Accelerating the MMD Algorithm using the Cell Broadband Engine

by
Michael Schlösser,
Rainer Herpers and Kenneth B. Kent

TR 10-202, May 26, 2010

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Reversible logic synthesis is an emerging research topic with different application areas like low-power CMOS design, quantum- and optical computing. The key motivation behind reversible logic synthesis is the optimization of the heat dissipation problem current architectures show, by reducing it to theoretically zero [2].

Currently, compared to conventional logic synthesis, design and synthesis methods for reversible functions are still in the beginning and need to be improved. Additionally, the synthesis of reversible function specifications is computationally very expensive and needs improvement in order to be used for an efficient design workflow.

The MMD algorithm, named after its originators Miller, Maslov and Dueck, is one approach to solve the synthesis problem for a certain group of reversible functions, the Toffoli based synthesis [3, 1, 4]. On top of the synthesis, the MMD algorithm tries to further optimize the synthesized reversible network. However, tests with the original implementation have shown that the computationally intensive optimization is very limited due to the time it takes to process especially large networks.

By utilizing modern parallel hardware architectures, like multicore systems, it is possible to reduce the time it takes to process and optimize input networks. This could result in a faster design and testing workflow, as well as the ability to process larger networks for potential real world applications in the future.

1.2 Structure

This work is structured into six chapters. The following description provides a quick overview over these parts. Readers already familiar with certain topics can skip chapters or sections.

The first chapter provides a quick introduction into the research areas covered and combined in this project. A short section outlines the motivations for this project and provides the reader with a rough idea why this work is of importance. The related work section then details previous work that has been done in this area.

Chapter two provides the reader with the basic knowledge that is necessary in order to understand the work done in this project. It will cover the basics of reversible logic synthesis, as well as the MMD algorithm, the target of acceleration. The section about the Cell Broadband Engine introduces the used hardware for this project. It aims to provide the reader with the fundamental knowledge about the hardware architecture that is necessary to understand how the acceleration problem has been tackled.

Chapter three then describes the actual project implementation. It starts with the preliminary work that had to be done in order to start implementing and rewriting the algorithm. The next section describes the most important data structures used by the implementation. In the section 3.3 the core item of the acceleration approach, the matching algorithm, is described. Firstly, the sequential approach will be explained, serving as a basis for the explanation of the transformation from a sequential to a parallel version. The last part of this chapter elaborates the problems that arose during the project development and the implementation process.

The next chapter will present and discuss the outcome of this project. It will answer the question whether the project goal is achieved on the basis of the benchmarks and tests.

Chapter 5 discusses improvements to the current approach that could be investigated for future work.

The last chapter then concludes the project and reflects the author's view upon the topic and outcome of this work.

Chapter 2

Background

2.1 Reversible Logic Synthesis

Reversible Logic Synthesis is a fairly new discipline and an emerging research topic. It has its root and application in several fields like quantum computing, low-power CMOS, nanotechnology and optical computing [4]. The fundamental principle behind Reversible Logic Synthesis is based on the fact that every logic computation that is not reversible generates a certain amount of heat for every bit of information that is lost [5, 6]. Very basic examples for irreversible gates are most of the fundamental logical operations like AND, OR and XOR. For example, consider the logical AND operation which has two inputs and one output. This means that during the computational process one bit of information gets lost. This lost information, however, generates the aforementioned heat. The only basic operation that is reversible is the NOT operation since it has one in- and one output and therefore does not lose any information. Other important properties of reversible operations are explained in the following.

Reversible functions (gates), in contrast to irreversible functions, have two fundamental properties or rather restrictions. Firstly, the number of outputs is equal to the number of inputs. Secondly, any output pattern has a unique preimage. In other words, a reversible (boolean) function $f(x_0, x_1, ..., x_n)$ is a bijection and performs permutations of the set of input vectors [4]. The logical XOR function $\oplus$, for example, can be transformed into a reversible function by adding an output that maps x to $\bar{x}$. The result is a function $(x \oplus y) \rightarrow (\bar{x}, x \oplus y)$

x	y	z	x	y	$z \oplus xy$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Table 2.1: Truth table defining the reversible AND operation.

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.

The logical AND operation, however, cannot be transformed into a reversible function by just adding a single output, since it is not possible to create a unique output pattern this way. In order to create the logical AND function it is necessary to add one in- and two outputs and map the set of input vectors to the vector $(x, y, z \oplus xy)$. This mapping results in the output vector shown in Table 2.1. For all constant $z = 0$ this function realizes the logical AND [2].

Additionally, the function shown in Table 2.1 describes the basic functionality of one of the best known and well studied reversible gates, called the Toffoli gate. Toffoli gates are the fundamental operation used in the MMD algorithm and therefore of special importance. In this example, Table 2.1 defines a Toffoli gate of size three.

In general, a Toffoli gate of size n can be defined as an operation 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 $TOFn(x_1, x_2, ..., x_n)$ 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

$$\begin{aligned} x'_i &= x_i, \quad i < n, && (2.1) \\ x'_n &= x_1 x_2 ... x_{n-1} \oplus x_n && (2.2) \end{aligned}$$

The graphical representation of Toffoli gates is shown in Figure 2.1. All lines corresponding to an input x_i with $i < n$ are depicted by the $\bullet$ and each target line by the $\oplus$ operation symbol.

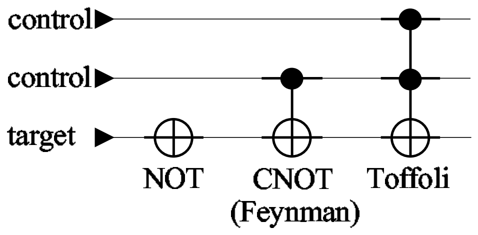


Figure 2.1: Graphical representation of Toffoli gates. NOT and CNOT are special cases of the generalized $x \times n$ Toffoli gate.

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 size three Toffoli gate is a generalization of all Toffoli gates of bigger size, since it defines how to map the value of the target line [1].

A reversible circuit or network 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. 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.

A logical consequence of the aforementioned restrictions of reversible gates is that a lot of unused inputs and outputs are introduced. Additional inputs are called *constant inputs* and the corresponding outputs are called *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 2.3 shows the relation between these sets [2].

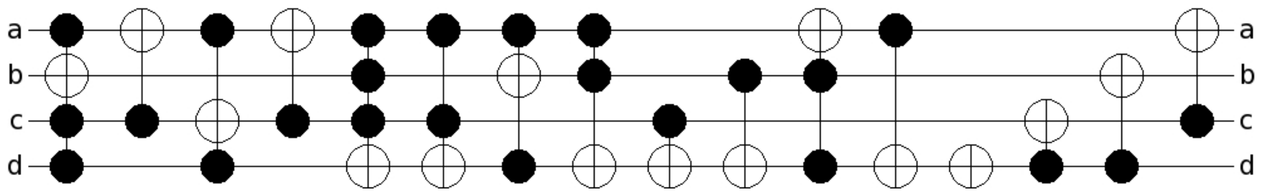


Figure 2.2: The function 4_49 has four in- and four outputs. It consists of 16 Toffoli gates of size one to four.

$$|I| + |C| = |O| + |G| \tag{2.3}$$

Research in reversible logic focuses on several areas. One of them is the synthesis of reversible networks. In other words, translate a formal reversible function specification $f(x_1, x_2, ..., x_n)$ (e.g. described by a truth table) into a reversible network, e.g. with Toffoli gates. Due 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, from which 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 [4].

