching techniques in different areas of computer science exist, the main focus of this thesis is not to investigate algorithmic improvements for template matching. Instead the focus lies on investigating how the MMD algorithm’s template matching approach can be optimized using different hardware architectures and implementations. Readers that are interested in general improvements of different template matching approaches are advised to consult existing material such as [13, 14, 15].

1.2 Structure

This work is divided into eight chapters, successively introducing a solid foundation of all topics involved in this thesis, a review of related works, a description of the actual contribution and an in-depth summary.
Chapter one starts with a brief introduction into the topic and an explanation of the relevance of this work.

Chapter two provides the reader with all background information needed in order to understand the work. It covers the basics of reversible logic, the MMD algorithm together with the template matching approach and introduces the hardware architectures which have been used in order to parallelize the algorithm. In the last Section, it covers the basics of an emerging programming framework for parallel programming, The Open Computing Language, which has been used for programming one of the parallel hardware architectures.

In Chapter three, related and previous works get introduced and reviewed. Here, the original sequential version gets discussed. The following two Sections review two works that, together with the sequential version, build the foundation of this thesis. The results discussed there give an explanation for the actual motivation behind this work.

In the fourth Chapter, two alternative implementation approaches, the cross-implementations, are introduced. These implementations were used to analyze how the two diverging parallel approaches behave, when being executed on the respective other platform. In other words, this means that the Cell/B.E. version was implemented and benchmarked on the GPU and the GPU version was implemented and benchmarked on the Cell/B.E.. The insights gained in this step are important for the following work.

The fifth Chapter then uses the previous results and analyses to further im-
prove the algorithm. The improvements performed for this work concentrate on the two main metrics, execution time and minimization result. The improved versions get benchmarked and compared to previous results.

Chapter six is a critique of the work performed in this thesis. Here, all results are discussed and compared with respect to the used hardware architectures and parallelization concepts. The critique draws a line beneath all gathered results and tries to provide advice for further developments in this area.

In Chapter seven, several aspects that might be interesting for future developments, and that might help to further improve the template matching approach are discussed.

Chapter eight draws a conclusion and summarizes all advantages and disadvantages that are related to this thesis work. Additionally, it tells the reader in a brief and conclusive manner if the motivation behind the work is justifiable, and if the intended results could be achieved.
Chapter 2

Background

2.1 Reversible Logic Synthesis

2.1.1 Introduction

As in the classical Logic Synthesis, the main goal of Reversible Logic Synthesis is to generate a circuit from a formal function specification. The reversible approach, however, differs in a few points from the aforementioned classical one. In the following, an introduction to Reversible Logic Synthesis is given. Mainly, the aforementioned differences are described and explained. This knowledge is necessary in order to understand the fundamental concepts of this work and in particular the MMD algorithm. Reversible Logic Synthesis has its root and application in several fields like quantum computing, low-power CMOS, nanotechnology and optical computing [16]. 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 [17], one big problem, heat dissipation, has not been solved yet. Among other problems, this very issue is one of the main motivations 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, however, is dissipated as heat which has been shown by [10, 11].

The most common and widely used operations are irreversible, such as the logical AND, OR and XOR operations. Looking at the logical AND operation will clarify what this actually means. This simple binary function has two inputs and one output which implies that if two bits of information go into the function and only one comes out, one bit of information is lost during the computational process. This lost information, however, generates the aforementioned heat. It is obvious to see that this applies to the other mentioned functions as well. Another very common and basic function, the binary NOT function, in contrast to the aforementioned functions, is reversible. One reason why it is reversible is due to the fact that no information is lost. The binary NOT function has one in- and one output. This implies that no heat is dissipated.

So, one major property of every reversible function is that the number of inputs and outputs is equal. Another fundamental property of each reversible function is that any output pattern has a unique preimage. In other words,
every reversible (boolean) function $f(x_0, x_1, ..., x_n)$ is a bijection and performs permutations of the set of input vectors [16]. Looking at the binary NOT function this holds true. Each input $x$ is simply flipped which conforms to the aforementioned restriction.

One question that comes to mind is if it is possible to transform an irreversible function to a reversible one. This question can be answered with yes. The logical XOR function $\oplus$, for example, can be transformed into a reversible function by simply adding an output which maps $x$ to $\bar{x}$. The result is a function $(x \oplus y) \rightarrow (\bar{x}, x \oplus y)$ with two in- and outputs and a unique output pattern with the output vector $(10, 11, 01, 00)$. One can clearly see that the basic properties of reversible logic hold.

Another example is the logical AND operation. However, this operation cannot be transformed into a reversible function by just adding a single output that worked with the XOR function. Since it is not possible to create a unique output pattern this way, the function needs to be extended. In order to create a reversible AND function, it is necessary to add one input and two outputs and map the set of input vectors to the vector $(x, y, z \oplus xy)$. By introducing this mapping, the output vector can be transformed as shown in Table 2.1. For all constant $z = 0$ this function realizes the logical AND [8].

A logical consequence of the aforementioned restrictions of reversible gates is that a lot of inputs and outputs are introduced. Additional inputs are called \textit{constant inputs} and the corresponding outputs are called \textit{garbage}. Let $I$ be
the set of all inputs (control and target) and $C$ be the set of all constant inputs. $O$ denotes the set of all outputs and $G$ the set of garbage lines. Formula 2.1 shows the relation between these sets [8].

$$|I| + |C| = |O| + |G|$$  \hspace{1cm} (2.1)

### 2.1.2 Toffoli Gates

The definition provided by Table 2.1 describes the basic functionality of one of the best known and well studied reversible gates, called the Toffoli gate. This type of gate is of special importance for this work, since Toffoli gates are

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z \oplus xy$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.1: Truth table defining the reversible AND operation.
the fundamental operation used in the MMD algorithm. Table 2.1 defines a Toffoli gate of size three which means three in- and outputs.

A Toffoli gate does not have to be of size three which means it can be generalized. It can be defined as a function with \( n \) inputs and outputs that passes the first \( n-1 \) inputs through unchanged and inverts the \( n^{th} \) input if all others are 1 [1]. Formally, a \( n \times n \) Toffoli gate can be written as \( T^F_n(x_1, x_2, ..., x_n) \) where \( x_1 \) to \( x_{n-1} \) are called control and \( x_n \) is called target. After passing through the gate, the value of each input vector component becomes

\[
\begin{align*}
x'_i & = x_i, \quad i < n, \\
x'_n & = x_1 x_2 ... x_{n-1} \oplus x_n
\end{align*}
\]

A common graphical representation used for Toffoli gates is illustrated in Figure 2.1. Here gates of sizes one, two and three are shown. All lines corresponding to an input \( x_i \) with \( i < n \) are depicted by a • and each target line by the \( \oplus \) operation symbol.

![Graphical representation of Toffoli gates](image)

Figure 2.1: Graphical representation of Toffoli gates. NOT and CNOT are special cases of the generalized \( x \times n \) Toffoli gate.
A Toffoli gate of size one, simply is a negation of the input. It does not have a control input. A size two Toffoli gate, also called Feynman or Controlled NOT (CNOT) gate, realizes an ordinary XOR operation on the target line. The Toffoli gate of size three is a generalization of all Toffoli gates of bigger size, since it defines how to map the value of the target line [1]. Several other reversible functions exist but since they are not of special interest for this thesis, an explanation is omitted but can for example be found in [1].

2.1.3 Reversible Networks

A single reversible function is rather uninteresting, such as a single AND function. Like their irreversible counterparts, reversible gates can be combined in such a way that they form a circuit. Like a single reversible function, a reversible circuit or network underlies certain restrictions. First of all, it is described by a cascade of reversible gates, such as Toffoli gates. Figure 2.2 shows such a cascade, with four lines (one line per input/output) and 16 Toffoli gates of various sizes. The term cascade simply means that all gates are strictly ordered and that only one gate is active at a time. Its output is the input for exactly one subsequent gate, implying a sequential flow through the circuit.

One goal of reversible logic synthesis is to create such a reversible network as described above, for example by translating a formal function specification $f(x_1, x_2, ..., x_n)$ provided by a truth table into a Toffoli based circuit. Due
Figure 2.2: The function $f_{4,49}$ has four in- and four outputs. It consists of 16 Toffoli gates of size one to four.

to additional constant inputs and garbage, reversible circuits get bigger than conventional boolean logic networks. Therefore research does not only concentrate on the synthesis but also on the minimization of such networks by applying a variety of techniques, like minimizing garbage or the total number of gates.

2.2 The MMD Algorithm

2.2.1 Introduction

The MMD algorithm, originally introduced in [1], aims to provide a set of methods in order to synthesize and minimize an arbitrary reversible function $f$ using Toffoli gates. It is split into two parts. The first one is the synthesis step which generates a reversible network from a formal function specification, represented as a truth table. Several optional modifications can be applied that in a lot of cases result in a smaller circuit, in terms of the number of gates and control inputs [16].

The second step is a transformation based approach, using a method called template matching in order to further reduce the number of gates or the total
cost of a circuit. The very basic idea behind this procedure is to find a set of gates in a given circuit where it is known that they can be replaced by a smaller or equally sized number of gates that implement the same function.

2.2.2 Synthesis

2.2.2.1 The Basic Algorithm

The basic synthesis algorithm is a greedy, naïve approach to identify Toffoli gates on the output side of the specification [1]. In order to achieve this, the Toffoli gates are chosen to progressively transform the output part of the function specification into the input part. A basic, yet very important assumption is that already transformed lines in the truth table are not affected by the following ones, leading to a strict order of the input side of the truth table.

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 attempts to transform the output mapping by applying Toffoli gates in such a way that after the successful termination

\[
f'(i) = i, \forall \ 0 \leq i \leq 2^n - 1
\]  

(2.4)

holds. A correctness analysis of this approach is shown in [16] and [1].
(a) Truth table of a reversible function, before applying the MMD algorithm.

<table>
<thead>
<tr>
<th>(cba)</th>
<th>(i)</th>
<th>(f(cba))</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>111</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

(b) Truth table after applying the basic MMD algorithm. After four iterations the synthesis step is finished, with \(cba = c^4b^4a^4\)

<table>
<thead>
<tr>
<th>(cba)</th>
<th>(ii)</th>
<th>(iii)</th>
<th>(iv)</th>
<th>(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>110</td>
<td>100</td>
<td>101</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>111</td>
<td>111</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 2.2: Example of the basic MMD algorithm.

To clarify how the algorithm synthesizes a given reversible function specification, consider the reversible function given in Table 2.2(a). It is obvious that the aforementioned requirement from Equation (2.4) does not hold. In order to fulfill it, the basic algorithm performs a set of steps. The first one tries to map the input vector \((0, 0, 0)\) to the desired output \((0, 0, 0)\). To achieve this, each bit, and therefore each corresponding line in the circuit, not equal to 0 is negated with a NOT gate. In this example, one NOT gate is required, flipping all outputs of line \(a\). Table 2.2(b) shows this in column \(ii\). As a result of this operation, the first five outputs map correctly to their inputs, leaving three outputs to modify. By adding a Toffoli gate of size three with \(a\) as the target line, the outputs change to column \(iii\) of Table 2.2(b). The rest of the application is straightforward and results in the desired state, respectively circuit, after adding two more Toffoli gates. Because the synthesis process
always modifies the function output the circuit is read in reverse in order to apply the gates.

![Circuit Diagram]

Figure 2.3: Resulting circuit for the function specified and synthesized in tables 2.2

Because this is a greedy approach, it results in a big network (circuit) with less than or equal to \((n - 1)2^n + 1\) (worst case) gates. This number of gates can be reduced by applying further approaches, like Control Input Reduction, Bidirectional application and Asymptotically Optimally Modification, also shown in [16] and [1].

### 2.2.2.2 The Bidirectional Algorithm

The basic algorithm, described in Section 2.2.2.1, always attempts to find a sequence of Toffoli gates on the output side of a function in order to accomplish its goal. Since the functions are reversible, it is also possible to do the opposite and see which direction results in a smaller circuit. However, it turns out that it is even possible to apply the algorithm bidirectionally [16]. This means that the algorithm applies its rules simultaneously on both the input and the output side, eventually yielding in a smaller circuit [8].

Again, to clarify the procedure an example shows the basic steps for the bidirectional algorithm.
Table 2.3 shows another reversible function specification. When applying step 1 of the basic algorithm, three NOT gates on the output side are needed in order to map the input 000 from 111 to 000. A closer look shows that applying only one NOT gate to the input value 001, has the same effect. When changing the input side, it is necessary to reorder the input side in such a way, that it is in standard truth table order again. This results in column ii. A simple reordering of the values 001 and 010 on the input size achieves the next step shown in column iii. The reordering is done using a Toffoli gate with a target line on $b$. For the last step a Toffoli gate of size three with its target on $c$ can be used. After the last application the synthesis is complete.

It can be seen in Figure 2.4 that the bidirectional approach only needs three gates, whereas the basic approach results in seven gates.

<table>
<thead>
<tr>
<th>cba</th>
<th>$i$</th>
<th>$ii$</th>
<th>$iii$</th>
<th>$iv$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c^0 b^1 a^1$</td>
<td>$c^1 b^0 a^1$</td>
<td>$c^2 b^2 a^1$</td>
<td>$c^3 b^3 a^1$</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>111</td>
<td>000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>111</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>001</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>001</td>
<td>111</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>011</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
<td>011</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>110</td>
<td>101</td>
<td>011</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 2.3: All necessary steps in order to create a circuit with the bidirectional algorithm.
Figure 2.4: Circuit A shows the result of the bidirectional application. Circuit B is the naïve result of the basic algorithm shown in Section 2.2.2.1.

2.2.3 Template Matching

To further simplify or reduce the number of gates in a network, in the second phase a concept called 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 in 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 the set of gates with $|A| = n$ realizing a function $f$ and $B$ the second set of gates with $|B| = m$ and $m \leq n$ realizing a function $g$. Then $A$ can be replaced by $B$, if $f = g$.

In [16], a template was initially defined as two sequences of gates that realize the same function. The first sequence of gates, gets matched against a given network. If a sufficient match is found it then gets substituted by the second sequence [1, 16]. Figure 2.5 illustrates the basic principle and shows a library
of templates, with its first and second sequence as used in [1]. It is important
to mention that the lines of each template are generic. This means that the
template’s lines must be associated with lines in the given circuit and then
need to apply the association consistently across the whole template. The
procedure is done by finding a match for one gate and then looking for
further matches for the other gates. It is important to mention that the
further matches do not have to be necessarily adjacent. When sticking to a
simple rule, it is possible to move gates in the circuit in such a way that they
do not affect the computation. This rule has been described in [1]. It states
that two adjacent Toffoli gates can be interchanged, iff the target line of gate
one does not intersect with the control lines of gate two and vice versa.

Let \( TOF_n(x_1, x_2, ..., x_n) \) be a Toffoli gate of size \( n \) where \( x_n \) is the target.

Then two adjacent Toffoli gates \( TOF_k(x_1, x_2, ..., x_{k-1}, x_k) \) and \( TOF_l(y_1, y_2, ..., y_{l-1}, y_l) \)
can be interchanged, iff the following holds

\[
x_k \notin \{y_1, y_2, ..., y_{l-1}\} \tag{2.5}
\]
\[
y_l \notin \{x_1, x_2, ..., x_{k-1}\} \tag{2.6}
\]

This way it is possible to shift gates in the circuit in such a way that a
template can be applied. Due to the two fundamental properties of reversible
functions (see Section 2.1) it is also possible to apply each template in reverse.