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 from which 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

The Basic Algorithm

The basic synthesis algoritm 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 outputs 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 now tries to transform the output mapping by applying Toffoli gates in such a way that after the successful termination

$$f'(i) = i, \forall\ 0 \leq i \leq 2^n - 1 \tag{2.4}$$

holds. A correctness analysis of this approach is shown in [4] and [1].

To clarify how the algorithm synthesizes a given reversible function specification, consider the reversible function given in Table 2.1(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 gets negated with a NOT gate. In this example, one NOT gate is required, flipping all inputs of line a. Table 2.1(b) shows this in column ii. As a result of this operation, the first five inputs map correctly to their outputs, leaving three inputs to modify. By adding a Toffoli gate of size three with a as the target line, the inputs change to column iii of Table 2.1(b). The rest of the application is straight forward and results in the desired state, respectively circuit, after adding two more Toffoli gates. Because the synthesis process always modifies

(a) Truth table of a reversible function, before applying the MMD algorithm.

	i
cba	$f(cba)$
000	001
001	000
010	011
011	010
100	101
101	111
110	100
111	110

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

	i	*ii*	*iii*	*iv*	*v*
cba	$c^0b^0a^0$	$c^1b^1a^1$	$c^2b^2a^2$	$c^3b^3a^3$	$c^4b^4a^4$
000	001	000	000	000	000
001	000	001	001	001	001
010	011	010	010	010	010
011	010	011	011	011	011
100	101	100	100	100	100
101	111	110	111	101	101
110	100	101	101	111	110
111	110	111	110	110	111

Table 2.2: Example of the basic MMD algorithm

the function output the circuit is read in reverse in order to apply the gates.

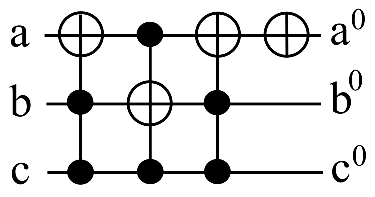


Figure 2.3: Resulting circuit for the function specified and synthesized in tables 2.2.2

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). This number of gates can be reduced by applying further approaches, like *Control Input Reduction*, *Bidirectional application* and *Asymptotically Optimally Modification*, also shown in [4] and [1].

The Bidirectional Algorithm

The basic algorithm, described in Section 2.2.2, always tries 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

[4]. This means that the algorithm applies its rules simultaneously on both the input and the output side, eventually yielding in a smaller circuit [2].

Again, to clarify the procedure an example shows the basic steps for the bidirectional algorithm.

	i	*ii*	*iii*	*iv*
cba	$c^0b^0a^0$	$c^1b^1a^1$	$c^2b^2a^2$	$c^3b^3a^3$
000	111	000	000	000
001	000	111	001	001
010	001	010	010	010
011	010	001	111	011
100	011	100	100	100
101	100	011	101	101
110	101	110	110	110
111	110	101	011	111

Table 2.3: All necessary steps in order to create a circuit with 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 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 done.

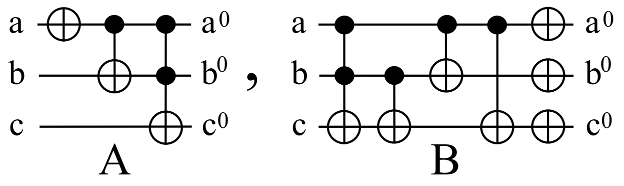


Figure 2.4: Circuit A shows the result of the bidirectional application. Circuit B is the naïve result of the basic algorithm shown in Section 2.2.2.

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

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 reversibility 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 [4], 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, 4]. 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 don't 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 don't 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 $TOFn(x_1, x_2, ..., x_n)$ be a Toffoli gate of size n where x_n is the target. Then two adjacent Toffoli gates $TOFk(x_1, x_2, ..., x_{k-1}, x_k)$ and $TOFl(y_1, y_2, ..., y_{l-1}, y_l)$ 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.

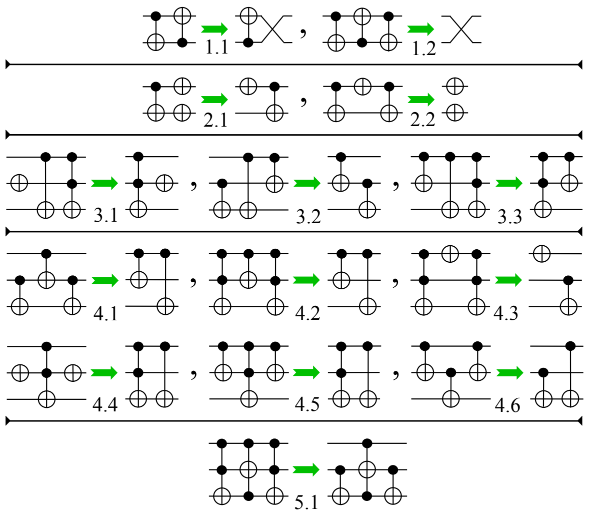


Figure 2.5: Template library defined in [1].

In [4] and [3], 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 $G^{-1}_{m-1}G^{-1}_{m-2}...G^{-1}_0$ 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 gets satisfied

$$G_1G_2...G_kG^{-1}_{k+s}G^{-1}_{k+s-1}...G^{-1}_{k+1} = Id \tag{2.7}$$

Observation 3: For $G_0G_1...G_{m-1} = Id$ and any parameter p, $0 \leq p \leq m$ $G_0G_1...G_{p-1} \rightarrow G^{-1}_{m-1}G^{-1}_{m-2}G^{-1}_{m-p}$ 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 of m gates which 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 [3].

2.3 The Cell Broadband Engine (Cell/B.E.)

2.3.1 Introduction

The Cell Broadband Engine (Cell/B.E.) is an implementation of the Cell Broadband Engine Architecture (CBEA), jointly developed by Sony, Toshiba and IBM. It is based upon the 64-bit PowerPC Architecture and extends it in order to provide a processor specification for parallel computing [7, 8]. In order to serve as a parallel processor, the Cell/B.E. is composed of a single PowerPC processor, called the PowerPC Processor Element (PPE), and eight smaller processors, the Synergistic Processor Elements (SPE) [9].

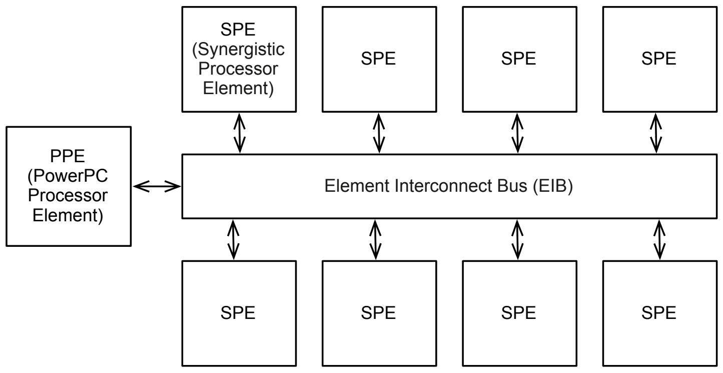

SPE (Synergistic Processor Element)
SPE
SPE
SPE
PPE (PowerPC Processor Element)
Element Interconnect Bus (EIB)
SPE
SPE
SPE
SPE

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

These processors are interconnected with a coherent bus system, the Element Interconnection Bus (EIB) which is used to provide data communication between the processors. Figure 2.6 illustrates the basic scheme of a CBEA compliant system.

The design idea behind the Cell Broadband Engine is to provide an extensible and scalable architecture that comes up with a solution for the following goals as described in [9]:

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

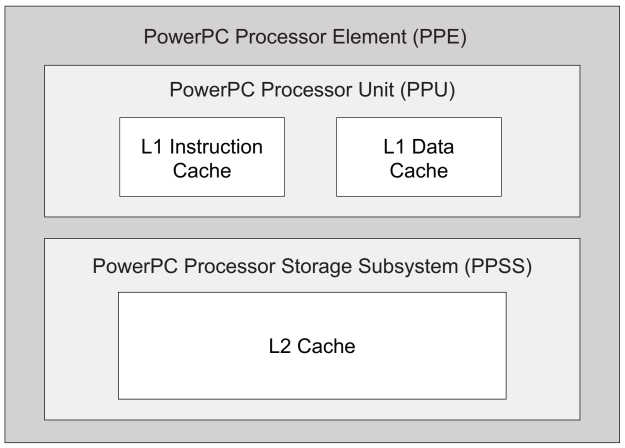
The way the two processor types were chosen make it possible to provide the aforementioned design criteria. The PPE is a general purpose CPU that can be used for any kinds of tasks, like running an operating system, managing system resources or other control intensive tasks (control-plane). The SPEs on the other hand are designed to fulfill compute intensive tasks (data-plane).
The way a Cell/B.E. program usually works is by implementing all control logic on the PPE. All computationally intensive work gets delegated to the SPEs which are highly optimized for these tasks. In other words, the PPE serves as the brain of each application, whereas the SPEs are highly effective workers that are able to perform a lot of computational work in parallel. Asynchronous Direct Memory Access (DMA) makes it possible to optimize memory usage and transfer between the PPE and SPEs.

Due to the architectural design, PPE and SPEs have a strong dependency relationship. The PPE does not have enough computational power on its own in order to achieve high performance computation. This is why it depends on the computational power of the SPEs. On the other hand, the SPEs are not designed to fulfill control intensive tasks, so they rely on the PPEs control logic.

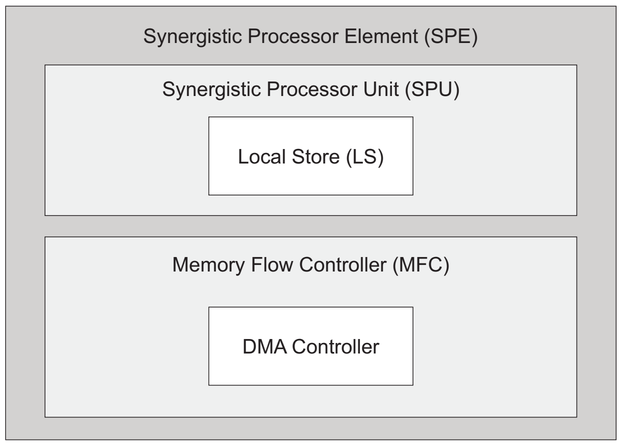
2.3.2 The PowerPC Processing Element

The PowerPC Processing Element is based on the 64-Bit PowerPC architecture and therefore can be programmed as such. It consists of two main units, the Power Processor Unit (PPU) and the Power Processor Storage Subsystem (PPSS) as depicted in Figure 2.7(a).

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.


PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

(a) The PowerPC Processing Element is a general purpose CPU used for control intensive tasks. (Source: [9])


Synergistic Processor Element (SPE)
Synergistic Processor Unit (SPU)
Local Store (LS)
Memory Flow Controller (MFC)
DMA Controller

(b) The Synergistic Processing Element is optimized for compute intensive tasks and gets all necessary data from the PPE. (Source: [9])

Additionally, it supports multithreading and handles two threads that can be seen as a 2-way multiprocessor with shared dataflow [9].

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 [9].

2.3.3 The Synergistic Processing Element

The SPE is a 128-bit RISC processor and consists of two main units, the Synergistic Processing Unit (SPU) which has access to a 256-KB Local Store and the Memory Flow Controller (MFC) which is responsible for memory transactions between the local store and main memory. Figure 2.7(b) illustrates the structure of a 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).

In order to work with data, each SPE has a Memory Flow Controller (MFC) which interfaces with the main memory and SPU. It uses a DMA controller to load and store data in its Local Store or in the main storage. All DMA calls get stored in a process queue and can be processed asynchronously, while the SPU can keep on working.

Chapter 3

Implementation

3.1 Prerequisites

3.1.1 Hardware: Playstation 3

The Playstation 3 (PS3) is a gaming console, developed by Sony Computer Entertainment and firstly announced in 2005 [10]. It contains a Cell/B.E. processor running at 3.2 GHz on each processor, the PPE and the SPEs. Unlike the Cell Processors described in the official IBM documentation, the PS3 is equipped with seven instead of eight SPEs, one reserved for redundancy, which still is compliant to the official Cell Broadband Engine Architecture specification. Its total floating point performance was announced as 218 GFLOPS [7, 9, 10]. However, in order to make manufacturing the Cell/B.E. more cost effective one of the seven SPEs is disabled, leaving only six freely programmable SPEs [11, 12].

There are several reasons why a Playstation 3 has been chosen for the implementation. Several other products on the market, like the IBM BladeCenter QS21, contain a Cell/B.E. processor [13] but when comparing prices 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. Therefore, the Playstation is very well suited as a proof of concept development platform.
Additionally, the Playstation has proven to be a good choice for scientific applications [12].

It provides everything needed for different scientific applications. Besides the Cell/B.E. processor, which is the actual reason to prefer a PS3 over a regular PC, it also contains a network interface which makes it possible to use the Playstation in a cluster to further extend computational power .

In order to use the Playstation 3 for scientific computing, Sony provided the ability to install a Linux distribution on the PS3. Coupled with the official IBM Software Development Toolkit, this feature makes it possible to utilize the PS3 as a normal Unix-based computer, for example for the aforementioned purpose.

Unfortunately, Sony removed this feature at first with the release of the Playstation Slim in August 2009 [14] and finally with the latest firmware upgrade in April 2010 for all remaining systems that get upgraded to the latest firmware [15]. It is, however, possible to still 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.