It is obvious that in this case, each substitution has to be applied in reverse
too.
4. TEMPLATE MATCHING
The circuits produced by the algorithm as described thus far ...
inputs and outputs. This can be viewed as extending the irreversible function specification to a larger reversible one.

Figure 2.5: Template library defined in [1].
In [16] and [18], an alternative view on templates has been introduced which has been used in this project. This alternative view is based upon a couple of observations regarding properties of reversible circuits.

**Observation 1:** Toffoli gates are self inverse, this means $G = G^{-1}$ holds. Considering a network $G_0G_1...G_{m-1}$ realizing a function $f$ then the network $G_{m-1}^{-1}G_{m-2}^{-1}...G_0^{-1}$ realizes $f^{-1}$. If $f$ realizes the identity function, denoted as $Id$ then this is true for $f^{-1}$ as well.

**Observation 2:** Any rewriting rule of the form $G_1G_2...G_k \rightarrow G_{k+1}G_{k+2}...G_{k+s}$ ensures the following is satisfied

$$G_1G_2...G_kG_{k+s}^{-1}G_{k+s-1}^{-1}...G_{k+1}^{-1} = Id \quad (2.7)$$

**Observation 3:** For $G_0G_1...G_{m-1} = Id$ and any parameter $p$, $0 \leq p \leq m$ $G_0G_1...G_{p-1} \rightarrow G_{m-1}^{-1}G_{m-2}^{-1}G_{m-p}^{-1}$ is a valid rewriting rule.

**Observation 4:** If $G_0G_1...G_{m-1} = Id$, then the network can be rewritten $m$ times in the form $G_1...G_{m-1}G_0 = Id$, $G_2...G_{m-1}G_0G_1 = Id$ and so on, resulting in $m$ so called cycles.

Using these four observations and the resulting properties, it is possible to redefine a size $m$ template as a cascade of $m$ gates that realizes the identity function. One important restriction is that at least one cycle cannot be reduced in size by applications of smaller or equal size templates [18].
2.3 The Cell Broadband Engine (Cell/B.E.)

2.3.1 Introduction

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 [19]. 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. As mentioned earlier, the Cell/B.E. is a heterogeneous processor. In contrast to a homogeneous multi-core CPU, like an Intel Core 2 Duo, 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) [20, 21]. The second type of processor is the so called Synergistic Processor Element (SPE) which is a highly specialized and lightweight CPU, compared to the PPE [19]. 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 2.6 illustrates the very basic scheme of a Cell Broadband Engine Architecture compliant processor, such as the Cell Broadband Engine.

On an architectural level, the Cell/B.E. has been designed in order to provide an extensible and scalable architecture that achieves a solution for the following goals as described in [19]:

21
Figure 2.6: The Cell/B.E. consists of one PPE and eight SPUs that are interconnected with the Element Interconnect Bus (EIB).

- optimize memory latency
- minimize energy consumption
- scale frequency limitation

When examining the underlying concept of the CBEA and how the processors were chosen, it becomes clear that the Cell/B.E. is able to provide the aforementioned criteria. In this concept, the PPE serves as a general purpose CPU, which can be found in most modern computers. Since it is built upon a well known standard, a lot of application do not have to be redesigned and can reuse the capabilities of the PowerPC Architecture. When developing a Cell/B.E. application, the PPE can be used for any kinds of tasks, like running an operating system, managing system resources or other control intensive tasks (control-plane). The SPEs on the other hand are designed to fulfill compute intensive tasks (data-plane).
The usual structure of a Cell/B.E. program is to implement the actual control logic on the PowerPC Processing Element. The computationally intensive work is delegated to the different Synergistic Processing Elements which are optimized for these tasks. One could describe the PPE as the brain of an application, whereas the SPEs serve as highly effective workers that are able to perform a lot of computational work in parallel. However, this design implies a strong dependency relationship. On the one hand, the PPE is not capable of doing highly parallel and computationally intensive work on its own which is why it is dependent on the SPEs. On the other hand, however, the SPEs are not designed to fulfill control intensive tasks, so they rely on the PPEs control logic. But in conjunction, both processors are able to fulfill the design criteria as described in the specification.

2.3.2 The PowerPC Processing Element

The PowerPC Processing Element, is based on the 64-Bit PowerPC architecture, can be programmed like any other PowerPC compliant CPU and supports both the 32-Bit and the 64-Bit instruction set. This way it is easily possible to port programs that have already been developed for a PowerPC architecture. Additionally, the PPE support dual-threading which makes Simultaneous Multithreading (SMT) possible [22].

As depicted in Figure 2.7(a) the PPE consists of several parts, of which one is the PowerPC Processing Unit (PPU) and the other one is the PowerPC Processor Storage Subsystem (PPSS).
These language extensions give C/C++ programmers much greater control over code performance, without the need for assembly-language programs (Figure 2.7).

Figure 2.7: Block diagrams of both Cell/B.E. processor types

The PPU is responsible for instruction control and execution. It includes the full set of 64-bit PowerPC registers, two 32 KB Level 1 caches for instructions and data, and several units like load and store, fixed and floating point units and so on. Besides these standard PowerPC features, it supports a Vector instruction set in order to perform Single Instruction Multiple Data (SIMD) calculations. Additionally, it supports multithreading and handles two threads that can be seen as a 2-way multiprocessor with shared dataflow [19].

The PPSS on the other side, handles all memory requests, from and to the PPE. It contains one big 512 KB large level 2 cache, various queues and a bus interface to the EIB [19].
2.3.3 The Synergistic Processing Element

The Synergistic Processing Element (SPE) is the second type of processor that gets described by the CBEA specification. In contrast to the PPE, the SPE has been newly developed for the Cell/B.E.. As already addressed in Section 2.3, the Synergistic Processing Element is a lightweight processor compared to the PPE. It uses a RISC instruction set and does not have any caches at all. Instead it works with Direct Memory Access over its Memory Flow Controller (MFC). This controller is responsible for interacting and communicating with the Element Interconnection Bus and the Main Memory. All fetched information gets stored in the SPE’s Local Store (LS) which is a 256 KB big local memory and is used for all data and program-code. It is capable of transferring 16 Bytes per cycle to the Synergistic Processing Unit’s execution unit, the Synergistic Execution Unit (SXU) which is part of the SPU [22]. Figure 2.7(b) illustrates the general structure of the SPE. Each SPU is an independent processor that can work on tasks assigned by the PPE. Due to the processor architecture these can be individual tasks, e.g. for a task parallelism approach, or collaborative tasks where each SPE works on the same set of data (data parallelism).
2.4 NVIDIA Tesla

2.4.1 Introduction

Tesla is an architecture for Graphics Processing Units (GPU) from NVIDIA. From the beginning, one design goal was not only to support the video gaming industry, which usually uses high end graphics cards for their productions, but also to target the usage of GPUs for scientific purposes. The idea is to establish the GPU as a co-processor that for example could be used for scientific simulations and calculations. Being available in the 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 [23, 24, 25].

2.4.2 Architecture

In order to provide a sufficient amount of processing power, the GPU is divided into several compute units called Streaming Multiprocessors (SM). The number of these units is not fixed and varies depending on the used graphics card. As a general rule of thumb one can say that the more SMs are on the GPU, the higher is the possible parallelism. Each of the Streaming Multiprocessors consists of several pieces from which the most important ones are the Streaming Processors (SP), and the local memory which is shared among the SM’s component parts. Each SM has exactly eight SPs which in contrast to the SMs is a fixed number. The so called Shared Memory, can
store 16KB and delivers fast access to the data. Figure 2.8 illustrates the architecture.

Figure 2.8: Structure of a CUDA Thread Processor. It consists of two Streaming Multiprocessors with each Streaming Processors each and a Shared Memory (compare [2]).

Since the shared memory of each Streaming Multiprocessor is very limited, each Tesla compatible GPU has several other possibilities to store data. Especially for large data sets this is important. 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. The size of global memory is dependent on the actual graphics card.
2.4.3 The CUDA Programming Model

In order to write a Tesla compliant program, NVIDIA invented the programming paradigm called CUDA and delivers a Software Development Kit consisting of an API, bindings for several programming languages such as C or C++ and a compiler that translates the actual source code. In order to be portable to different CUDA based graphics cards, the source code is compiled into a generic byte code format that, in a second step, is translated to a hardware specific byte code. This way NVIDIA ensures portability and scalability of CUDA programs.

Figure 2.9: In order to provide code portability and scalability, CUDA source code gets compiled to a generic format. In a second step a specialized version is compiled that fits to a specific hardware configuration (Source: [3])

The basic programming model behind the CUDA paradigm is the Single Instruction Multiple Thread (SIMT) paradigm. This model is related to a Single Instruction Multiple Data (SIMD) approach with the difference that
the CUDA architecture does not have dedicated vector processing units on the chip. The basic idea of the SIMT approach, however, can simulate the SIMD processing with its huge amounts of Processing Elements. This means that each Processing Element can be seen as a vector component processing unit. This way it is possible for the programmer to simulate a SIMD engine. CUDA defines a set of these units as so called warps with a size of 32 threads. Not only can a programmer simulate a SIMD engine but also write thread-level code for independent threads [24].

The basic idea behind this model is to separate the program logic into two parts: the host and the kernel. The host has the role of a controller. That means the program logic is present in this part of the program. This includes different tasks like setting up the manycore environment, loading data-structures and preparing the data for parallel computation. The kernel on the contrary is a quite minimal piece of code that is dispatched to the different cores and defines the actual parallelism. This means one kernel definition is instantiated \( n \) times and each of those instances is responsible for working on a piece of data provided by the host program. The number of kernel instances that can run in parallel is strongly dependent on the actual hardware architecture.

The usual lifecycle of a CUDA application involves the following steps:

1. initialize host and device

2. allocate memory on device
3. transfer data to device

4. computation on device

5. send results back to host

6. free memory on device

A kernel is defined as a regular C style function with a special qualifier that marks this piece of code as a kernel. Listing 2.1 shows an example kernel that computes the square of each component of an array $a$ and stores the result in an array $b$. 

Figure 2.10: The lifecycle of a SIMT application in five simple steps.
Listing 2.1: CUDA kernel that calculates the square function

This minimal approach, however, is an advantage and a disadvantage at the same time. The advantage of this concept is the lightweight interpretation of what a thread should be. Each kernel has its inputs and at least one output and performs a dedicated task on a set of data. A good example for this is the matrix multiplication.

When it comes to more complicated calculations that maybe need dynamic memory allocation, task switching or significant branching, this approach quickly becomes cumbersome to program. These minimalistic, lightweight threads are pure executional units that can perform a limited set of tasks. Used wisely they do it really good and first of all fast.

Additionally, it is up to the programmer to ensure that all computational resources are used as good as possible. It is not a good idea to calculate too small problems with this method, since the organizational overhead costs
more than any advantages gained. Given a sufficiently large problem the number of threads and its workload and distribution are another tough task to solve.

2.5 The Open Computing Language

2.5.1 Introduction

The Open Computing Language (OpenCL) is a royalty-free, cross-platform specification for programming multi- and manycore architectures [5]. The first version of the OpenCL specification was released on December 2008 [26] and is maintained by the Khronos Group, an independent consortium of media-centric companies focussing on the creation of open standards for various technologies and platforms, including parallel computing, graphics and dynamic media [27]. A vastly known standard maintained by the Khronos Group is the Open Graphics Library, better known as OpenGL. Current members of the working group are among others AMD, NVIDIA and IBM. These three members already provided implementations for their hardware architectures CUDA, Stream and Cell/B.E. (see [28, 29, 30]). Figure 2.11 illustrates the general architecture of an OpenCL application and all the involved layers, starting from the application level and going down to the hardware level.
2.5.2 Architecture

The architecture is divided into five layers. The two lowest layers represent hardware specific logic that differs from platform to platform. Where the hardware layer represents for example a GPU, the next layer exposes a driver interface to the next layer, the OpenCL runtime. This is the first layer of interest that implements the internal OpenCL architecture on a platform level. That means the OpenCL runtime is a custom platform based implementation. The OpenCL API layer defines an interface for programmers that enables them to interact with the runtime. The last layer is the actual application that utilizes the features provided by OpenCL.

On a technical level, OpenCL utilizes several standards in order to fulfill the task of an abstract, device agnostic programming framework. The programming language used for writing OpenCL compliant programs is a C99 derivative with several extensions for parallel computation like special qualifiers (see [31, 5]). A second important standard utilized is the IEEE 754 stan-
standard used for providing consistent floating point arithmetic on or between devices [32]. This way an OpenCL application can ensure the correctness of calculations. Using these two standards as a solid basis, it is possible to write platform independent programs that are robust and portable.

2.5.3 Programming Model

OpenCL uses a programming model similar to SIMT, called Single Program Multiple Data (SPMD) [5]. Each application gets split into two parts: host and device. The host is responsible for the management of program logic and devices. Usually, this is the CPU running the operating system. The device could be any OpenCL compliant hardware, even the host CPU itself. In the context of manycore applications one or more GPUs can be used, which is the real benefit of OpenCL. Not only is it possible to create a device agnostic application in which the hardware can be replaced (e.g. NVIDIA by ATI) but different hardware architectures could be mixed in order to maximize the outcome (e.g. manycore together with multicore). Of course one should clearly keep in mind that this is no trivial task but the possibility makes OpenCL a future proof framework for parallel programming.

Similar to the SIMT programming model SPMD uses the concept of kernels that are submitted and instantiated on a compute device. In OpenCL each instance of a kernel is called a work-item and is organized in work-groups. The organization into groups and items enable a programmer to implement coarse or fine grained parallelism, dependent on the application’s or algo-
Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item.

Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space supported in OpenCL 1.0 is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension. Each work-item’s global ID and local ID are N-dimensional tuples. The global ID components are values in the range from zero to the number of elements in that dimension minus one.

Figure 2.12: OpenCL is able to manage and communicate with more than one compute device. This way it is possible to use different hardware architectures or at least one or more of the same (e.g. NVIDIA CUDA graphics cards) (Source: [5])

The definition of an OpenCL kernel is similar to that of a CUDA kernel. Listing 2.2 is the OpenCL version of the kernel already shown in Section 2.4.3. The function and variable qualifiers differ from the CUDA version. The determination of the kernel id is a simple function call.
Listing 2.2: OpenCL version of the square kernel

In order to provide a unified view onto a compute device’s memory, the OpenCL standard also defines a memory model that is the same for all implementing architectures. In this model the accessible memory is composed of four different types (compare [5]):

1. Global Memory

2. Constant Memory

3. Local Memory

4. Private Memory

Global Memory is available to both host and device. It permits read and write access to all work-items and work-groups. Dependent on the hardware
architecture, caching is possible.

Constant Memory is a region of Global Memory that does not get changed during runtime. Compute Devices have read only access, whereas the Host is able to write into Constant Memory.

Figure 2.13: The OpenCL Memory Model defines an abstract view onto a potential memory hierarchy. It is the hardware vendors task to suit a sufficient mapping between abstract model and physical hardware (Source: [6]).

It is important to keep in mind that OpenCL is a standard, not a ready made software solution. This means that each hardware architecture needs its own, particular implementation of the specification. It is the programmers task to ensure that all requirements that have been defined in the specification are properly considered. This does not only include implementing the function prototypes that are provided by the Khronos Group but also the abstraction
and the mapping of the physical hardware architecture to the OpenCL Layer Model.
Chapter 3

Related and Previous Work

The following chapter describes three different implementations of the MMD algorithm and builds the foundation for following chapters.

The first section deals with the original, sequential implementation of the MMD algorithm which serves as the basis for the following parallel applications. It is based on the theory described in Section 2.2.

In the following two sections two different, parallel approaches to accelerate the MMD algorithm are presented. The first one uses the Cell Broadband Engine and the second one utilizes an NVIDIA GPU in combination with OpenCL. Since two different hardware architectures that target different execution models have been used, both parallel approaches differ significantly from each other in order to utilize the underlying parallel programming paradigms.
3.1 Sequential approach

The original implementation of the MMD algorithm is based on the papers published by Maslov, Miller and Dueck. Basically, it is a sequential step by step implementation of the theoretical approach described and discussed in the related papers [1, 16, 18].

It is possible to divide the program into three phases. During the first phase, the program loads an already synthesized reversible network. After that a template library is loaded that is needed for the last step, the actual template matching algorithm.

Figure 3.1: General processing phases of the original MMD algorithm

The following two sections will describe the necessary data structures and how they are created in order to run the matching algorithm.
3.1.1 Data Structures

As already mentioned before, phase one loads an already synthesized reversible circuit from an external file at runtime. The specification for those circuits are encoded in a plain text file-format and is explained in [33]. A circuit is a list of gates which have to suffice the properties of reversibility, as described and explained in Chapter 2.1.

The practical realization, written in the C programming language, uses the struct `circuit_t` in order to describe a reversible circuit. It contains three variables which store the cost of the circuit implementation (`totalCost`), the number of gates (`numGates`) and the number of in- and outputs of that circuit (`numLines`). According to the rules of reversibility, the number of in-and outputs is always equal which means that only one variable is needed to store this value. A fourth variable then defines a pointer to a buffer of size `numGates` which contains the definition of its gates.

The gates are defined in the struct `gate_t` and originally contain the variables `swapf`, `target` and `control`. The variable `swapf` is of type Boolean and stores if a gate is a so called SWAP-gate as defined in [1]. Since this implementation of the MMD template matching algorithm only uses Toffoli gates, this variable can be omitted. It only has been included for future releases or extensions of the implementation that might consider SWAP gates. The variables `control` and `target` store the position of the respective lines. As explained in Chapter 2.1, each reversible Toffoli gate with \( n \) in- and outputs has exactly one target line and \( n - 1 \) control lines. Both variables are represented as integers that
describe a bitwise mapping of the lines. This means that each line of a circuit or gate is related to a single bit of that integer.

![Diagram of bitwise mapping of lines](image)

Figure 3.2: Mapping of reversible gates to memory

For a 32-bit machine this means that circuits with up to 32 lines can be calculated. Circuits with more in- and outputs cannot be used unless a 64-bit architecture is available. Despite the fact that this solution saves memory, another advantage is the possibility to use binary operators like bit-shift in order to work with the data.

Similar to circuits, the templates are stored in external files are loaded at runtime. As already described in Section 2.2.3, a template is a kind of reversible network which calculates the identity function $id$. From these circuit specifications it is now possible to derive a set of patterns which can be used for the template matching. These patterns are represented by the structs $cpat_t$ and $tpat_t$ and are referenced in the struct $template_t$. The variable $ng$ stores the size of the template (number of gates) and the variables $csize$ and $tsize$ describe how many target- and control-patterns are available. The last two mentioned variables share the relation
The struct `cpat_t` stores a pattern for each line of the template which consists only of controls. The struct `tpat_t` stores patterns for all lines that include targets only or a mixture of targets and controls. Similar to the encoding of gates these patterns get stored as bit patterns too. Again, it is possible to save memory and use fast memory operations with binary operators.

### 3.1.2 The Matching Algorithm

When all data structures are configured, the application can enter the third and last phase which is the actual template matching.

Algorithm 1 shows the sequential version of the template matching approach. It takes two inputs: a synthesized circuit \( C \) and the template library \( T \), both encoded as described in Section 3.1.1. After successful termination the algorithm returns a modified circuit \( C' \) with

\[
|C'| \leq |C| \tag{3.2}
\]

The algorithm starts at gate 1 and template 1 (array index 0) and tries to match the selected template. There are two possible outcomes; either the selected template finds a pattern that can be replaced or the attempt fails. If no match could be found starting from gate \( \text{start} \), the algorithm tries to match all templates, where \( |T| \) denotes the number of templates in the
Algorithm 1 Sequential Template Matching

1: input: $C, T$
2: output: $C'$
3: $C' = C$
4: $start = 1$
5: while $start < |C'|$ do
6:   $i = 0$
7:   flag = true
8:   while flag == true do
9:     if $t_i \in T$ matched then
10:        flag = false
11:        start = offset()
12:     else
13:        $i = i + 1$
14:        if $i \geq |T|$ then
15:           start = start + 1
16:        end if
17:     end if
18:   end while
19: end while
library. In other words, the starting position \( start \) does not get changed until all attempts to match the templates have failed. If this is the case, the algorithm proceeds to the next gate and repeats the whole procedure again.

If a template can be matched and replaced successfully, the procedure starts again with the first template. During the matching procedure, the algorithm keeps in mind which positions have failed to match. It is obvious to see that these gates do not have to be matched again. To avoid these unnecessary matches and instead of setting \( start = 1 \), it is set to a value offsetting these gates.

### 3.2 Parallel Approach using the Cell/B.E.

In a first attempt to accelerate the MMD algorithm, the Cell Broadband Engine has been employed. As already discussed in Section 2.3, the Cell Broadband Engine is a heterogeneous multi-core CPU, consisting of a full-featured PowerPC processor and eight dedicated lightweight processors. The proposed approach tries to distribute the work equally on the available SPEs and take advantage of the available multi-core structure.

#### 3.2.1 Prerequisites

**3.2.1.1 Hardware: Playstation 3**

The actual hardware available for this work is the Sony Playstation 3 (PS3). This device is a gaming console, developed by Sony Computer Entertainment
and firstly announced in 2005 [34]. While the main purpose of this device lies in gaming, the first versions of the PS3 could also be used for other tasks, since it was possible to install a GNU/Linux based operating system. The interesting fact for this work, however, is that the Playstation 3 uses a Cell Broadband Engine in order to fulfill its work. The used Cell/B.E. runs at 3.2 GHz on both processors, the PPE and the SPE. The total floating point performance was announced as 218 GFLOPS [20, 19, 34].

Compared to the official specification (CBEA), the PS3 is only equipped with seven instead of eight SPEs. However, in order to make the manufacturing process of the Cell/B.E. more cost effective, one of the seven SPEs is disabled, leaving only six freely programmable SPEs [35, 36].

Several reasons led to the decision of a Playstation 3 for the implementation. Unlike other products on the market that use Cell/B.E. processors as well, for example the IBM BladeCenter QS21 [37], the Playstation turns out to be the most cost-effective solution. While an IBM BladeCenter QS21 starts in an upper four digit range, the Playstation 3 is situated in a lower three digit range. this makes the Playstation 3 a perfect prototyping platform. Additionally, the Playstation has has been successfully used for scientific applications before [36]. The PS3 provides everything needed for different scientific purposes. Another interesting fact about the PS3 is that it has a network interface which could for instance be used to distribute work to other PS3s and therefore increase computational power.

As already mentioned above, Sony provided the ability to install a Linux
distribution on the Playstation. Coupled with the official IBM Software Development Toolkit, this feature makes it possible to utilize the device as a normal Unix-based computer.

Unfortunately, Sony removed the ability to install Linux with the release of the latest version, the Playstation Slim in August 2009 [38]. The published firmware upgrade from April 2010 also removed this feature for all older versions [39]. However, it is still possible to use the Linux installation under the condition not to upgrade the PS3 system. But there is no doubt that Sony has made the usage of the PS3 as a cost-efficient solution for scientific applications not only harder but also unattractive.

### 3.2.1.2 Software

In order to start developing on the Playstation 3, a few preparations have to be made. The first step is to install a GNU/Linux distribution and the second step is to install and configure the IBM Software Development Kit. The following two paragraphs describe the aforementioned steps in order to be able to start developing.

There exists several Linux distributions that can be installed on a PowerPC architecture like the Cell/B.E.. However, the Fedora distribution is officially supported by IBM and since IBM also provides the official Software Development Kit, that runs well on Fedora, this distribution has been chosen. A more detailed description of how to install and configure GNU/Linux can be found in [12].
After installing the operating system, the Software Development Kit must be installed in order to write and compile applications for the Cell Broadband Engine. The official SDK is provided by IBM over their developerWorks website and can be downloaded free of charge [40].

The SDK is a GNU based compiler tool-chain and a set of libraries in order to program applications for the Cell/B.E. processor [19, 41]. When using Fedora, the SDK can be easily installed using the operating system package management tool.

### 3.2.2 Implementation

The sequential version of the MMD template matching algorithm already shows good results for small circuits but can become very slow and inefficient for larger networks especially those with many positive matches. Besides the cost of each replacement, the main reason is the locality of the approach. This means that only one template is matched at a time, leaving the rest of the circuit untouched. The following work describes an approach that tries to solve the aforementioned issue by taking advantage of the Cell Broadband Engines hardware architecture, especially the available SPEs.

#### 3.2.2.1 The parallelization concept

In order to solve the locality problem, one approach to accelerate the algorithm is to perform the following tasks
Figure 3.3: In order to process an input network in parallel it is partitioned into $t$ parts which are processed independently. After successful termination the parts is merged to a single circuit again.

1. partition the circuit $C$ into $t$ parts,

2. minimize these parts individually in parallel

3. merge the eventually minimized parts

However, two important conditions must hold

1. Reversible networks must retain their reversible properties after merging

2. Under the condition that the first assumption is true, merging the independently minimized parts must still compute the same function $f_C = f_{C'}$

The first condition can be easily proofed by considering a reversible circuit of size one. By successively adding more gates, one at a time, the circuits
expands. Remembering the property that each outcome of a gate is a permutation of the input, and the fact that reversible circuits are organized in cascades, the output of gate $G_1$ denotes the input of exactly one successor, a gate $G_2$. Because the output of $G_1$ is a permutation of the its input vector, this permutation of course is the input for gate $G_2$. However, $G_2$ is a reversible gate as well which means that its output again is another permutation of the inputs. Since every input vector is permuted in each stage of the network, this concludes that the basic requirement of a reversible network holds.

It is easy to see that this also applies to the concatenation of whole reversible circuits. Concatenating two reversible networks does not break the rules of reversible logic and again results in a valid reversible network.

In order to prove the second assumption consider a reversible network $G_1G_2G_3...G_s$ with $s$ gates implementing a function $f$. This network can be partitioned into $t$ parts, as illustrated in Figure 3.3. Every part $t_i$ computes a function $f_i$ with $1 \leq i \leq t$.

\[
\begin{align*}
  f_1 &= G_1G_2\cdots G_p \quad (3.3) \\
  f_2 &= G_{p+1}G_{p+2}\cdots G_q \quad (3.4) \\
  &\vdots \quad (3.5) \\
  f_t &= G_{r+1}G_{r+2}\cdots G_s \quad (3.6)
\end{align*}
\]
As already proven, the single parts can be merged. This means the concatenation of the single function results in the total function

\[ f = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_t \]  

(3.7)

After applying the template matching algorithm each part corresponds to a function \( f_i \) and the result is a function \( f'_i \) both with \( 1 \leq i \leq t \). The following observation can be made after the successful termination of the template matching algorithm.

\[ f_i = f'_i \quad \forall \; 1 \leq i \leq t \]  

(3.8)

\( f' \) is the function that is computed after successful termination with

\[ f' = f'_1 \circ f'_2 \circ f'_3 \circ \cdots \circ f'_t \]  

(3.9)

Equation (3.7) and (3.8) can now be concatenated. This leads to the following conclusion, showing that condition two holds.

\[ f = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_t \]  

(3.10)

\[ = f'_1 \circ f'_2 \circ f'_3 \circ \cdots \circ f'_t \]  

(3.11)

\[ = f' \]  

(3.12)

Both conditions hold true which means that a reversible network can be subdivided and computed independently.
3.2.2.2 The PPE implementation

As already explained in Section 2.3, the PowerPC Processing Element is responsible for preparing the actual computational work and distributing it to the Synergistic Processing Elements. Figure 3.4 shows the necessary steps that the PPE has to perform.

![Flowchart](image)

Figure 3.4: The PPE prepares the input circuit for the parallel execution and initializes the SPEs. After all SPEs have finished computation the PPE gathers the computed results and merges the circuit.

The first step is to partition the circuit. Here it is possible to think about different strategies, however for this work, most intuitive one has been chosen. Each chunk has roughly the same size, leaving each SPE a comparable amount of work. Since it is very unlikely that each circuit can be divided...
evenly, it is possible that the last chunk is slightly larger than the other ones. For this project the following strategy has been chosen. Let \( g \) be the total number of gates, \( n \) be the number of parts and \( s_i \) the size of each part with \( 1 \leq i \leq n \). Then the size of each part is

\[
s_i = \begin{cases} 
\left\lfloor \frac{g}{n} \right\rfloor & \text{if } i < n, \\
\left\lfloor \frac{g}{n} \right\rfloor + (g \mod n) & \text{else}
\end{cases}
\] (3.13)

This leaves for the last SPE a maximum of \( g \mod n \) more gates to compute which is computable in a reasonable time.

During the second step, the SPEs are prepared for execution. Each SPE executes the same code on a different set of data that is assigned by the PPE. In order to start the computation, the compiled SPE source code needs to be loaded and a context needs to be set up for each SPE. Then the PPE program needs to prepare the necessary data. In case of the template matching algorithm this information consists of the following information that is referenced in a C struct.

1. number of lines
2. size of the circuit-part (number of gates)
3. start address of the circuit-part
4. start address of the template library buffer
5. start address of the output buffers for minimized circuit-part
6. size of the output buffers (size of minimized circuit-part)

After all data is initialized, the SPEs contexts are ready to be executed. Each SPE context is managed by a POSIX thread running on the PPE. After successful termination of all threads the PPE can retrieve the computed results from main memory and merge them into one reversible circuit again.

3.2.2.3 Implementation on the SPE

Basically, the version running on the Synergistic Processing Elements is structured like the sequential approach. However, in order to get started the SPE needs to be prepared too. As mentioned in Section 3.2.2.2, each SPE is assigned a part of the circuit and other needed data and information such as the templates. This means that as a first step the SPE needs to load this data from main memory into the local store memory. This occurs by invoking DMA calls and loading the needed data directly by using the start address which is assigned to the SPE by the PowerPC Processing Element (Listing 3.1).

```c
1  spu_mfcdma64(&buf, mfc_ea2h(addr), mfc_ea2l(addr),
              size, tag, MFC_GET_CMD);
2  spu_writech(MFC_WrTagMask, 1 << tag);
3  spu_mfcestat(MFC_TAG_UPDATE_ALL);
```

Listing 3.1: A SPE loads data into its local store by issuing a DMA call.
Once all buffers have been created, the actual template matching algorithm can begin. After successful termination each assigned part will be smaller or of equal size. The resulting buffer is then written back to main memory by issuing a DMA call similar to the one shown in Listing 3.1.