3.1.2 Software

Before the actual development process on a Playstation 3 can start, a couple of preparations have to be made. First of all it is necessary to install a Cell compatible Linux distribution. Secondly, the IBM Software Development Kit needs to be installed and configured.

In the following two sections the procedure of preparing a Playstation 3 for scientific application is described.

Installing the OtherOS: GNU/Linux

The old versions of the Playstation 3 (before PS3 Slim) were shipped with the regular Playstation operating system, called Game OS. Fortunately, Sony provided the possibility, to install other operating systems, called Other OS. Since the Cell Broadband Engine contains a regular 64-Bit PowerPC processor, several GNU/Linux (named Linux in the following) distributions are able to run on the Cell/B.E. and therefore on a Playstation 3.

First of all, there is the Yellow Dog distribution, officially supported from Sony, but also other distributions, like Ubuntu or Fedora can be installed. Since Fedora is the officially supported distribution by IBM, who also provide the official Software Development Kit, this distribution has been chosen for this project.

There are several tutorials that describe how to install the OtherOS. For this project an official tutorial from the IBM developerWorks online platform has been used as a basis [16]. All Yellow Dog Linux related information also apply for the Fedora Linux distribution. The installation process is very straight forward and consists of the following steps

1. partition the PS3 hard disk drive
2. install a new boot loader
3. install the linux distribution

The first step is to partition the hard disk inside the Playstation 3. The GameOS provides a menu point for this and performs this step automatically. In order to be able to boot an operating system other than the original GamesOS, the boot loader has to be replaced with a new one. It also can be installed via a GameOS menu point. All that is needed is a USB stick that contains the actual boot loader. After installing the boot loader the PS3 is ready to install the Linux distribution of choice. After a reboot the Fedora installation starts automatically and can be installed like on every other computer.

After the installation Fedora works with a regular X11 based Graphical User Interface (GUI). It turned out that the PPE is not powerful enough to allow work with the GUI. Therefore the X11 server should be disabled. Instead, working with a SSH based remote connection proved to be a sufficient setup.

Installing the Cell/B.E. SDK

In order to write programs for the Cell Broadband Engine, a Software Development Kit is needed. IBM provides the official SDK over their developerWorks website and can be downloaded free of charge [17].

The SDK contains a GNU based compiler tool-chain and a set of libraries in order to program applications for the Cell/B.E. processor [9, 18]. After downloading the SDK from the IBM website, it can be installed with the rpm package management software provided by Fedora.

3.2 Data Structures

The original implementation of the MMD algorithm has been realized in the C++ programming language. It is divided into three phases, reading in and processing an already synthesized circuit specification, reading in and processing a library of templates and finally executing the template matching algorithm. Figure 3.1 illustrates the program flow and the three phases.

In the first phase, an already synthesized circuit gets read in and processed. The specification of that circuit is defined in an external file and has to be loaded and processed at runtime. The specifications are encoded in a plain text file format provided at [19].

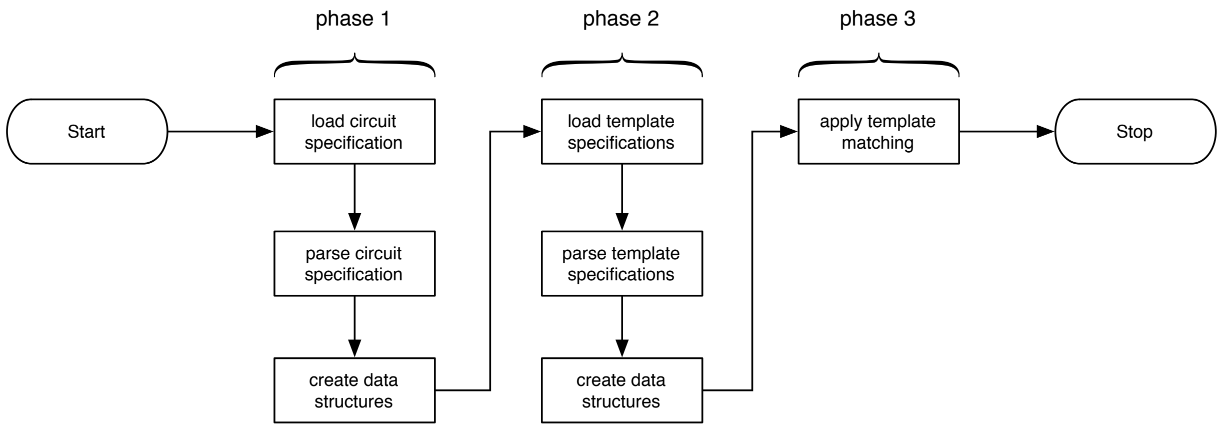

phase 1
phase 2
phase 3
Start
load circuit specification
parse circuit specification
create data structures
load template specifications
parse template specifications
create data structures
apply template matching
Stop

Figure 3.1: The implementation of the MMD algorithm is divided into three phases. The first two phases prepare all necessary data structures by loading and parsing external specifications. In the last phase, the data structures are needed for the template matching.

Each circuit is composed of a number of lines and gates. The number of lines corresponds to the number of inputs and outputs. As described in Section 2.1 the number of in- and outputs of a reversible logic gate are equal. The same applies to the whole circuit, so it is not necessary to store two different values for in- and output. The number of gates is self-explanatory. The total cost of a circuit are measured in quantum costs. Since, the MMD

```

struct circuit_t {
    int numLines;
    int numGates;
    int totalCost;
    struct gate_t** gates;
};

struct gate_t {
    bool swapf;          // is it a swap-gate
    int target;          // variable to be inverted
    int control;         // conditions for inversion
};
```

Listing 3.1: A reversible circuit is stored as a list of gates. Each gate is composed of control and target lines.

algoritm is based on the synthesis and optimization of Toffoli gate based networks, each gate in the circuit specification can be regarded as a Toffoli gate consisting of a target and one or more controls.

The target and controls are encoded as single integers which has several benefits. First of all, with this representation it is possible to store up to 2^n lines in one integer, where n is the number of bits. On a Playstation 3 running in 32 bit mode this is four bytes, respectively it is possible to store up to 32 lines in one integer. This encoding does not only have small space requirements but also makes it possible to work with fast bit operations.

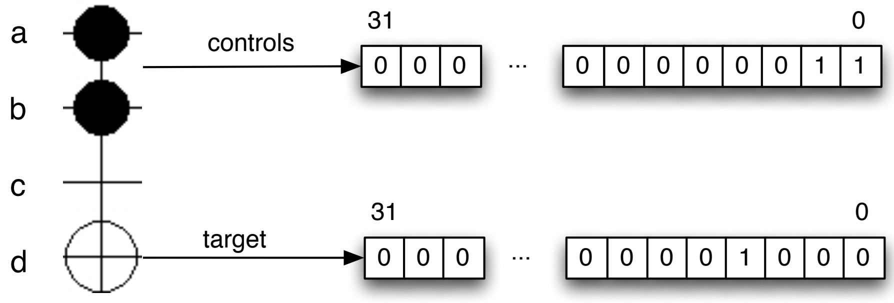


Figure 3.2: In order to map a Toffoli gate to memory, an encoding with minimum space requirements has been chosen. It maps each gate to a power of two.