### 3.2.3 Results

The parallel implementation has been tested with a variety of reversible circuits of different sizes and properties. All circuit definitions can be downloaded from [33, 42]. Tables 3.1 and 3.2 show the results for two different categories of circuits. The circuits shown in Table 3.1 are interesting because none of them have a regular structure which is why this set of functions is called unstructured reversible functions (urf). These circuits are the largest synthesized circuits that can be downloaded at the aforementioned source. The circuits listed in Table 3.2 have been chosen because these circuits only show little or no positive template matches at all. This means that the computing time spent here only measures the pure matching algorithm. Costly memory operations while replacing a template are non existant. Taking these circuits into consideration shows how expensive an actual replacement in terms of additional computation time is.

All tests have been executed with exactly six SPEs which means that each physically available SPE was responsible for computing one part of the circuit.

While the aforementioned results have been achieved with an one to one
<table>
<thead>
<tr>
<th>function</th>
<th>before</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>before</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>urf1</td>
<td>11554</td>
<td>7225</td>
<td>7233</td>
<td>681.13</td>
<td>128.06</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
<td>3251</td>
<td>286.54</td>
<td>53.15</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
<td>2674</td>
<td>147.10</td>
<td>26.97</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>urf5</td>
<td>10276</td>
<td>5582</td>
<td>5585</td>
<td>642.15</td>
<td>133.39</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>urf6</td>
<td>10740</td>
<td>5455</td>
<td>5488</td>
<td>810.13</td>
<td>149.72</td>
<td>5.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: The runtime and minimization results of the unstructured reversible function (urf) benchmarks show that the acceleration of the algorithm was successful but at the cost of the resulting circuit size.

<table>
<thead>
<tr>
<th>function</th>
<th>before</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>before</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>22.11</td>
<td>3.39</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>25.95</td>
<td>3.94</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>48.27</td>
<td>7.87</td>
<td>6.3</td>
<td></td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
<td>284</td>
<td>13.91</td>
<td>2.13</td>
<td>7.2</td>
<td></td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
<td>637</td>
<td>31.14</td>
<td>4.74</td>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1541</td>
<td>1541</td>
<td>78.91</td>
<td>14.46</td>
<td>5.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Speedup comparison of functions with almost equal minimization behavior

mapping of SPEs and circuit parts, Figure 3.6 shows the result of the computation with an increasing number of parts. It can be seen that a larger number of parts may lead to a better result in terms of less execution time. However, although speed can be improved significantly this approach comes at a cost. It was not possible to reduce the number of gates in comparison to the sequential approach. In almost every case the result is worse which shows that the resulting circuit cannot be reduced as good as with the sequential version. This problem will more likely become larger when increasing the number of parts. This can be explained by the fact that an increasing number
Figure 3.5: Both graphs show the growth behavior of the functions tested in Table 3.1 and 3.2. It can be seen that the parallel version decreases the runtime significantly.

Figure 3.6: An increasing number of threads leads to a better runtime behavior.
of parts results in smaller chunks with more cuts. These cuts, however, are not recognized during template application. Possible template matches are lost here.

3.2.4 Conclusion

The overall results, demonstrated in Section 3.2.3, show that the acceleration of the MMD algorithm was a success. However, the results also show that in some cases speeding up the algorithm comes at a cost. This means the size of the circuit after the template matching using the parallel approach is worse than the sequential results. This, however, is contra-productive, since the algorithm’s purpose is to minimize a given circuit as much as possible. In any case, parallelizing computationally intensive reversible logic synthesis algorithms has many benefits, since the productivity while developing, testing and synthesizing reversible circuits grows. It also offers the possibility to work with circuits of sizes that have not been computable in previously acceptable time.

The Cell Broadband Engine is highly suited for tasks like these and there should be a lot more potential within the hardware in order to further optimize the runtime. As already mentioned, runtime is not the most important factor when it comes to synthesizing the smallest possible circuit but the newly achieved computational power can be used to improve the algorithm in such a way that it produces better results in less time.
The parallel approach attempts to execute a thread on every single gate.

### 3.3 Parallel Approach using NVIDIA Tesla

This work focuses on a different approach to accelerate the MMD algorithm. The main target for this work is a NVIDIA Tesla based GPU architecture where the algorithmic approach considers the massive available parallelism. This version has been implemented with OpenCL that supports testing the implementation on a CPU based architecture too. In the following, the general parallel approach and the results are discussed.

#### 3.3.1 Implementation

The basic idea of this parallel approach is to map each gate of the input circuit to a GPU thread. If a circuit $C$ has $|C| = n$ gates, this will result in $|C|$ threads running in parallel on the GPU. The idea has been illustrated in Figure 3.7. The windows seen represent not only threads but also templates that start matching at the encircled areas.

Instead of programming with NVIDIA’s CUDA programming API, OpenCL has been used. There are various reasons for this choice. First of all OpenCL...
is an open, royalty-free cross-platform standard. This means the implementation of the algorithm is not tied to a specific hardware platform. This improves portability to new or better architectures in the future. Additionally, OpenCL seems to be a future standard. Many companies are interested in its further development. Steps from NVIDIA to provide official OpenCL support, aside from their own technology CUDA (see [29]), or ATI who completely recommend the usage of OpenCL in favor of their own technology Brook (see [28]) show that OpenCL will play an important role in the field of parallel processing in the future. Due to the general structure of the algorithm, the original approach had to be altered. The sequential version always starts matching with the first template on a given position. On a mismatch it tries to match the next template at the same position until all templates have been attempted. From this point it advances to the next position where it starts with the first template again. This parallel version, however, takes a different approach. It tries to match on $|C|$ positions at the same time without changing the template. Only if no match could be found on all $|C|$ positions it advances to the next template. This, however, changes the algorithm’s behavior from template centric to position centric. While the sequential algorithm follows a strict left to right order in terms of template replacements, this approach might consider to replace a template somewhere between position 1 and $|C| - |T_i|$ where $|T_i|$ is the size of template $i$. This shift of emphases can be reviewed in Algorithm 2.

In the first phase all needed data structures are sent to the GPU. This in-
Algorithm 2 Parallel Template Matching

input: $C, T$
output: $C'$

\[ C' = C \]

setup OpenCL

\[ i = 0 \]

while $i < |T|$ do

create buffers

send data to device

execute kernel (parallel template matching)

barrier

\[ M = \text{results from kernel} \]

\[ \text{miss} = \text{true} \]

for $j = 0$ to $|M|$ do

if $m_j \in M$ is match then

replace($m_j, C$)

\[ \text{miss} \leftarrow \text{false} \]

break

end if

end for

if $\text{miss} = \text{true}$ then

\[ i \leftarrow i + 1 \]

end if

end while
cludes the circuit itself and the template that is used for matching. Due to a restriction on the OpenCL side, it is not possible in OpenCL to transfer data structures to the compute devices that contain pointers to potentially a second set of pointers. This restriction is due to the fact that pointers that point to an address on the host become invalid on a device, since the device has a different address space. This, however, means that all memory locations referenced by a pointer must be sent separately. For the actual source code this means that all complex data structures, as they are needed for the template matching, must be decomposed on the client side, and recomposed on the host side. Furthermore, it is not possible to allocate memory dynamically on the device. Calls to the malloc function are not available as well as array declarations with squared brackets with dynamic size (e.g. $a[n]$). Due to the way data structures have been defined in the algorithm and due to its dynamic nature, it is impossible to predict the exact amount of the memory needed. Memory consumption is dependent on several input factors such as the current template size. It is, however, possible to calculate the maximum amount of memory that is going to be used by all buffers during algorithm execution. The problem with this approach is that a lot of unnecessary memory has to be allocated and transmitted. This of course has an effect on the time needed to transfer all buffers to the device. Here it is possible to take advantage of the fact that OpenCL kernels are compiled at runtime. Before compilation it is possible to define constants and to inject them into the source code. These constants are then replaced just before
compilation. This way it is possible to simulate dynamic memory on the GPU.

\[ a[SIZE] \rightarrow a[5] \]  \hspace{1cm} (3.14)

When completed the GPU can start executing the kernel. After successful termination a list of positive and negative matches can be read from GPU memory. The actual replacement happens on client side, since the GPU is only responsible for the matching. In this version of the implementation only one positive match is replaced in order to avoid overlapping matches that may break the circuit. After successful replacement the altered circuit is sent back to the GPU and the matching procedure starts again.

In the following section, the results of the current implementation are discussed.

### 3.3.2 Results

Since this version of the MMD template matching has been implemented with OpenCL, it was possible to test it on a GPU and a multi-core CPU. All test have been executed on an Apple MacBook (Late 2008) with an Intel Core 2 Duo CPU running at 2Ghz. Additionally, the parallel version has been tested on a NVIDIA GeForce 9400M GPU. The CPU consists of two cores which results in two parallel threads, one on each core. The GPU consists of two compute units with eight cores per unit. This results in a total number of
16 concurrent threads. These values have been determined using the device query API provided by OpenCL. Each circuit has been tested three times. The final result is the average over these three samples.
Listing 3.2: The function wallClockTime returns the current time in seconds. The difference between the values stored in start and end is the actual runtime.

For this project, two units of measurement are of interest. The first one is the elapsed time from program start to successful termination and the second one is the resulting number of gates after applying the template matching algorithm. After starting the program, a timestamp is stored as demonstrated in Listing 3.2. Just before exiting the program another timestamp is stored.
Figure 3.8: Figure 3.8(a) shows the values for the parallel version. The GPU is nearly twice as slow as the CPU. Figure 3.8(b) shows the current problem with large circuits with many template matches.

The difference between these two timestamps denotes the total runtime of the application in seconds.

Table 3.3 shows all results, for one sequential and two parallel versions. The used circuits all have special characteristics, such as size or number of replaced templates. The first thing to recognize is that the parallel version produces never produces a better result than the sequential version. The problem with this result, however, is that it makes it difficult to compare the real speed up, because the parallel versions tend to minimize gates less than the sequential one. In order to approximate the speed up, two circuits that cannot be minimized any further have been chosen. When looking at the circuits plus63mod8192 and plus127mod8192 it is observable that the pure matching procedure is way faster in parallel than the sequential version. But when considering a bigger circuit, like urf2 which has many template substi-
## Table 3.3: Comparison between the sequential and the two parallel versions.

<table>
<thead>
<tr>
<th>function</th>
<th>before</th>
<th>after (i)</th>
<th>after (ii)</th>
<th>CPU (i)</th>
<th>CPU (ii)</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>7.609247</td>
<td>0.654159</td>
<td>1.752376</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>14.235284</td>
<td>1.168325</td>
<td>3.008879</td>
</tr>
<tr>
<td>urf1</td>
<td>1517</td>
<td>1487</td>
<td>1488</td>
<td>22.014799</td>
<td>4.421026</td>
<td>8.039400</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
<td>3913</td>
<td>73.778419</td>
<td>221.297445</td>
<td>411.929324</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
<td>2680</td>
<td>41.625869</td>
<td>12.570977</td>
<td>24.662143</td>
</tr>
<tr>
<td>hwb9</td>
<td>1959</td>
<td>1708</td>
<td>1725</td>
<td>30.808588</td>
<td>21.625921</td>
<td>49.436990</td>
</tr>
</tbody>
</table>

(i) sequential version (ii) parallel version

As the sequential version. Generally, the parallel CPU version is almost twice as fast as the GPU version and the parallel CPU version is faster than the sequential CPU version. Only when it comes to many template substitutions the parallel version shows worse results in comparison to the sequential approach. Circuit hwb9 even shows a result with the following relation

\[ t_{CPU_p} < t_{CPU_s} < t_{GPU} \]  \hspace{1cm} (3.15)

where, \( t \) denotes the spent time, \( p \) and \( s \) stand for parallel and sequential.

The results of this parallel approach seem to be unpredictable and dependent on many factors as size and number of replaced gates. While it is hard to analyze and justify the resulting number of gates, it is possible to come up with an explanation for the speed results. Two important reasons are

1. **memory transfers**: when launching the kernel, a lot of data has to be sent back and forth. In order to work properly, the kernel needs
a fresh copy of the circuit after each successful match. Additionally, the templates have to be interchanged, when no further matches can be found for the current one. These memory transfers are significantly slower on the GPU than on the CPU.

2. **compute capability**: when looking at the kernel source code one can see that many loops and conditionals are involved. A full x86 core which is designed for purposes like this can benefit. The GPU cores, however, are not designed to contain such a complex program logic. Of course it is possible, as this implementation shows, but it is far from optimal.

The circuit urf2 is a good example for the aforementioned reasons. During the first iteration 5030 have to be transferred to the compute device. Each gate occupies 16 Bytes which means that transferring 5030 gates results in $5030 \cdot 16 = 80480$ Bytes plus additional data from the struct `circuit_t` (see Section 3.1.1) which is another 16 Bytes. After the first replacement the altered circuit must be resent to the compute device. In other words, each single positive template has a huge cost in terms of memory transfers.

### 3.3.3 Conclusion

Although the MMD algorithm can be accelerated in most cases, the outcome comes at a cost and seems to be dependent on the used hardware architecture. Generally, the GPU shows a better result than the sequential version but this
is not true for every case. On top of this fact, this approach shows a worse minimization result than the sequential version.

The MMD algorithm is an unpredictable algorithm which, in this context, means that it is not possible to predict the number of gates that can be replaced in advance when processing an arbitrary circuit. However, it should be possible to optimize the aforementioned points in order to achieve a lower execution time. Some approaches to achieve this goal are discussed in the following chapters.
Chapter 4

The Cross-Implementations

The following sections discuss two approaches to cross-implement the already described parallel versions. Cross-implementation, in this context, means that parallel approach number one (Section 3.2) is implemented on a GPU and that the parallel GPU version (Section 3.3) is implemented on the Cell Broadband Engine.

The goal of this part of the thesis is to analyze the behavior of two different parallel algorithmic approaches on different hardware architectures. The results will be compared to the original, parallel implementations. The respective analysis of each cross-implementation aims to explain the actual outcome and gives a hint for future versions.
4.1 Cell/B.E. on GPU

4.1.1 Design and Implementation

The first cross-implementation is an approach to port the parallel version discussed in Section 3.2 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.

In direct comparison to the original GPU version the key points are listed in Table 4.1.

<table>
<thead>
<tr>
<th>Original Implementation</th>
<th>Cross-Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>template replacement on host</td>
<td>template replacement on compute device</td>
</tr>
<tr>
<td>only one template at a time on compute device</td>
<td>whole template library on compute device</td>
</tr>
</tbody>
</table>

Table 4.1: Differences between original GPU- and Cross-Implementation

The implementation, again, has been realized with the Open Computing Language. Reasons for choosing OpenCL over the native CUDA approach have been outlined in Section 2.5 and apply for this work as well.

Like the Cell/B.E. version, this cross-implementation consists of three phases. While the first two phases are identical to the aforementioned version, namely loading data and creating the needed basic data structures, the third phase
is a preprocessing step which is needed in order to initialize the GPU and prepare all necessary data. During this preprocessing step, the data structures are altered in such a way that they match the constraints set by the NVIDIA Tesla hardware and OpenCL. The process is a mixture of the steps needed for the Cell/B.E. and the GPU version. First of all the circuit needs to be partitioned. In order to be able to compare the cross-implementation to the Cell/B.E. version, this step uses the same logic to partition the circuit. This means the different parts of the circuit are equal to the chunks in the original implementation for the Cell Broadband Engine. In comparison to the original GPU implementation, the complete minimization process takes place on the GPU. This means no sending back and forth of the altered circuit as in the original GPU version. An implication of this is that only one OpenCL buffer is needed and allocated on the compute device. This step is necessary because in the Cell/B.E. version all work is done on the SPEs. Additionally, the whole template library is transferred to the GPU instead of single templates. Since a template is composed of four scalar values and two pointers to buffers of variable size, the templates have to be decomposed before they can be sent to the GPU. Remembering the explanation in Section 3.3.1, this is a necessary step due to limitations of OpenCL. The result of the decomposition are three one-dimensional buffers and two one-dimensional lookup tables. The lookup tables are used to map indices to positions in the aforementioned buffers that refer to the data which is needed for each template. Listing 4.1 shows the code used to construct a valid template for the
library item with index currentTemplate. The buffer tplBuffer is of type template_t and simply stores a list of templates and their scalar values ng, csize and tsize. The access for this pattern is a simple one-to-one mapping and therefore does not need a lookup table. The buffers cpatBuffer and tpatBuffer are the one-dimensional representation of the buffers valid_cpat and valid_tpat which are needed by each template. Since these buffers are not equally sized, each sub-buffer occupies a different amount of memory. The lookup tables cpatMap and tpatMap have been precomputed on the host in order to map the index currentTemplate to the according sub-buffer.

```
1 struct template_t tpl;
2 tpl.ng = (&tplBuffer[currentTemplate])->{ng};
3 tpl.csize = (&tplBuffer[currentTemplate])->{csize};
4 tpl.tsize = (&tplBuffer[currentTemplate])->{tsize};
5 tpl.valid_cpat = &cpatBuffer[cpatMap[currentTemplate]];
6 tpl.valid_tpat = &tpatBuffer[tpatMap[currentTemplate]];
```

Listing 4.1: Code used to reconstruct a valid template from one-dimensional buffers

The resulting template can then be used for the matching and replacement process.
Another problem that had to be solved for this implementation is related to dynamic memory allocation. In order to replace a match with the matching template, the algorithm needs to allocate and release temporary memory. This, however, is not possible on a GPU, as already discussed in Section 3.3.1. The solution to this problem is to allocate a sufficiently sized buffer on the GPU before launching the kernel (on the host side) and to manage this buffer, which can be treated as a heap, manually. This way it is possible to dynamically store any data during runtime of the kernel. The only data which needs to be stored on the heap during runtime is of type gate\_t which makes the management easier, since the whole heap can be of type gate\_t as well.

The aforementioned buffers already can be used to execute the algorithm on the GPU. In a first attempt, the whole SPE code was identified to be executed on the GPU. This included all steps of Algorithm 1 (see Section 3.1.2). However, this was not possible and led to unpredictable crashes for certain circuits and circuit sizes, however, only on the GPU but not on the CPU. Since OpenCL can be executed on different target platforms it was possible to test the implementation not only on the GPU but also on the CPU. While running the algorithm on the CPU did not cause any problems, execution on the GPU would randomly crash. Extensive debugging sessions could not solve the problem. The lack of real debugging tools showed one big disadvantage of GPU programming at the moment. For instance, it is not possible to stop the execution of a GPU algorithm with a tool like the
GNU Project Debugger (gdb), a commonly used tool capable of stopping an application during runtime and allowing inspection of several properties like memory, buffers or stack frames [43]. On the GPU it is not even possible to utilize functions of the stdio library which means as a consequence that no IO, for example via printf, is available. All debugging that can be done at the moment has to be performed by uncommenting code and observing if the program still crashes. This process was very cumbersome and took several weeks without finding a solution to the problem. The only solution that worked in the end was to externalize some parts of the code to the host side which means executing it on the CPU. An implication of this is that the cross-implementation is no real one-to-one mapping of the SPE version. However, this solution made it possible to finish the implementation at all.

Algorithm 3 Cross-Implementation Main Loop

1: create buffers
2: send necessary data to GPU
3: set kernel arguments
4: loop
5: execute kernel
6: wait for completion
7: for all circuit parts do
8: if all parts are finished then
9: break LOOP
10: end if
11: end for
12: end loop
13: load minimized parts from compute device

Algorithm 3 shows the new host side code used to compute the main loop. In
each iteration of the outer loop, the kernel is executed and the host waits until all computation is done. Then it fetches intermediate results for every part of the circuit from the compute device and checks if all parts have finished minimizing. If this is not the case, the algorithm starts the kernel again with new parameters, like the offset position that has already been discussed in Section 3.1.2. If all threads have successfully minimized the circuit, the algorithm leaves the outer loop and is ready to load the minimized parts from the compute device.

4.1.2 Results

To compare the outcome of this implementation, the benchmarks performed match exactly the ones in [12]. This means that the MMD template matching has been applied to the same circuits.

<table>
<thead>
<tr>
<th>function</th>
<th>before</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>GPU</th>
<th>PPE</th>
<th>6 SPEs</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>surf1</td>
<td>11554</td>
<td>7225</td>
<td>7233</td>
<td>7233</td>
<td>681.13</td>
<td>128.06</td>
<td>1301.42</td>
</tr>
<tr>
<td>surf2</td>
<td>5030</td>
<td>3250</td>
<td>3251</td>
<td>3251</td>
<td>286.54</td>
<td>53.15</td>
<td>528.46</td>
</tr>
<tr>
<td>surf3</td>
<td>2732</td>
<td>2674</td>
<td>2674</td>
<td>2674</td>
<td>147.10</td>
<td>26.97</td>
<td>512.01</td>
</tr>
<tr>
<td>surf5</td>
<td>10276</td>
<td>5582</td>
<td>5585</td>
<td>5585</td>
<td>642.15</td>
<td>133.39</td>
<td>93.77</td>
</tr>
<tr>
<td>surf6</td>
<td>10740</td>
<td>5455</td>
<td>5488</td>
<td>5488</td>
<td>810.13</td>
<td>149.72</td>
<td>1235.95</td>
</tr>
</tbody>
</table>

Table 4.2: Cross-implementation results of the URF functions on GPU in comparison to the original results on a Cell/B.E.

In Table 4.2 two things are observed. The first to observe is the resulting number of gates after the application of the template matching algorithm. Columns four and five show that the cross-implementation produces the same
Table 4.3: Cross-implementation results running on a GPU showing functions with no or few template matches

<table>
<thead>
<tr>
<th>function</th>
<th>Number of gates</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
<td>PPE</td>
</tr>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1541</td>
</tr>
</tbody>
</table>

result as the original Cell/B.E. version. This measurement is important since it is an indicator for the correctness of the implementation itself. The more interesting result, however, can be seen in columns six, seven and eight. The last of the three columns shows the elapsed time of the GPU-based cross-implementation in comparison to the Cell/B.E. version. Here it is clearly visible that this version does not perform as good as the original parallel approach. In some cases it is even slower than the sequential PPE version which is interesting and needs further analysis.

Following Table 3.2 in Section 3.2.3, Table 4.3 shows the results of the GPU-based version for the same circuits used in the original Cell/B.E. version. The table structure is similar to the structure used for Table 4.2 and shows the correctness of the cross-implementation by comparing the resulting number of gates with the sequential, the SPE and the GPU version. These circuits have, again, been chosen because almost no templates matched during processing. As traceable in columns two to five, this is the case. While columns three and four are borrowed from Table 3.2 for better comparability, col-
umn five shows the result of the GPU-based version. The last three columns show the runtime of each version in direct comparison. Again, the cross-implementation is slower than the original Cell/B.E.- and even slower than the sequential version.

All benchmarks basically show the same result. While the minimization result of the algorithm is the same as the original implementation, no speed up could be achieved. In fact, the opposite is the case. All benchmarks are significantly slower than the parallel counterpart of the Cell/B.E. version. Later, these results will be further analyzed in order to find answers to the question why the cross-implementation is so much slower (see Section 4.3).

4.2 GPU on Cell/B.E.

4.2.1 Design and Implementation

This cross-implementation takes the original approach explained in Section 3.3 and ports it to the Cell Broadband Engine. Like the first cross-implementation, described earlier, again, the goal is to examine if a different parallel approach runs well on this kind of hardware architecture and if it might have any benefits over the other combinations of implementations and hardware architectures examined so far.

In a first step it is necessary to look how the actual hardware capabilities of the Cell/B.E. can be mapped to the algorithmic approach which is designed to apply as many templates in parallel as possible. For the Cell Broadband
Engine this means that each Synergistic Processing Element is responsible for performing exactly one template match. The responsibility of choosing the templates and executing the main loop is externalized to the PPE which is now in charge of more managing tasks.

Firstly, the PPE chooses a starting template and a starting position for the algorithm to begin the actual matching. Secondly, a SPE context and a POSIX thread are created for each of the six SPEs involved. This means there are always six POSIX threads running concurrently on the PPE while matching. After loading the binary that is going to be executed on the SPEs, all necessary data structures are prepared before execution. This procedure is similar to the one performed in the original implementation. Again, a C programming language structure is used as a communication interface between PPE and SPEs. For example, this structure carries information about the starting address of the circuit and the template library but also tells the SPE which template to execute and in which buffer to write the results. Each SPE has a unique identification number (or thread id) which maps every SPE to exactly one position within the reversible circuit. On this position the SPE then performs the template match. Since always six SPEs are involved, the actual execution of the matching procedure shifts in a non-overlapping sliding window of size six over the circuit. Non-overlapping in this context means that the window always jumps six positions to the right in each iteration.

In the next step, the POSIX threads are executed which means that the
SPEs start matching a template against the circuit. On the SPE, several DMA transfers are responsible for loading all necessary data structures into the local store memory. These transfers involve loading the circuit and creating the template library. The actual template application does not differ very much from the previous implementations. The biggest difference in comparison to the original Cell/B.E. version is that no template matching happens on the SPE itself. All necessary information for a template match are stored in prepared buffers and sent to the main memory over the Element Interconnection Bus. When passing back the control to the PPE, it can then directly access these buffers and start working with the data.

When all SPEs successfully terminated execution the PPE is back in control for all necessary and further steps. In a first step, the PPE checks the result buffer filled by the SPEs during computation for positive matches. For this it iterates over the buffer and checks a flag which tells the PPE if a match at a given position $x$ was positive. This result buffer is always a one-to-one mapping of each SPE. This means each SPE has a dedicated memory in which it can write its results. If no positive match could be found in the buffer, the aforementioned sliding window shifts six positions to the right and starts the computation from the beginning. However, if there is a positive match, the PPE replaces the template at the position on which the match was found. After a successful replacement, the algorithm keeps on matching with the current template until no further matches can be found. Like the OpenCL version, the algorithm stops when all templates have been matched
against all positions in the circuit.

4.2.2 Results

Again, in order to measure the outcome of this cross-implementation and for better comparability, the exact same circuits as used in Section 3.3.2 have been used.

<table>
<thead>
<tr>
<th>function</th>
<th>Number of gates</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
<td>after GPU</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
</tr>
<tr>
<td>urf1</td>
<td>1517</td>
<td>1488</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3913</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2680</td>
</tr>
<tr>
<td>hwb9</td>
<td>1959</td>
<td>1725</td>
</tr>
</tbody>
</table>

Table 4.4: Cell/B.E. cross-implementation results in comparison to the sequential and the original OpenCL version

Table 4.4 shows the results of the original implementation executed on a NVIDIA Tesla based GPU and the Cell Broadband Engine. Columns three, four and five show the number of gates of the circuit, whereas the two latter show the results after the minimization process. The last two columns are a direct runtime comparison in seconds between the GPU and the Cell Broadband Engine.

Like the first cross-implementation (Section 4.1), this one shows the exact same results as the original GPU implementation. Again, this is a proof for the correctness of the implementation but the actual problem remains.
The parallel version never shows a better result than the sequential versions (compare with Tables 3.1 and 3.3).

When examining the last two columns, it is obvious to see that the cross-implementation shows very bad results compared to the OpenCL based implementation that is executed on a GPU. The execution time of the cross-implementation is between two and twelve times slower than its original counterpart. As in Section 4.1, a following analysis tries to identify the reasons for the bad outcome and to give answers for following steps.

4.3 Analysis

The results discussed in Sections 4.1.2 and 4.2.2 show that both cross implementations cannot compete with the results that have been achieved with the first parallel implementations. This, of course, leads to the question “Why?” and needs further analysis. The following paragraphs will try to answer this question and give an outlook for future development.

The main reason for the bad results shown by the cross-implementations can be explained with a structural divergence between the cross-implemented parallel approaches in combination with the used hardware. Structural divergence, in this context, means that general hardware structure and algorithmic approach do not match. A first assumption in this context is that the algorithmic approaches cannot be interchanged between different hardware architectures and that structure matters.
The general hardware structure of a NVIDIA Tesla based GPU is defined by its many, lightweight streaming processors that are the primary thread processors in the streaming multiprocessor. These streaming processors perform all fundamental operations and execute the actual computation. Because GPUs provide many of these cores and can handle hundreds of threads simultaneously, this means that a GPU’s processing capability is highly dependent on the level of parallel granularity. The more cores, or respectively streaming processors, can work in parallel on a fine grained set of independent data, distributed over all computing resources, the better the performance will be [24]. The reason for this is that a GPU can hide the limited functionality of each core, for example compared to a x86 CPU, by executing as many threads in parallel as possible.

While on the one hand there is the NVIDIA Tesla architecture that needs fine grained parallelism in order to achieve high performance results, on the other hand is a parallel approach that does not take this into account. In this case the algorithm that has been implemented on the Cell/B.E. first, does not consider fine grained parallelism. Instead, it focusses on coarse grained data that is processed by only a few cores. The GPU used for benchmarking in this thesis offers one streaming multiprocessor which means that eight cores are available and up to 768 threads can run in parallel [24]. All benchmarks, however, have been performed with only six threads in order to be able to compare the results to the six threads used on the Cell/B.E. (one thread per available SPE). The result of this approach, of course, is that much of
the available parallel potential of the GPU is wasted. Even if more threads would be scheduled on the GPU, the minimization result would only become worse. The reason for this, of course, is that the more parts a circuit has the less gates are available for template replacements. Many parts lead to a high level of fraction, which in turn leads to small search spaces and many cuts. Obviously, the result of this is the loss of potential template replacements across two parts and therefore a poorer minimization result.

A similar structural problem exists on the Cell Broadband Engine and the second cross-implemented parallel approach. The hardware structure of the Cell/B.E. is designed in such a way that a few cores can process a big amount of data in parallel. When remembering the ultimate goals of the whole Cell Broadband Engine Architecture one of them was on the one hand to have one general purpose processor which is good at running control-intensive code but not very performant on processing huge amounts of data. On the other hand there are the Synergistic Processor Elements that are optimized to process large data sets. Additionally, the Element Interconnection Bus allows to transfer large amounts data between PPE and SPEs very efficiently via Direct Memory Access in order to hide memory latency [41]. This means that coarsely grained data sets that leave big portions to work for each SPE works best with this kind of structure.