Like circuits, templates are also stored in external files and need to be loaded and processed

```

struct template_t {
    int ng;                // number of gates in the pattern
    int csize;             // size of cpat
    int tsize;                      // size of tpat
    struct cpat_t* valid_cpat;  // the valid control pattern
    struct tpat_t* valid_tpat;  // the valid pattern on target
       lines
};

struct cpat_t {
    int ng;                // number of gates in the pattern
    int cpattern;          // pattern
    int rev_cpattern;      // reverse pattern
};

struct tpat_t {
    int ng;                // number of gates in the pattern
    int tpattern;          // targets in the patter
    int cpattern;          // controls in the lines
    int rev_tpattern;      // pattern
    int rev_cpattern;      // reverse pattern
};
```

Listing 3.2: A template is defined as a list of valid control and target patterns.

before executing the template matching algorithm. As described in Section 2.2.3 templates can be described as a sequence of gates that implement the identity function, in other words each template is a circuit itself computing the function *id*. After reading in a template specification, it is at first stored as a circuit of type *circuit_t*. From this circuit specification it is possible to derive a set of patterns that can be used for the template matching.

3.3 The Matching Algorithm

3.3.1 Sequential Approach

The original implementation of the MMD algorithm has been programmed with the C++ programming language. The first step for this project was to examine the source code in order to understand the data structures and the practical implementation of the template matching theory. During this process, it became clear that the original source code is very ineffective, especially when it comes to costly memory operations. In order to fix this problem, the algorithm has been implemented from scratch with the C programming language, using the C99 standard [20]. All data structures and algorithmic implementations have been adopted, but memory management and operations have been optimized. The port already showed good runtime improvements over the C++ version. These results will be discussed in Chapter 4.

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 3.2. After successful termination the algorithm returns a modified circuit C' with

$$|C'| \leq |C| \tag{3.1}$$

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 *start*, the algorithm tries to match all templates, where $|T|$ denotes the number of templates in the 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 starts the whole procedure again.

If a template could 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 don't have to be matched again. To avoid these unnecessary matches and instead of setting $start = 1$, it gets set to a value offsetting these gates.

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
```

3.3.2 Parallel Approach

While the sequential algorithm already shows good results in minimizing the input circuit, it can become very slow with increasing circuit size and the number of positive matches. Another problem is the locality of the matching approach. This means only a small part of the circuit can be tested for sufficient matches, leaving the rest idle. Being able to match a template on several parts of the circuit at the same time could improve the speed of the algorithm significantly. Using the computational power of the six available SPEs of the Cell/B.E. to achieve this goal is the basic idea followed in this project.

The parallelization concept

The template matching algorithm can be accelerated by partitioning the input circuit C into t parts, minimizing these parts independently in parallel and in the end merge them

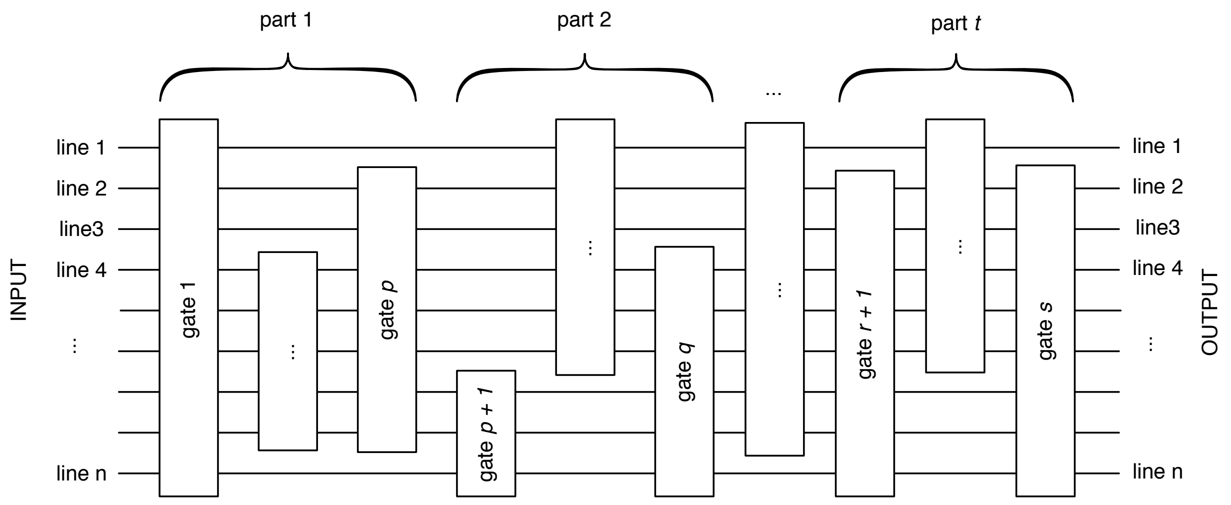


Figure 3.3: In order to process an input network in parallel it gets partitioned into t parts which get processed independently. After successful termination the parts get merged to a single circuit again.

together while still computing the same function. In order to work, this concept requires two assumptions to hold

1. It must be possible to merge reversible networks while holding their reversible properties.
2. If the first assumption holds, when merging the independently minimized parts they still must compute the same function.

The first assumption can be proofed by considering a reversible circuit of size one and then successively adding more gates. The outcome of a network consisting only of one gate of course is a permutation of the input. Since reversible circuits are organized in cascades, the output of gate G_1 denotes the input of exactly one successor, a gate G_2. If the output of G_1 is a permutation of the input, this permutation of course is the input for gate G_2. But because G_2 is a reversible gate as well, its output again is another permutation of its inputs, which holds the basic requirement of a reversible network.

It is obvious 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.

For the second assumption to hold consider a reversible logic 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. Each part now computes a function f_i with $1 \leq i \leq t$.

$$\begin{aligned} f_1 &= G_1G_2\cdots G_p && (3.2)\\ f_2 &= G_{p+1}G_{p+2}\cdots G_q && (3.3)\\ &\vdots && (3.4)\\ f_t &= G_{r+1}G_{r+2}\cdots G_s && (3.5) \end{aligned}$$

The original function f then can be described as

$$f = f_1 \circ f_2 \circ f_3 \circ \cdots f_t \qquad (3.6)$$

When applying the template matching algorithm to the independent parts of the circuit, where 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 procedure.

$$f_i = f_i' \quad \forall\, 1 \leq i \leq t \qquad (3.7)$$

After successful termination the reversible network computes the function f' with

$$f' = f_1' \circ f_2' \circ f_3' \circ \cdots f_t' \qquad (3.8)$$

Combining (3.6) and (3.7) leads to the following conclusion, showing that assumption two holds.

$$\begin{aligned} f &= f_1 \circ f_2 \circ f_3 \circ \cdots f_t && (3.9)\\ &= f_1' \circ f_2' \circ f_3' \circ \cdots f_t' && (3.10)\\ &= f' && (3.11) \end{aligned}$$

The two proofs show that it is possible to subdivide the input circuit and to apply the template matching algorithm.

The PPE implementation

The implementation on the PowerPC Processing Element is divided into five phases, illustrated in Figure 3.4. In the first stage the input circuit C gets partitioned. The number of parts defines how many SPEs get used, whereas it is possible to divide the circuit in more than six parts. If this is the case, one SPE is responsible for calculating more than one part.

There are several possible strategies for partitioning the circuit but for this project the most intuitive has been implemented. It simply divides the number of gates by the number of wanted parts. In most cases a circuit cannot be divided evenly into the same number of gates, leaving the last SPE with more gates than the rest. However, this is no big problem since the maximum number of gates which the last SPE has to compute more than the others is limited by the number of parts.

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

$$s_i = \begin{cases} \lfloor \frac{g}{n} \rfloor & \text{if } i < n, \\ \lfloor \frac{g}{n} \rfloor + (g \bmod n) & \text{else} \end{cases} \tag{3.12}$$

This leaves for the last SPE a maximum of $g \bmod n$ more gates to compute which is computable in a reasonable time if n is not too big. Reasons why choosing a big n is not recommended is explained in Section 3.4.

In the next phase the SPEs get prepared for execution. This process involves opening the compiled SPE binary, creating a context for each used SPE and loading the binary into this context. One important part of this phase is to provide each SPE with all necessary information needed for the computation. In case of the template matching algorithm this information consists of the following information stored in a C struct whose address gets submitted to each SPE main entry point.

1. number of lines
2. size of the circuit-part (number of gates)
3. start address of the circuit-part

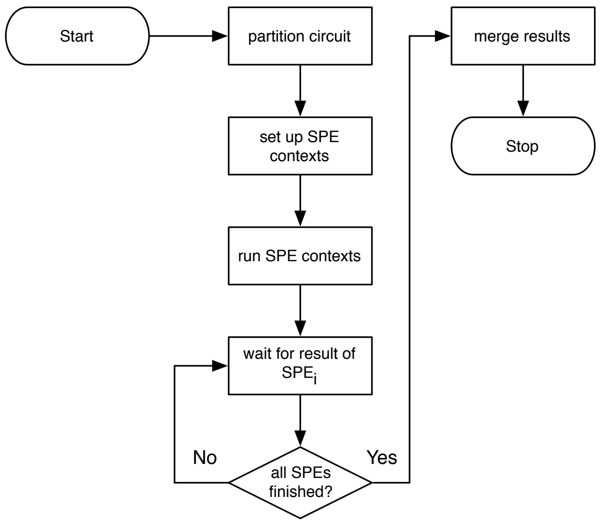


Figure 3.4: The PPE prepares the input circuit for the parallel execution and initializes the SPEs. After all SPEs have finished computation the PPE gathers the computed results and merges the circuit.

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)

The addresses for list points 4 and 5 are trivial since each address points to the beginning of each buffer. List point 3 needs to be computed and is dependent on the number of parts and their size. Let all gates be stored in a buffer *gates*. The number of parts is n, each of size s_i, then each starting address a_i can be obtained by multiplying the current index of each circuit-part with its size. Algorithm 2 illustrates this, where the & operator is used to retrieve the address, like in the C programming language.

Algorithm 2 Obtain address of circuit-parts