The parallel approach, however, focusses on many small problems. When looking at the actual implementation that has been done for this thesis, one can see that relatively small chunks of data are processed. In particular, for
each of the six available SPEs, this means that only 20 gates are processed. Of course, processing 20 gates is fairly fast and not very computationally intensive, however, the Cell Broadband Engine has to switch contexts very often. This means that control switches back and forth between PPE and SPEs rapidly. This involves setting up new compute contexts, loading the SPE binary, preparing the data that needs to be transferred, setting up the POSIX threads which control each SPE and gathering the results after the processing is complete. While this step only has to be performed once for the first parallel approach, it has to be performed an unpredictable number of times for the second. Additionally, the PPE needs to process and analyze the result set and as already mentioned, it is not optimized for tasks like this. The reason why this implementation shows bad results is a sum of many small factors that together accumulate to a substantial cost.

4.4 Conclusion

Up to this point four implementations on two different hardware architectures have been described, benchmarked and analyzed in order to optimize the template matching step of the MMD algorithm. While the first iteration of the work done so far (Chapter 3) was mostly successful, for the second iteration (Chapter 4) no positive outcome or benefits could be achieved. Both cross-implementations show the exact same result in terms of minimized circuit size. On the one hand this is a sign of algorithmic correctness
in comparison to the original approaches but on the other hand the same problems remain. The parallelization of the template matching comes at the cost of bigger circuits.

The far worse result, however, is the loss of computational performance. The analyses in Section 4.3 shows, the theoretical algorithmic approaches are not made to be interchangeable between different hardware architectures. The actual computing time is, in some cases, worse than the sequential approach which is not the desired outcome when parallelizing an algorithm.

Since both cross-implementations do not have any benefits over the sequential or original parallel approaches, these implementations can be abandoned. The focus of the following work now is to further analyze the existing parallel approaches, to discover their weaknesses and to optimize those in order to improve the overall results.
Chapter 5

Implementation Improvements

The two cross-implementations described in Chapter 4 showed that parallel architectures and algorithms are not always faster as or superior to sequential approaches of the same problem. Any improvement in results are experienced due to the correct algorithm implemented on the matching hardware architecture.

The original parallel implementations in Chapter 3 delivered better results which is why all further efforts to improve the algorithms is directed into the proceeding versions. The following Chapter describes the improvements that have been implemented and delivers some results that show if these improvements were successful.
5.1 Cell/B.E. Improvement I: Optimizing Minimization Results

5.1.1 Motivation

The results in Tables 3.1 and 3.2 showed that some circuits could not be minimized as good as the sequential version managed to achieve. However, since the main purpose of the MMD template matching algorithm is to minimize a given circuit as small as possible the achieved results are not desirable.

In the following, an approach is described that tries to fix that problem and implements a version which delivers better results than the original version.

5.1.2 Design and Implementation

Before being able to improve the aforementioned problem it must be clear what the actual issue is and why the parallel version delivers worse results.

As already shortly mentioned in [12], the problem can be explained due to the fact that several, completely independent parts are processed and in the end are merged back together. Since a template shifts across a circuit from left to right, it might be possible that a potential template match could be located across the border of two independent parts of the network.

One potential solution to this problem, which also has been implemented as the first improvement in this work, is to introduce a concept based on a window which is defined by a set of gates that are matched again. It is
obvious that a window lies on top of a former cut and therefore is composed of gates of two different parts. This makes it possible to re-run the template matching algorithm and to consider those gates that could not be taken into account before.

Let a circuit $C$ be partitioned into $n$ parts. Every circuit then has $n - 1$ cuts which implies that also $n - 1$ windows have to be created and processed. Apart from additional computational overhead this window based approach has no further disadvantages over the original approach. For this let $C'$ be the circuit after the first pass and $C''$ the circuit after the second pass. After the template application the following relation will always be satisfied.

$$C'' \leq C' \leq C$$

(5.1)

This relation is easy to proof, since $C'$ and $C''$ are direct dependents of $C$. If no successful template matches could be found during a first pass, $C' = C$ holds and since a template match never adds gates to a circuit $C' > C$ never can happen which means that $C' \leq C$ holds. The same argumentation can be used for the second pass so that relation 5.1 holds true.

One important choice that has to be made is the actual size of the window. This means how much space it should occupy before and after the cut. As already mentioned, it is clear to see that the window has to start before and has to to end behind the cut. On the one hand the window has to be large enough in order to cover enough gates on each side, so that the probability of
a potential match is sufficiently high. On the other hand the window should not be too large because this would cause another unnecessary computational overhead. The way the algorithm tries to match gates against a template is to perform a look-ahead for potential matching gates. Look-ahead means that the algorithm takes advantage of the fact that gates can be interchanged if this change does not affect the function definition. Details of this property have been described in Section 2.2.3 and in the works of Maslov et. al [16, 1, 18]. As already described in the aforementioned works, the actual implementation uses a heuristic for a maximum look-ahead. This means that the algorithm stops trying to interchange gates after an empirical determined number $n$. The reason for a restricted look-ahead is that the bigger a look-ahead is, the less probable it is to find gates that can be interchanged. Since the template matching algorithm, like the synthesis, is bidirectional, a good choice for a window size is $2n$, where all gates are equally distributed to the left and right of the cut. Since the Cell Broadband Engine can only transfer data that is a multiple of 16 [20], the window size has to be expanded if

\[ 2n \mod 16 \neq 0 \quad (5.2) \]

holds. The following function defines a sufficiently sized window for a look-
ahead parameter $n$

$$f(n) = \begin{cases} 
2 \cdot (n + (n \mod 16)) & \text{if } 2n \mod 16 \neq 0, \\
2n & \text{else}
\end{cases} \quad (5.3)$$

The current implementation uses $n = 20$ because this number showed the best relation between the minimization result and computational effort. Each window then covers $f(20) = 48$ gates, 24 on each side of the cut.

One important restriction is that two windows are not allowed to overlap. This must be considered when determining the number of cuts and for smaller circuits. This is necessary because overlapping windows might interfere with potential positive matches. Consider two windows $W_1$ and $W_2$ as sets of gates with $W_{W_1 \cap W_2} = W_1 \cap W_2 \neq \emptyset$. Assuming two positive template matches, one in $W_1$ and the other in $W_2$ and both involving the same gate $g \in W_{W_1 \cap W_2}$, several scenarios are possible.

1. According to the aforementioned interchangeability rule, $g$ is moved to a different position within each window, either with $g \in W_{W_1 \cap W_2}$ or $g \notin W_{W_1 \cap W_2}$. When merging the windows after processing, the algorithm inserts one gate at two different positions which theoretically is not possible. Practically, each window has its own copy of this gate $g$ which means that it is duplicated and therefore bloats the circuit. This, however, is a violation to the rule that a template match never bloats a circuit.
2. Gate $g$ is removed by a template. Theoretically, this implies that one single gate $g$ is removed twice from $W_{W_1 \cap W_2}$. But assuming it is removed from $W_1$ firstly, creating a new set $W'_{W_1 \cap W_2} = W_{W_1 \cap W_2} - g$, $W_2$ could not perform a valid template match since $g \notin W_{W_1 \cap W_2}$ anymore. Practically, this leaves a circuit $C'$ with $C' \leq C$ which does not violate the rule mentioned in Enumeration 1 but may break the function definition.

This issue, however, only becomes a problem for smaller circuits or circuits with too many cuts. Since the focus of the parallel implementation lies on large circuits this problem should occur rarely.

From a programming point of view, one benefit of this approach is that no additional binary for the SPE is necessary. Like a partition, each window itself can be seen as a chunk from the circuit and the algorithm performs the exact same task. The important part is to redefine the key values for the structure which serves as a communication interface between PPE and SPE (see Section 3.2). These key values are the window size which is now just the constant 48 and the starting address of the first gate of each window. This can be obtained with the code shown in Listing 5.1.
Listing 5.1: Obtaining the starting address of the first gate of each window.

The buffer gates contains all gates, and therefore defines the actual circuit.
The buffer new_slices stores the position of each cut after the first template application. This lookup table can be obtained when merging back the parts after the first pass of the algorithm.

5.1.3 Results

In order to be able to compare the results of this improvement and to be consistent with the previous discussions of Cell/B.E. related results, the same set of reversible functions has been tested. Reasons why these benchmark functions were chosen have been discussed earlier (Section 3.2.3) and still apply.

Table 5.1 shows all of these results. The general table structure is divided into three parts. The first part and column just shows the benchmarked function. The actual parts of interest are part two which shows the minimization results and part three which shows the runtime of the algorithm.

The second part contains four columns. The first column shows the original size of the function before the template application. The second column
Table 5.1: The runtime and minimization results of the window based approach show that the minimization result could be optimized with a small time penalty.

<table>
<thead>
<tr>
<th>function</th>
<th>orig</th>
<th>seq</th>
<th>1st pass</th>
<th>2nd pass</th>
<th>1st pass</th>
<th>2nd pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>3.40</td>
<td>5.41</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>3.97</td>
<td>6.04</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>7.89</td>
<td>9.96</td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
<td>284</td>
<td>282</td>
<td>2.12</td>
<td>3.98</td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
<td>637</td>
<td>637</td>
<td>4.92</td>
<td>6.80</td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1541</td>
<td>1541</td>
<td>1541</td>
<td>13.23</td>
<td>15.34</td>
</tr>
<tr>
<td>urf1</td>
<td>11554</td>
<td>7225</td>
<td>7233</td>
<td>7225</td>
<td>128.90</td>
<td>130.85</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
<td>3251</td>
<td>3250</td>
<td>53.11</td>
<td>54.99</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
<td>2674</td>
<td>2674</td>
<td>27.46</td>
<td>29.47</td>
</tr>
<tr>
<td>urf5</td>
<td>10276</td>
<td>5582</td>
<td>5585</td>
<td>5579</td>
<td>133.19</td>
<td>135.16</td>
</tr>
<tr>
<td>urf6</td>
<td>10740</td>
<td>5455</td>
<td>5488</td>
<td>5455</td>
<td>149.07</td>
<td>151.71</td>
</tr>
</tbody>
</table>

shows the result produced by the sequential version. Column three shows the result produced by the first parallel version. The last column, finally, shows the outcome of the window-based approach. The third part of the table contains two columns. The first column shows the runtime of the first parallel implementation while the second shows the runtime after the window-based approach. For both parts, the comparison between first parallel version and window-based is labeled as 1st pass and 2nd pass since the window-based approach is dependent on the first parallel implementation and runs in a second pass right after it.

When looking at columns three to five it is clearly visible that the second run, using the window-based approach shows an improvement to the first parallel version. While almost every minimization result of the first parallel
implementation shows worse results than the sequential version, this defect could be resolved in the latest version.

A look into the last two columns shows that the window-based improvement leads to a time penalty with an average of 2.04 seconds. However, since the window is very small compared to an individual circuit part, this longer runtime is almost negligible. Generally, processing the 2\textsuperscript{nd} pass is fast since in the small search space defined by the window does not contain many template matches. Usually, this search space only contains those template matches that have been missed in the 1\textsuperscript{st} pass due to the partitioning of the circuit.

5.2 Cell/B.E. Improvement II: Vectorizing Execution by using SIMD

5.2.1 Motivation

Until this point, the work being performed on the SPE is strictly sequential and only operates with scalar values. This means that each instruction works with exactly one piece of data at a time. While this approach delivers fast and first and foremost valid results, it does not take the real capabilities of the SPE into account. One of the SPEs strengths is the execution using Single Instruction Multiple Data (SIMD) which allows the SPE to execute several pieces of data in only one single instruction [35]. A transition from
scalar to SIMD-based code could help to further accelerate the application.

### 5.2.2 Single Instruction Multiple Data on the SPE

As the name already implies, Single Instruction Multiple Data can work on multiple pieces of data in one single instruction. It is obvious to see the advantages of this approach. Depending on the amount of data that can be processed at the same time, let's assume $n$ elements with $n \in \mathbb{N}$ and $n > 1$, a theoretical speedup of $n$ could be achieved.

When writing a SIMD-based application, all data must be laid out in vectors which is why SIMD code is often called vectorized code. On the SPE, these vectors can have a varying number of elements depending on the data type that is going to be processed. Generally, the SPE distinguishes between the five different data types listed in the following enumeration.

- byte - 8 bits
- halfword - 16 bits
- word - 32 bits
- doubleword - 64 bits
- quadword - 128 bits

Figure 5.1 illustrates this approach by showing a SPE register and how these data types fit in. All of the 128 available registers on the SPE are 128 bits...
wide and can store between 16 8 bit values (byte) and one single 128 bit value (quadword).

Figure 22. Register layout of data types and preferred (scalar) slot

Chapter 3. Programming the SPEs 61

Figure 5.1: Different data types used by the SPE, their register layout and preferred scalar slot [7].

This means when vectorizing program code, a register for example can process 16 values of type char or four values of type int in one instruction [7]. An interesting fact about the Synergistic Processing Unit is that it does not include separate registers for scalar values and operations. When programming without implicit use of SIMD, all scalar values are stored in the SIMD registers too. For each data type the register has a preferred scalar slot in which the scalar value is stored. The grey highlighted register elements in Figure 5.1 show this preferred slot. For example a char is preferably located in Byte 3 of the register and an integer occupies bytes 0 to 3.

The general way a SIMD program works is that data buffers have to be mapped to the aforementioned vectors. Consider two data buffers \( a \) and \( b \), each of size 16, storing integer values. Each element \( a_i \) of buffer \( a \) shall be
added to element $b_i$ of buffer $b$. A scalar program would iterate over each element and instruct the operation, one at a time.

```c
1 int a[16], b[16], c[16];
2 // fill buffers a and b with data
3
4 for (int i = 0; i < 16; i++)
5 {
6   c[i] = a[i] + b[i];
7 }
```

Listing 5.2: Version working with scalar values.

As already mentioned, a vector of integers can operate on four values at the same time. This means the aforementioned data buffers can be stored in eight vectors, four for each buffer. Now it is possible to work with these buffers by issuing SIMD operations.

```c
1 int a[16], b[16], c[16];
2 // fill buffers a and b with data
3
4 // create vectors
5 vector int* va[4] = (vector int*)a;
6 vector int* vb[4] = (vector int*)b;
```
Listing 5.3: Version working with SIMD instruction

As can be seen in Listing 5.3 the code is optimized for SIMD execution. The loop only iterates 4 instead of 16 times and the spu_add intrinsic which is provided by the IBM SDK, issues the execution of the operation. The result is the same as in Listing 5.2 with the difference that only four instructions have been used.

This simple example illustrates the efficiency of Single Instruction Multiple Data. There are, however, several disadvantages which make utilizing SIMD no easy task.

1. The data processed must be independent: SIMD is data parallel processing and as a general rule of parallel programming dictates, no dependencies between the processed data is allowed. Whenever data independence cannot be guaranteed, SIMD processing is not possible. This results in a mix of scalar and vectorized code and a performance loss.

2. No selective execution: For example, when laying out data buffers for
SIMD execution, it is not possible to only instruct three of four elements of a vector to be executed. The all or nothing principle is valid. Either all elements of the vector (for example four) are instructed or none. This is a particular problem for data that is not a multiple of 2, 4, 8 or 16 and does not fit perfectly into the corresponding vector.

3. Not processable patterns: All elements of a vector must process the same operation. It is not possible to define an individual operation per element.