```
1: for $i = 0; i < n; i++$ do
2:     $a_i = \&gates[i \cdot s_i]$
3: end for
```

When all context parameters are correctly set for each SPE, the context can be executed. Because the PPE waits for each context to terminate, the implementation uses POSIX threads on the PPE side to invoke all SPEs concurrently. When all SPE's finished their

```
spu_mfcdma64(&buf, mfc_ea2h(addr), mfc_ea2l(addr), size, tag,
    MFC_GET_CMD);
spu_writech(MFC_WrTagMask, 1 << tag);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Listing 3.3: A SPE loads data into its local store by issuing a DMA call.

computation, the SPE contexts need to be destroyed. Because all SPE's wrote their results to an output buffer, the PPE can access these memory locations and merge the circuit again.

Implementation on the SPE

The work on each Synergistic Processing Elements is structured like the sequential approach, it even uses the same algorithm as shown in algorithm Listing 1. Because the SPEs access the main memory over DMA calls, it was necessary to adapt the sequential source code in order to load and store data.

As mentioned in Section 3.3.2, the PPE passes the address of a struct to the SPE's main entry point, containing all necessary values and addresses the SPE needs in order to work properly. Because the SPE only has access to the struct's address, the very first thing to do is to issue a DMA call in order to load the struct into the SPE's local store. Listing 3.3 shows the code needed for loading data into the local store. This code is used for all DMA calls throughout the SPE application.

The function *spu_mfcdma64* takes six arguments and interacts with the Memory Flow Controller in order to start the DMA call. The first argument is a pointer to a buffer located in the local store, in which the loaded data will be copied. The next two parameters are a 64-bit address to the location in main memory from which the data gets loaded. The two macros *mfc_ea2h* and *mfc_ea2l* split the provided address into two 32-bit parts, one containing the upper and the other the lower 32 bits. The size parameter specifies the amount of bytes that need to be copied. The next parameter specifies a DMA tag in the range of 0 to 31 which specifies an identification number which is used to check the completion of a DMA transfer. The last parameter specifies the transfer direction and can take two values *MFC_GET_CMD* for reading data into the local store and *MFC_PUT_CMD* to write from the local store into main memory. The next two calls to *spu_writech* and *spu_mfcstat*, where *spu_writech* is a

```
struct gate_t {
    bool swapf;
    bool pad1[3];          // pad with 3 bytes
    int target;
    int control;
    int pad2;              // pad with 4 bytes
};
```

Listing 3.4: The struct gate_t must be padded in order to be transferred between PPE and SPE.

macro, are used to wait for the DMA call to complete.

Once the struct containing all necessary parameters has been loaded, its values can be used to create all necessary buffers in the local store and to load all needed data from main memory. As described in Section 3.3.2, this data is composed of all circuit related data, the template library and information needed to write back the results.

The next step that follows is the execution of the template matching algorithm with the loaded circuit-part. The process here is identical to the process described in 3.3.2. After successful termination, a DMA call is issued to write back the data to main memory.

3.4 Problems

Several problems and issues appeared while implementing the parallel approach, all of them related to memory management.

First of all, the Cell Broadband Engine only works properly, if all memory operations between PPE and SPE are memory aligned. This means that all structures and buffers need to be aligned on a certain byte boundary in memory and in some cases need to be padded to a multiple of 16. Using the example of a C struct, padding means that the struct needs to be inflated artificially to fit into a 16 byte fetch instruction. Considering the struct *gate_t* from Listing 3.1 which is 9 byte it needs to be padded with 7 additional bytes. The resulting struct looks like Listing 3.4. After padding the struct, it is composed of 4×4 bytes $= 16$ bytes and perfectly fits into a DMA instruction. If all structs are not padded properly, misaligned buffers will result in a bus error. Additionally, each buffer needs to start on an address that

```c
// create a buffer of gates, aligned on a 16 byte boundary
struct gate_t gates[1024] __attribute__((aligend(16)));
```

Listing 3.5: Ensuring that a buffer is aligned properly, is done with the attribute keyword.

```c

#define MAX_LOAD 0x4000

void spe_dma_get(void* buf, unsigned int addr, size_t size, int
    tag)
{
    unsigned int i = 0;
    unsigned int next = addr;
    size_t chunk_size;
    do {
        chunk_size = ((addr + size) - next >= MAX_LOAD) ? MAX_LOAD
            : (addr + size) - next;
        spu_mfcdma64(&buf[i], mfc_ea2h(next), mfc_ea2l(next),
            chunk_size, tag, MFC_GET_CMD);
        spu_writech(MFC_WrTagMask, 1 << tag);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        next += chunk_size;
        i += chunk_size;
    } while (next < addr + size);
}
```

Listing 3.6: A single DMA call has a maximum possible size of 16 KB and needs to be split into chunks in order to load larger circuits.

is a multiple of 16. It is possible to ensure this by using the *__attribute__* keyword, as shown in Listing 3.5 [9].

A single DMA call can issue a maximum of 16 KB which means in order to transfer larger amounts of memory, it needs to get split into chunks. However, some circuit-parts can become bigger than this maximum size. For these cases the DMA load and store needed to be rewritten. Listing 3.6 shows the solution that has been implemented for this project and solved previous issues, which previously lead to undefined behavior and bus errors.

Chapter 4

Results

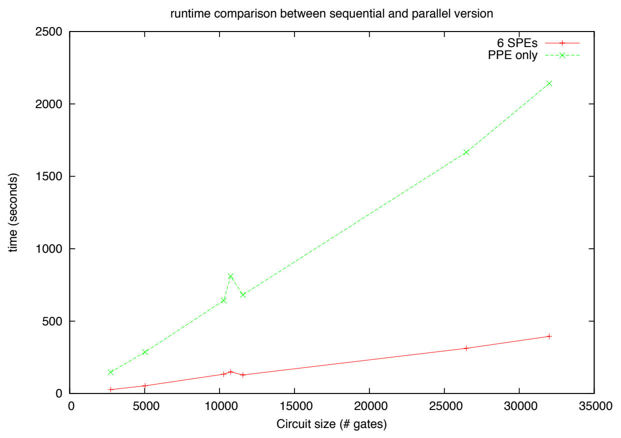
As mentioned in Chapter 3, the first optimization and also acceleration has been achieved by rewriting the algorithm in the C programming language. Table 4.1 shows these results with a selection of smaller circuits that already show the potential of the C port. All following tests that run on the PPE only (sequential version) are based on the C version. All time measurements show the elapsed time in seconds.

For all tests a set of reversible networks with varying size and complexity have been chosen. All of these and more circuit specifications can be downloaded at [19, 21]. The size of a network is defined only by its number of gates, whereas the complexity in this context means the number of gates that can be reduced. Each replacement comes at a cost, so that the more gates that can be reduced, the longer the algorithm needs for the whole process. This means that a circuit with less gates than another but with a higher complexity needs longer to terminate.

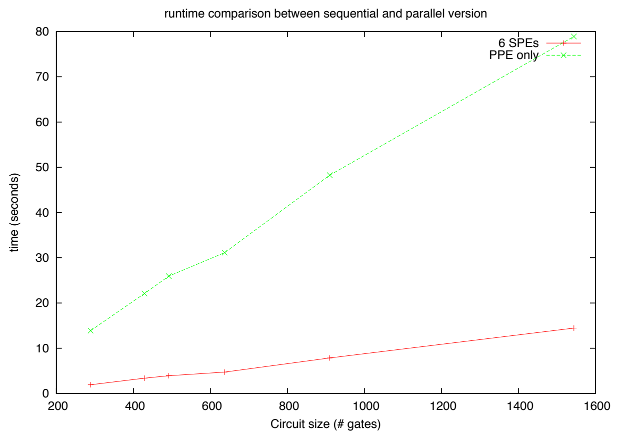
Table 4.2 shows the results of the aforementioned circuits. It consists of the unstructured

function	C++	C99
hwb7	86.03	13.91
hwb8	364.54	31.14
hwb9	2029.21	78.91

Table 4.1: The C99 implementation already shows a huge speedup. The results show the elapsed time in seconds.



(a)



(b)

Figure 4.1: Both graphs show the growth behavior of the functions tested in Table 4.2 and 4.3. It can be seen that the parallel version decreases the runtime significantly.

	Number of gates			Elapsed time		
function	before	PPE	6 SPEs	PPE	6 SPEs	speedup
urf1	11554	7225	7233	681.13	128.06	5.3
urf2	5030	3250	3251	286.54	53.15	5.3
urf3 1	2732	2674	2674	147.10	26.97	5.4
urf3 2	26468	15517	15527	1666.55	312.22	5.3
urf4	32004	9969	10085	2141.69	394.89	5.4
urf5	10276	5582	5585	642.15	133.39	4.8
urf6	10740	5455	5488	810.13	149.72	5.4

Table 4.2: 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.

	Number of gates			Elapsed time		
function	before	PPE	6 SPEs	PPE	6 SPEs	speedup
plus63mod4096	429	429	429	22.11	3.39	6.5
plus63mod8192	492	492	492	25.95	3.94	6.5
plus127mod8192	910	910	910	48.27	7.87	6.3
hwb7	289	284	284	13.91	2.13	7.2
hwb8	637	637	637	31.14	4.74	6.5
hwb9	1544	1541	1541	78.91	14.46	5.4

Table 4.3: Speedup comparison of functions with almost equal minimization behavior

reversible function (urf) benchmark. This benchmark is a set of reversible functions that do not have a regular structure in their specifications and are the largest that are currently available on [19] and provided a good basis for testing the parallel implementation. The parallel version used six SPEs which means that all SPEs in the Playstation 3 have been responsible for exactly one circuit-part.

The table shows for each function the number of gates before and after the template application and the time needed to successfully terminate. It can be clearly seen that the acceleration of the MMD algorithm was a success. However, this acceleration comes at cost. When comparing columns two, three and four it can be seen that the parallel version in most cases delivers worse results than the sequential one running on the PPE only. An example for this is the urf3 function which has been tested in two versions. The first version (urf3 1) uses a circuit that has been synthesized and minimized with the MMD algorithm but without template matching (see section 2.2.2). The algorithm started with 2732 gates and could further minimize it to 2674 gates on both the PPE and the parallel version. The second version, however, shows a slightly worse result as all other functions do. The difference can be explained by the way the parallel algorithm works. Since every circuit is split into n parts there are $n-1$ cuts. This can become a problem if a possible template match could be found across a cut. All parts get processed independently and therefore don't affect any other part. This means each template match across a cut won't be recognized.