4. Out of order memory access and conditional branches: SIMD instruction works perfectly well for use-cases as demonstrated in Listing 5.3 that guarantee unified access to memory. It becomes a lot more complex when special memory access patterns (for example non-linear buffer indices) or conditional branches are involved. Since it is still possible to use SIMD in these cases it makes programming a lot more difficult and complex.

5. Dynamic Memory Allocation: SIMD works best with fixed buffer sizes that can be perfectly aligned for SIMD access. Buffers of varying sizes or dynamically created buffers at runtime can complicate programming. In a worst case scenario preparing all buffers for SIMD may result in larger memory consumption (memory is a rare resource on SPEs) and might have negative effects on performance (no performance gain or even performance loss).
The implementation described in the following section attempts to identify functions and areas that can be vectorized and used for further speed improvements.

5.2.3 Design and Implementation

There are two possibilities for a potential SIMD based version. The first and more time consuming one is to completely rewrite the current implementation with the purpose of vector based instructions in mind. The second one is to analyze the current implementation and to identify SIMD compatible code regions. The second version then will result in a mixture of scalar and vectorized code. For this work, the second variant has been chosen. This means that in a first step, SIMD compatible code must identified. The enumeration in Section 5.2.2 briefly shows some of the criteria that have to be considered when trying to identify the aforementioned code regions. In the following, the process of converting a piece of code from scalar to vector-based is presented.

```c
1 void template_extract_pattern(struct template_t* tpl,
                                struct circpat_t* tempCircPat, bool dir, int offset, int len)
2 {
3     ...
4     for (int i = 0; i < tpl->csize; i++)
5         {
```
tempCircPat->cpat[i] = dir ? extract (tpl->valid_cpat[i].cpattern, offset, len) :
extract (tpl->valid_cpat[i].rev_cpattern, offset, len);

Listing 5.4: Scalar version of the function template_extract.

Listing 5.4 shows a small excerpt of a function that is used to extract a pattern from a given template definition depending on the current direction of the algorithm (left to right or right to left). In this case for each control line an integer value is stored in the buffer tempCircPat->cpat for an index i. This example exclusively works with scalar values which means that on each instruction exactly one value is calculated. The for-loop iterates tpl->csize times, and on each iteration the function extract returns a single value.

Listing 5.5 shows the completely vectorized version of the code shown earlier in Listing 5.4. In order to vectorize this piece of code several steps have to be performed. First of all, the buffers have to be expanded. Since vectors operate on fixed sizes, the size of a buffer should be a multiple of its vector. In this particular example, integer values are calculated. Remembering the Cell/B.E. registers illustrated in Figure 5.1, this means when operating with integers the vector size is four. The function alloc_size4vec with the second
argument set to 4, expands the buffer, if necessary, to a size which is a multiple of four. The conversion of a regular buffer to a vector is a straightforward task on the Cell/B.E.. Since both data structures share the same memory layout, it is sufficient to cast the scalar buffer to a buffer of vectors as shown in line 9.

```c
void simd_template_extract_pattern(struct template_t* tpl, struct circpat_t* tempCircPat, bool dir, int offset, int len)
{
  // expand tpl->csize to multiple of 4, so it fits in vec4
  size_t csize4vec = alloc_size4vec(tpl->csize, 4);

  tempCircPat->cpat =
    malloc(csize4vec * sizeof(unsigned int));

  vector unsigned int* vcirccpat =
    (vector unsigned int*)tempCircPat->cpat;

  vector unsigned int vdir = (vector unsigned int)
    spu_cmpeq(
```

103
spu_splats((unsigned char)1), spu_splats((
unsigned char)dir));

vector unsigned int zeros = spu_splats(0u);

for (int i = 0; i < csize4vec >> 2; i++)
{
  const unsigned int stride = i * (csize4vec >> 2);
  vector unsigned int tplcpat = (vector
      unsigned int) {
    tpl->valid_cpat[stride].cpattern,
    tpl->valid_cpat[stride + 1].cpattern,
    tpl->valid_cpat[stride + 2].cpattern,
    tpl->valid_cpat[stride + 3].cpattern
  };

  vector unsigned int tplrev_cpat = (vector
      unsigned int) {
    tpl->valid_cpat[stride].rev_cpattern,
    tpl->valid_cpat[stride + 1].rev_cpattern,
    tpl->valid_cpat[stride + 2].rev_cpattern,
    tpl->valid_cpat[stride + 3].rev_cpattern
  };

Listing 5.5: Vectorized version of the function template_extract

One particular problem in SIMD applications are branches. The variable dir which is used to indicate the direction of the algorithm has to be converted to a vector. In the scalar version this variable is used for branching. The conventional branching instruction (keyword: if), however, cannot be used for executing vectorized code. This is why special SIMD instructions have to be used. One advantage of using these special instructions is that costly branches are completely eliminated. The downside is that the code becomes more complex and is harder to maintain in the future. The way branching in vectorized programs works is to calculate a mask that stores bitwise ones and
zeros for each element of the vector, representing true or false. The Cell/B.E. SDK knows a couple of low level instructions and high level intrinsics for languages like C that create these masks. An example for such an intrinsic is spu_cmpeq which checks each element of two input vectors $a$ and $b$ for equality and stores the result in an output vector $c$. For the aforementioned variable $dir$ this means that a vector $vdir$ either consisting of zeros or ones is created. The intrinsic spu_splats is used in order to create an vector with each element set to a given input value. One further characteristic of vectorized code is that loops do not have to iterate as often as loops that iterate over scalar code. In this example the loop only needs to iterate

$$\frac{cs\text{size} vec}{|vcirccpat|} = \frac{cs\text{size} vec}{4}$$

(5.4)
times, since four elements can be processed at the same time. The intrinsic spu_sel performs a bitwise selection on two input vectors $a$ and $b$ depending on a pattern which is specified in an input vector $c$. For each bit in vector $c$ set to 0, the corresponding bit of vector $a$ is selected and set in an output vector $d$. Otherwise, the corresponding bit of vector $b$ is selected and set in $d$. The variable $vdir$ serves as a pattern since it either contains only zeros or ones. In a first step, spu_sel checks if $vdir$ is equal to 1. If this is the case the result of simd_extract is stored in $vcirccpat[i]$, otherwise a vector only containing zeros is returned. The next line performs the same action but this time it checks if $vdir$ is set to 0. If this is the case, again, the result of
*simd_extract* in *vceirccpat[i]* but this time with a different first argument. If *vdir* is not equal to 0, the result of the line above will be returned.

The function *simd_extract*, of course, has been vectorized. Before this it only accepts scalar arguments and returns a single, scalar integer value. Now, it accepts a vector as first argument and returns a vector. The vectorized input values are created inside the loop body (lines 19 - 32 in Listing 5.5). The comments partly show the old operations.
vector unsigned int simd_extract(vector unsigned int* vnum, int offset, int len) {
  // num >> offset
  vector unsigned int temp = my_sr_uint(vnum, offset);
  vector unsigned int mask = spu_splats(0u);
  vector unsigned int ones = spu_splats(1u);
  for (int i = 0; i < len; i++) {
    // mask = mask << 1 + 1
    mask = spu_add(spu_sl(mask, ones), ones);
  }
  // temp & mask
  return spu_and(temp, mask);
}

Listing 5.6: Vectorized function extract.

Unfortunately, the used Cell/B.E. SDK version did not come with a vectorized right shift operator. The solution to this problem was to devise a custom implementation shown in Listing 5.7.
Listing 5.7: Custom SIMD right shift operator.

5.2.4 Results

The previous section discussed an additional approach to accelerate the MMD algorithm. While the first improvement focussed on an algorithmic approach, the second one attempts to take advantage of some hardware capabilities provided by the Cell Broadband Engine, or more specifically the Synergistic Processor Elements.

Table 5.2 shows the results of the aforementioned improvement. In order to benchmark the improvement, the already known and discussed functions have been used. It has the same structure as the previously used Table 5.1
in Section 5.1.3. This means it shows a direct comparison between the sequential and the improved version.

<table>
<thead>
<tr>
<th>function</th>
<th>Number of gates</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>seq</td>
</tr>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1541</td>
</tr>
<tr>
<td>urf1</td>
<td>11554</td>
<td>7225</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
</tr>
<tr>
<td>urf5</td>
<td>10276</td>
<td>5582</td>
</tr>
<tr>
<td>urf6</td>
<td>10740</td>
<td>5455</td>
</tr>
</tbody>
</table>

Table 5.2: Results of vectorized Cell/B.E. implementation. The runtime could be reduced in comparison to the previous version.

The results show that the use of the SIMD have an impact on the algorithm performance. Larger functions with many matched templates benefit greatly from the vectorization. In these cases the Cell/B.E. shows the real benefits of its architecture in processing large amounts of data efficiently. Because the Cell/B.E. internally only operates with vectors, the use of SIMD is a much more native approach to exploit the hardware. The benefit can be seen in Table 5.2.
5.3 GPU Improvement I: Improving Speed

5.3.1 Motivation

The results shown and discussed in Section 3.3 show that Graphics Processing Units have a lot of potential when it comes to fast execution of the MMD algorithm. Especially the pure matching step, in which no replacement of templates is involved, the highly parallel hardware architecture can be very beneficial when it comes to fast execution. But as soon as it comes to positive template matches the architecture shows its weaknesses. Many costly data transfers over the comparatively slow communication bus presents a bottleneck.

The following section handles an approach that tries to solve the aforementioned issue of costly data transfers. The assumption, less data transfers is equal to faster execution is the fundamental concept of this implementation. Additionally, this approach tries to improve the cooperation between host and compute device in such a way that each architecture can profit from its strengths.

5.3.2 Design and Implementation

As already described in Section 3.3, in this approach, the GPU is only responsible for matching a given template against all available gates using as much parallelism as possible. This means, leveraged by the possibilities of the many core architecture and in particular by the used GPU, it is possible
to match up to eight gates at the same time. However, one downside of the current implementation is that as soon as a positive template match has been found and the template has been replaced, the whole circuit is sent back to the GPU and is re-matched. The original idea for this step was to emulate the behavior of the sequential implementation which follows a strict left to right order.

The basic idea in order to improve the algorithmic concept and to speed up the execution, especially for Graphics Processing Units, is to change its behavior on positive template matches and to concentrate more on the strengths and benefits of the two architectural platforms that are involved. Like the work described in Section 3.3.2 and Section 4.2, the first platform is a regular x86 architecture based CPU and serves as the host and the second, of course, is the GPU itself which serves as the OpenCL compute device. Obviously, since the algorithm has been implemented with the Open Computing Language, the compute device could also be the CPU itself.

Instead of only replacing the first positive template match and to stick to the aforementioned left to right order, this time the algorithm will try to replace as many templates as possible in one iteration. Iteration, in this context, means one round of template matching. Considering a circuit $C$ of size $n$, with $n > 0$, one iteration of template matching tries to match $n$ gates in parallel ($n$ threads). The result is an array of size $n$, of which $m$ elements, with $0 \leq m \leq n$, indicate positive matches. This array serves as a look-up table for several other buffers that are computed on and loaded from the
compute device to the host, and that provide all necessary information in order to replace a template. One of these buffers stores a list of all gates related to a positive template match. The idea now is to iterate over the whole look-up table array and to replace these positive matches sequentially. This means on the first encounter of a positive template match, the associated gates will be replaced. However, the following two problems have to be solved in order to be able to apply the new concept

1. Theoretically, it is possible that two consecutive, positive matches might interfere, since the same gates are involved. Remembering the look-ahead described earlier in Section 5.1.2 this would lead to an inconsistent circuit. The only way to avoid this issue is to skip this interfering match. Since the size of the look-ahead is known, it is possible to increase the index of the next position inside the result array. A constant value of 20 has been chosen, since this is the size of the look-ahead search space.

2. After a replacement circuit $C$ has been altered and has less gates than before. As a consequence, all calculated indices for the look-up table are invalid and can not be used for further computation. Assuming an index $i$ that matched a template at position $C_i$ inside the circuit, it is obvious that it now points to a completely different gate than before. In order to solve this problem, the old indices have to be recalculated in such a way that they re-match the old circuit structure. In fact, only the
next positive matching index has to be updated. The aforementioned index \( i \) will now point to \( i - d \), where \( d \) is the number of gates that have been replaced successfully. This means for each succeeding replacement the index will point to the correct position again.

After solving the aforementioned problems, the algorithm can execute normally and replace all positive template matches during one iteration. It works with the same template as long as new matches can be found. If there are no further positive matches in the result array, the next template is chosen. The algorithm terminates, when all templates have been tested successfully.

### 5.3.3 Results

The first parallel version implemented for NVIDIA Tesla based GPUs showed very varying results in speed. While some circuits executed very fast, others showed poor results, especially when many templates could be replaced (e.g. urf2). The main purpose of this improvement was to solve this problem and to achieve more reliable and constant results.

Table 5.3 shows the results of the previously discussed improvements. Column five shows the minimization results in direct comparison to old versions. Column two shows the original circuit size before the application of the template matching, column three shows the result of the sequential approach. The fourth column shows the results of the first parallel GPU based version. It can clearly be seen that the improvement did not have any positive effects
<table>
<thead>
<tr>
<th>function</th>
<th>Number of gates</th>
<th>Elapsed time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>orig</td>
<td>seq</td>
</tr>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1543</td>
</tr>
<tr>
<td>urf1</td>
<td>11554</td>
<td>7225</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
</tr>
<tr>
<td>urf5</td>
<td>10276</td>
<td>5582</td>
</tr>
<tr>
<td>urf6</td>
<td>10740</td>
<td>5455</td>
</tr>
</tbody>
</table>

Table 5.3: Results of the first GPU improvement. A speed-up could be achieved in comparison to the first GPU implementation.

on the minimization result. In some cases a worse result has been achieved (for example function urf2).

The last two columns are a direct speed comparison between the first parallel GPU version and the improvement. Here three results really are interesting. These are the results for large circuits (urf1, urf5, urf6). The first parallel version shows really bad results here. The improvement, however, shows a significant speed-up. This has a simple reason. Due to the high number of template replacements, many transfers between host and compute device have to be performed. For each successful template replacement, the whole circuit has to be sent again. Especially here, the new implementation can show its real potential and gain a speed-up between 38 and 44.
5.4 GPU Improvement II: Improving Minimization Results

5.4.1 Motivation

While the results in Section 5.3.3 already show a decent speed improvement, the actual minimization result could not benefit from this approach as expected. Most of these results are not very good compared to the original sequential approach or the results obtained using the Cell/B.E.. In order to have a better understanding for comparison and to evaluate how much better the minimization result can become, it is necessary to further improve the algorithmic approach.

5.4.2 Design and Implementation

The idea used for this implementation improvement is based on the following assumption. When replacing the positively matched gates as described in Section 5.3.2 it becomes clear that some potential replacements cannot be performed because they are too close due to the aforementioned look-ahead search space. This means that it is very likely that these skipped matches can still be replaced. In order to achieve a better result in terms of minimization it is necessary and beneficial to apply the matching algorithm at least once more. This way it is possible to process the skipped gates.

Because it is not possible to predict the number of additional template ap-
applications needed to gain the best possible result, an approach has been introduced that can be best described with a saturation analogy. The template application is repeated as often as a certain saturation level has been reached. This maximum saturation level is reached when no further template matches can be found.