Table 4.3 shows a couple of other functions which other than the urf functions might show regular structures and are not artificially created in order to test reversible logic synthesis methods. It can be seen that all functions deliver the same results and therefore can be compared directly. The speedup of the first three functions lies between 6.3 and 6.5. The hwb functions lie between 5.4 and 7.2. It is interesting to see that the speedup is nearly constant for all functions that cannot be minimized any further and maps to the number of used SPEs. The function hwb9 can be further minimized and produces the same output. Here the speedup goes down to 5.4. The largest speedup can be seen at function hwb7 but this speedup can be explained by the minimizing costs of the sequential version. Compared to the speedup results of Table 4.2 these results are slightly better which can be explained by the complexity of the circuits used. Since the urf functions show no regular structure, are bigger and can be minimized significantly they do have a higher complexity which results in a smaller speedup of an average of 5.3.

Another interesting result shows Figure 4.2. With an increasing number of threads, here using the example of the hwb9 function, the runtime can be accelerated even more. In this example the result has not been affected by the number of circuit-parts. Generally, the number of circuit-parts plays an important role. When increasing the number of threads, the number of parts becomes smaller. On the one hand, smaller parts mean that each part can be processed faster since less matching attempts have to be calculated. However, on the other hand, decreasing the size of each circuit-part leads to a smaller area in which potential matches can be found. In other words, the smaller the circuit parts, the faster the computation but the worse the result. Of course this is not always true, as the results of Figure 4.2 show.

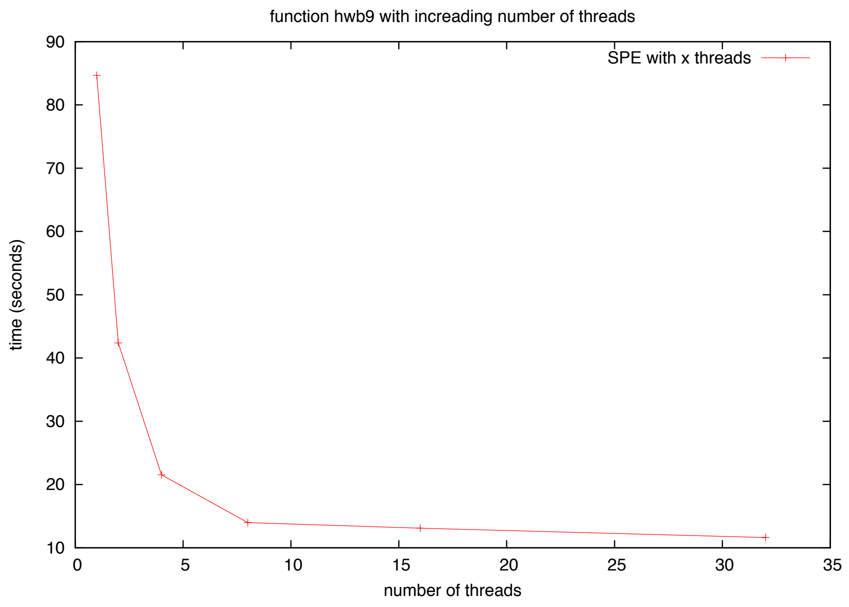


Figure 4.2: An increasing number of threads leads to a better runtime behavior.

Chapter 5

Future Work

The results show, that a speed improvement could be achieved. However, the Cell/B.E. architecture has the potential to achieve even better results in future versions. There are several questions and issues that could be improved. In the following, these questions shall be presented and discussed.

The first thing that could further speed up the runtime significantly, is the usage of the Cell/B.E. SIMD capabilities. With this improvement, it could be possible to work on several template matches simultaneously on the Synergistic Processing Elements. A different approach might be to vectorize just one single match which, at the moment, tests for matching control and target lines sequentially. With the SIMD approach, this could be done in one vector operation. Another approach to speed up the runtime could be the optimization of memory management and DMA transfers. At the moment all DMA transfers operate on the same channel and need to wait until one transfer is done. Using overlapping or asynchronous data transfers could be helpful in order to gain performance.

At the moment, the encoding of a Toffoli gate allows to store up to 32 inputs. In order to be possible to compute circuits with more than 32 inputs, a more generic but still space-effective solution can be found. The computational power of the current implementation should be big enough to handle that many inputs.

The results have also shown, that the quality of the minimization process needs to be optimized. Most circuits show worse results than the sequential approach and therefore need to be optimized. The computational power of the Cell/B.E. could be used to further improve these results too, leading to a smaller circuit that can be calculated in less time.

One last approach that might be worth investigating is using a different matching approach. At the moment each SPE calculates one part of the circuit sequentially. Investigating further parallelization concepts that might exploit the hardware better than the current implementation and a comparison between these approaches is a topic for future work too.

Chapter 6

Conclusion

The overall results, demonstrated in Chapter 4, 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 and is not for free. 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. But 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 haven't been computable in acceptable time beforehand.

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.

Bibliography

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

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

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

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

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

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

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

[8] H. P. Hofstee. (2005, May) Introduction to the Cell Broadband Engine. White Paper. Accessed: 2010-05-04. [On-

line]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D/$file/2053_IBM_CellIntro.pdf

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

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

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

[12] E. O. D. Sevre, M. D. Christiansen, M. Broten, S. M. Wang, and D. A. Yuen, "Experiments in scientific computation on the playstation 3," *Visual Geosciences*, vol. 13, no. 1, pp. 125–132, July 2008.

[13] IBM. Ibm bladecenter qs21. Accessed: 2010-05-04. [Online]. Available: http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs21/index.html

[14] Sony Computer Entertainment Inc. (2009, August) New slimmer and lighter playstation 3 to hit worldwide market this september. Press Release. Accessed: 2010-05-04. [Online]. Available: http://www.scee.presscentre.com/content/Detail.asp?ReleaseID=4842&NewsAreaID=2

[15] (2010, April) Ps3 firmware 3.21 coming april 1st. Official Playstation 3 Blog. Accessed: 2010-05-04. [Online]. Available: http://blog.eu.playstation.com/2010/03/29/ps3-firmware-3-21-coming-april-1st/

[16] J. Bartlett. (2007, January) Programming high-performance applications on the cell be processor, part 1: An introduction to linux on the playstation 3. Accessed: 2010-05-04. [Online]. Available: http://www.ibm.com/developerworks/library/pa-linuxps3-1/

[17] IBM. Cell broadband engine resource center. Accessed: 2010-05-04. [Online]. Available: http://www.ibm.com/developerworks/power/cell/

[18] IBM Redbooks, *Programming the Cell Broadband Engine Architecture: Examples and Best Practices.* Vervante, 2008.

[19] REVLIB - The Online Resource for Reversible Functions and Circuits. Accessed: 2010-05-04. [Online]. Available: http://revlib.org/

[20] ISO, “The ANSI C standard (C99),” ISO/IEC, Tech. Rep. WG14 N1124, 1999. [Online]. Available: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf

[21] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, “RevLib: An Online Resource for Reversible Functions and Reversible Circuits,” in *ISMVL ’08: Proceedings of the 38th International Symposium on Multiple Valued Logic*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 220–225.