**Algorithm 4 Saturation estimation**

```plaintext
var saturation
var level
repeat
  saturation = number of gates
  apply template matching
  level = number of gates
until level < saturation
```

The results which could be achieved by this algorithmic enhancement are presented and discussed in the next Section. It will be interesting to see if the basic assumption was correct and especially what the time penalty was and how it affected the overall performance.

### 5.4.3 Results

Table 5.4 shows the results for the second improvement. This improvement focussed on the GPU architecture in combination with the second parallel approach. The table shows a comparison between the first parallel GPU implementation and the improvement. Columns two to four show the minimization result, while the last two columns focus on the speed-up. Columns five and seven are the newly gained results.
<table>
<thead>
<tr>
<th>function</th>
<th>orig</th>
<th>seq</th>
<th>1st ver</th>
<th>2nd impro</th>
<th>1st ver</th>
<th>2nd impro</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus63mod4096</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>429</td>
<td>1.64</td>
<td>1.65</td>
</tr>
<tr>
<td>plus63mod8192</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>492</td>
<td>1.75</td>
<td>1.79</td>
</tr>
<tr>
<td>plus127mod8192</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>910</td>
<td>3.00</td>
<td>3.06</td>
</tr>
<tr>
<td>hwb7</td>
<td>289</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>1.84</td>
<td>2.87</td>
</tr>
<tr>
<td>hwb8</td>
<td>637</td>
<td>637</td>
<td>637</td>
<td>637</td>
<td>1.72</td>
<td>1.71</td>
</tr>
<tr>
<td>hwb9</td>
<td>1544</td>
<td>1541</td>
<td>1543</td>
<td>1543</td>
<td>4.21</td>
<td>8.19</td>
</tr>
<tr>
<td>urf1</td>
<td>11554</td>
<td>7225</td>
<td>9371</td>
<td>7240</td>
<td>1679.23</td>
<td>87.70</td>
</tr>
<tr>
<td>urf2</td>
<td>5030</td>
<td>3250</td>
<td>3913</td>
<td>3252</td>
<td>411.92</td>
<td>36.51</td>
</tr>
<tr>
<td>urf3</td>
<td>2732</td>
<td>2674</td>
<td>2680</td>
<td>2680</td>
<td>24.66</td>
<td>13.69</td>
</tr>
<tr>
<td>urf5</td>
<td>10276</td>
<td>5582</td>
<td>8039</td>
<td>5583</td>
<td>1713.74</td>
<td>76.37</td>
</tr>
<tr>
<td>urf6</td>
<td>10740</td>
<td>5455</td>
<td>8023</td>
<td>5456</td>
<td>2090.56</td>
<td>105.77</td>
</tr>
</tbody>
</table>

Table 5.4: Results of the second GPU improvement. The minimization result could be improved in comparison to the first GPU implementation.

One can clearly see that this improvement is very valuable in terms of the minimization result. In comparison to the previous implementations, the improvement achieves a better minimization result. The results are very close to those of the sequential version. Of course, this is not an optimal solution yet but it proves that the algorithmic approach could be improved successfully.

For most circuits, the runtime results are slower compared to the ones of the previous version. This outcome, however, has been expected due to the saturation approach the algorithm executes several times until no further matches can be found. In general, a significant speed-up is achieved. Since, among other things, the NVIDIA Tesla architecture has been designed for scalability it is very likely that the execution can be accelerated even more on GPUs that provide more streaming multiprocessors.
Chapter 6

Critique

The results discussed in the previous Chapters all show that utilizing parallelization concepts in conjunction with reversible logic synthesis, and in particular with the MMD algorithm, can help to improve the usability and effectiveness for gathering results or a future design flow. This work presented two different algorithmic concepts, both implemented on two different hardware architectures. The resulting four implementations give an insight into these concepts and their advantages and disadvantages. The questions that arises is which combination of parallel concept and hardware architecture is the most suitable for the given task. The following paragraphs discuss and review both, the parallel concepts and the utilized hardware architectures in order to find an answer to the aforementioned question.

The first parallel concept, originally introduced in [12], is a straight forward, divide and conquer based approach that partitions the input circuit into
equally sized parts. These parts can be processed independently and merged back together after the processing is completed. The structure of the Cell Broadband Engine can take advantage of this concept by mapping each part of the split-up circuit to a Synergistic Processing Element. A key concept of this approach is that each part of the circuit, and respectively each SPE, is fully responsible for processing the parts. The host side, namely the PPE, is not involved in any processing. It is only used to manage the SPEs, to gather the results and to merge them back into one single circuit.

In contrast to the first parallel concept, the second one, introduced in Section 3.3, tries to take advantage of the fact that Graphics Processing Units can occupy and manage hundreds of threads at the same time. Instead of taking full responsibility for matching and replacing, this concept only focuses on the matching step and sends the results for each gate back to the host. The host is responsible for the evaluation of the results and the actual replacement of any positive template match.

The results acquired for both parallel approaches clearly show that each concept matches exactly one hardware architecture. In other words, this means that the first parallel concept runs well on the Cell Broadband Engine and the second concept runs better on a GPU. Up to this point, only the actual runtime performance has been taken into consideration. When looking at the runtime results discussed in the respective Sections, the second parallel concept is a better choice. However, the main purpose of the MMD algorithm’s template matching step is to minimize an input circuit as small as
possible. Taking this purpose into consideration, the first parallel concept delivers slightly better results. However, the circuits minimized by the first parallel approach are only approximately 1% smaller than the ones minimized by the second parallel approach. This is an average of 2.9 gates for all functions that have been benchmarked in this work. When only considering the minimization result, the first parallel approach is the better choice.

Apart from the utilized parallel concepts, the two hardware architectures play an important role too, especially for future developments in the field of reversible logic synthesis and of course for the actual implementation of the parallel concepts. The results of Chapters 4 to 5 clearly show which parallel concept should be implemented on what kind of hardware architecture.

The first parallel concept maps best to hardware architectures like the Cell Broadband Engine. One benefit of the Cell/B.E. architecture is the heterogeneous concept of only one full featured CPU in contrast to the lightweight and highly optimized Synergistic Processing Elements. The SPUs that are located on the SPEs can take advantage of the $128 \times 128$ bit registers which allow fast access to data stored in them. Additionally, the concept of manually managed and scheduled memory transfers over the Element Interconnection Bus and the SPE’s DMA controllers are an advantage since a high bandwidth can be achieved. The SPE is a pure vector processor based on the SIMD paradigm. Utilizing SIMD instructions can leverage the real potential of the SPE. The results shown in Section 5.2.4 are proof for this. On the other hand, working with scalar data will result in poorer performance
because all scalar data has to be converted to a vector first. The current implementation of the template matching turned out to be very difficult to vectorize. The implementation is dependent on many branches, loops with conditional exit points and non-linear memory access patterns. Constructs as the aforementioned, however, will not work with SIMD code and should be avoided. The Cell/B.E. works best with data structures that can be vectorized and follow a linear code path with regular memory access patterns [41].

In order to fully exploit the Cell/B.E.’s hardware capabilities, the template matching algorithm needs to be completely re-implemented. A simple conversion of an arbitrary program written for a general purpose processor is not possible, except the program is not very complex. Unfortunately, this is not the case for the MMD template matching approach. However, when not looking at the actual hardware itself but instead at the general concept behind the hardware, a heterogeneous multi-core environment on a single chip, the overall results show that this kind of concept can be extremely valuable if the algorithm can be adapted to the special hardware.

The second utilized hardware was a NVIDIA GPU based on the Tesla architecture. The gained results show that this kind of hardware, which fits into the category many-core architecture [44], matches the second parallel concept. After the improvements implemented in this thesis, the GPU version shows good runtime results but lacks minimization results as good as the ones running on the Cell/B.E.. This kind of architectures generally benefits from its possibility to schedule and manage several hundred threads.
distributed over the available cores of the GPU. However, similarly to the Cell/B.E., these cores are no full featured units like a PowerPC or a x86 based CPU. Again, a GPU tries to take advantage of the concept of specialized, lightweight cores that perform extremely well on a certain set of problems or problem space. During this work it became clear that the template matching, in its current implementation, is not part of this set. The results show a decent speed-up in comparison to the sequential version. But when trying to exploit hardware specific features it becomes very difficult due to various reasons. Like SPEs, the GPU cores are designed to execute straightforward operations. This means, branches, expensive function calls or irregular memory access should be avoided. The used GPU architecture works best with fixed sized buffers that can be optimized for memory access before being sent to the GPU. However, the only fixed sized buffers in the current implementation are the circuit itself and the basic definition of each template. During runtime, several other buffers have to be generated whose size and content for example is dependent on previous runtime calculations. Additionally, as already mentioned in the preceding paragraph, the implementation consists of many branches and loops with conditional exit points. Another disadvantage of GPUs, especially in contrast to the Cell/B.E., is that data transfers run over a comparatively slow bus. In particular, when initializing the GPU and transferring all necessary data to the device, like the circuit itself or a template, a lot of potential performance is lost. This downside could be hidden however, by utilizing the fastest possible bus between main memory
and GPU. Again, in order to fully exploit a GPU’s hardware capabilities, it would be necessary to completely rewrite the template matching algorithm. The question whether to use parallel approach one or two and respectively which hardware architecture is difficult to answer. While none of the used hardware architectures seem to be perfectly suited for the current implementation, the GPU and its matching parallel concept seem to have more benefits when comparing GPU and Cell/B.E.. When neglecting the worse minimization result and only focussing on the execution time, the GPU already is faster than the Cell/B.E. version, although the used GPU for benchmarking the implementation is one of NVIDIA’s low-end products. The simple assumption is that a more costly GPU with a faster bus interface (for example PCI express) and more dedicated cores can process the algorithm even faster. Additionally, dedicated GPUs are a future-proof investment since they are developed continuously (NVIDIA just released the new Fermi architecture [45]), despite the fact that most modern computers already have a built-in graphics card. Here comes the advantage of using OpenCL into play, since it should be possible to easily port and run the current implementation on newer generations of GPUs too. The theoretically gained performance and more research and development could be used to close the gap between the minimization results of GPU and Cell Broadband Engine.

However, one has to ask the question if a re-implementation of the algorithm is worth the effort since it is no trivial task to map the algorithm to specialized hardware. Both, the Cell/B.E. and a GPU, fall into the category of hetero-
geneous multi-, respectively many-core environments. The heterogeneity of both architectures has been proven to be an advantage for many problem categories like the ones described in [44]. However, the MMD algorithm with its current implementation does not quite fit into the scheme that can benefit of the specialized hardware. Modern homogeneous CPUs for example work with up to 16 cores. Examining the use of this kind of hardware might lead to a better result. However, in the future when implementing an algorithm for the first time, parallel processing and special hardware capabilities, for example of GPUs, should be considered from the beginning. If the problem can be mapped efficiently, it is likely that a homogenous multi-core environment cannot compete with the specialized heterogeneous counterpart. The programming paradigm provided by OpenCL could also help to make it easier to implement parallelized versions of reversible logic synthesis algorithms that can be adapted to the most suitable hardware.
Concluding, Table 6.1 shows a comparison matrix that provides an overview over all implementations and the gained results on each of the benchmarked hardware architecture. The results shown here are the average execution time and percentage minimization of all benchmarked circuits.

<table>
<thead>
<tr>
<th></th>
<th>Cell/B.E.</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Implementation</td>
<td>527.82</td>
<td>5934.27</td>
</tr>
<tr>
<td>Cross-Implementation</td>
<td>1903.13</td>
<td>4015.77</td>
</tr>
<tr>
<td>Improved Implementation</td>
<td>453.83</td>
<td>339.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Cell/B.E.</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Implementation</td>
<td>0.15579 %</td>
<td>0.06328 %</td>
</tr>
<tr>
<td>Cross-Implementation</td>
<td>0.06328 %</td>
<td>0.15579 %</td>
</tr>
<tr>
<td>Improved Implementation</td>
<td>0.15686 %</td>
<td>0.15568 %</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison matrix of all implementations and architectures.

It is clearly visible that both runtime and minimization result could be improved on both platforms. Even if the latter is not as obvious as the former, a positive tendency is observable. Further optimizations and improvements may lead to even better results.
Chapter 7

Future Work

The results of this work have shown that parallelizing the MMD algorithm is possible, and can lead to a better and more useful design flow for reversible logic synthesis related applications. For future versions, however, several improvements could be implemented. These improvements, basically, fall into four different categories that will be addressed in the following paragraphs.

The first category of improvements aims to further optimize the execution time of the template matching approach. Whether it is the Cell Broadband Engine or a GPU architecture, the current implementation should be reimplemented with a focus on vectorization. Both architectures are able to work on vectorized data sets. While the Cell/B.E. supports a native SIMD engine, a GPU can simulate SIMD by exploiting the many cores and the general OpenCL programming paradigm.

The second category is related to the improvement of the minimization result.
At the moment the minimization results are equal or close to the results of the sequential version. Since the template matching approach aims to minimize a circuit as small as possible, some research should be done on this issue. The additional computational power that has been gained by the parallelization of the algorithm could be used to either optimize the current or implement other template matching strategies than the ones discussed in this thesis.

The third category aims to extend the template matching approach in such a way that it could be used for synthesis methods other than the MMD algorithm. At the moment the parallel version only works with Toffoli gates, hence it is necessary to introduce gates other than Toffoli. It already has been shown that an extension of the template matching approach is possible for different kinds of circuits [46, 47]. An extension of the parallel version makes it possible to use the template matching algorithm for a wider range of synthesis methods. This would provide researchers with a usable tool for optimizing circuits. In a worst case scenario no template can be replaced which will leave a given input circuit unchanged. As the presented results show, the time penalty in a synthesis design workflow is very small when no templates can be replaced.

For the fourth category of potential future work, the algorithm should be evaluated more, especially in comparison to newer or other hardware architectures. Additionally, the approach of using OpenCL for hardware independent, device agnostic development could be subject of further development and evaluation.
Chapter 8

Conclusion

The work discussed and presented in this thesis shows the general improvement of the MMD algorithm’s template matching optimization step. Compared to the works presented in Chapter 3, the algorithm could be successfully improved in two ways, execution time and minimization result. For both examined parallel concepts, the algorithm terminates faster and finds an exacter solution. However, the improvement of the minimization result has to be seen in relation to the original, sequential version. While the first parallel approach now delivers equally good results, the second parallel approach lacks the same precision. The newly achieved runtime results make the template matching approach usable and effective, even for large circuits that are based on Toffoli gates.

Additionally, in this thesis two multi-core architectures, the Cell Broadband Engine and NVIDIA Tesla GPUs, have been examined. The emphasis of this
examination was the usefulness of these architectures, in the context of the MMD algorithm’s template matching step. While the results on both architectures could be improved, the current implementation makes it difficult to exploit the special features of both hardware architectures, such as SIMD on the Cell/B.E. This means that a lot of potential of these platforms is unused. In order to leverage the potential, the implementation had to be rewritten.

However, the parallelization of the template matching approach might help to improve other works in the field and make them more usable and effective in the future. Especially, the whole design flow in reversible logic synthesis, where none exists so far [9] and it could benefit from parallelizing algorithms to make it more effective to develop, optimize and test both existing and new functions.
Bibliography


[Online]. Available: http://dx.doi.org/10.1109/ICCAD.2003.73
Vita

Candidate’s full name: Michael Schlösser
University attended: B.Sc. Computer Science 2008, Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Nordrhein-Westfalen, Germany

Publications: