Synthesizing Software from a ForSyDe Model Targeting GPGPUs

Master of Science Thesis

Gabriel Hjort Blindell

ROYAL INSTITUTE OF TECHNOLOGY

Stockholm, January 2012

TRITA-ICT-EX-2012:13

Examiner
Assoc. Prof. Ingo Sander
KTH/ICT/ES

Supervisor
Dipl. Inf. Christian Menne
KTH/ICT/ES

Author: Hjort Blindell, Gabriel <ghb@kth.se>
Title: Synthesizing Software from a ForSyDe Model Targeting GPGPUs
Thesis number: TRITA-ICT-EX-2012:13

Royal Institute of Technology (kth)
School of Information and Communication Technology (ict)
Research Unit: Electronic Systems (es)
Forum 105
164 40, Kista
Sweden

This document was typeset in LaTeX, with *Kp-Fonts* as font package. All figures were produced, either partly or entirely, using the *TikZ* package.

Document build: 2012-02-20 16:11:42

Abstract

Today, a plethora of parallel execution platforms are available. One platform in particular is the GPGPU – a massively parallel architecture designed for exploiting data parallelism. However, GPGPUs are notoriously difficult to program due to the way data is accessed and processed, and many interconnected factors affect the performance. This makes it an exceptionally challenging task to write correct and high-performing applications for GPGPUs.

This thesis project aims to address this problem by investigating how ForSyDe models – a software engineering methodology where applications are modeled at a very high level of abstraction – can be synthesized into CUDA C code for execution on NVIDIA CUDA-enabled graphics cards. The report proposes a software synthesis process which discovers one type of potential data parallelism in a model and generates either pure C or CUDA C code. A prototype of the software synthesis component has also been implemented and tested on models derived from two applications – a Mandelbrot generator and an industrial-scale image processor. The synthesized CUDA code produced in the tests was shown to be both correct and efficient, provided there was enough computation complexity in the processes to amortize the overhead cost of using the GPGPU.

Keywords: ForSyDe, abstract program models, software synthesis, GPGPU, CUDA, C

Acknowledgments

During the course of this thesis project, several persons have contributed one way or another, and to whom I owe my gratitude.

I wish to thank my supervisor, Christian Menne, for his encouragements and keeping the project on track and within scope. Among other things, his support has been invaluable in helping me understand how ForSyDe works.

I wish to thank Ingo Sander for his continuous interest and input on the project. His inquiries have given rise to many contemplations that has helped to drive the project forward, especially in writing the report.

I wish to thank Seyed Hosein Attarzadeh Niaki for our discussions about the implementation of the component. Although I would probably have been able to solve many of the problems on my own, it would doubtlessly have taken much longer.

I wish to thank Christian Schulte for asking about the support for exploiting reduction data parallelism – a type which I had not previously considered.

Last but not least, I wish to thank everyone at *TEX StackExchange*[1] for helping me with the problems I had of writing this report and producing the figures therein. Without them the material would have been much less accessible.

Gabriel Hjort Blindell
Stockholm, January 2012

[1]http://tex.stackexchange.com

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSL	Domain-Specific (programming) Language
DPFP	Double-Precision Floating Point
ForSyDe	Formal System Design
FPU	Floating Point Unit
GPGPU	General-Purpose GPU
GPU	Graphics Processing Unit
GraphML	Graph Markup Language
ILP	Instruction-Level Parallelism
SDF	Synchronous Data Flow
SDK	Software Development Kit
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Threads
SM	Streaming Multiprocessor
SP	Streaming Processor
SPFP	Single-Precision Floating Point
VHDL	VHSIC HDL
VHSIC	Very-High-Speed Integrated Circuit
XML	eXtensible Markup Language

CHAPTER 1

Introduction

This chapter presents the motivation behind this thesis, its objectives and the strategic approach to the thesis problem. The chapter closes with an overview of the document structure of the report.

1.1 MOTIVATION

In the last few decades we have seen tremendous advances in microelectronics. Faster and denser chips allow more complex systems which puts an ever-increasing demand on system designers to consider low-level details and complexity management. Moreover, with chips reaching their clock rate limit, multicore platforms are favored to a greater extent than ever. Future systems are thus required to be parallelized in order to make efficient use of the underlying architecture. This entails intricate communication and synchronization schemes that further exacerbates the design process. Consequently, this also leads to more effort and resources being put into testing and verification to ensure that systems are correct and efficient. At the same time aggressive market competition forces companies to limit the development cycles for their products.

Recognizing these challenges, Stephen Edwards et al. and Kurt Keutzer et al. advocate modeling systems on as high an abstraction level as possible in order to contain the complexities of system design [18, 31]. This is achieved, they argue, by applying system modeling methodologies which promote component reuse, verification and early error detection, which in turn yields rapid development. However, this also creates an abstraction gap between the model level and the implementation level. One such methodology which attempts to bridge this gap is ForSyDe.

ForSyDe is a work in progress actively under development by the Department of Electronic Systems (ES) of the School of ICT at the Royal Institute of Technology (KTH), Sweden. A research project conducted by the same department explores the capabilities of modeling and synthesizing highly parallel systems in high abstraction models such as ForSyDe. As part of its research, an attempt is made to model and synthesize a streaming application for image processing. The application is based on an existing full-scale industrial program provided by XaarJet AB, a company specializing in piezoelectric drop-on-demand ink-jet printing. Due to the nature of its functionality and high throughput demand, the application seems to be a good candidate for parallel execution on a throughput-oriented architecture, e.g. a GPU. It was thus decided to extend ForSyDe with a software synthesis component which enables system models to be synthesized into parallel C code optimized for execution on such platforms. The development of that component is the assignment of this thesis project.

1.2 OBJECTIVES

The main goal of this thesis is to investigate and realize the implementation of a software synthesis component for ForSyDe supporting optimized code generation targeting GPGPUs. The component shall use a file containing the graph representation of the system model as input. As it would be beyond the scope of a M.Sc. thesis to consider a fully generic tool, the component only needs to support a subset of the standard operations from the image processing domain and ForSyDe constructs, and only needs to target GPGPUs as execution platform. Thus the objectives are defined as follows:

1. Perform a literature study, targeting the following areas:
   - ForSyDe (also includes a brief study of Haskell and SystemC),
   - synchronous data flow models,
   - GPGPU architecture,
   - parallel programming using CUDA,
   - memory topologies,
   - compilation techniques,
   - software synthesis with related work, and
   - optimization techniques for mapping GPGPU resources.
2. Determine the minimum set of language constructs and operations that must be supported by the software synthesis component in order to successfully synthesize the application model.
3. Identify main challenges of implementing the software synthesis component (e.g. how to identify computational workload eligible for parallelizing, how to distribute such workload on the GPU for efficient execution, etc.).

4. Prioritize and select which challenges in objective 3 are to be addressed in this thesis.
5. Develop solutions to selected challenges.
6. Implement a prototype of the software synthesis component which accepts a file containing the application model represented in an extended GraphML format as input, and outputs CUDA-annotated C code optimized for execution on a GPGPU.

1.3 STRATEGY

The thesis project is divided into four phases:

1. Literature study, wherein necessary knowledge of related work and background information is gathered. This also includes identifying the main challenges hinted at in objective 3.
2. Development and implementation of a software synthesis component prototype which accepts a GraphML representation of a ForSyDe model but outputs sequential C code only. This phase obviously also involves the conceiving of necessary methods and algorithms for tackling the challenges identified in phase 1.
3. Iteratively enhance the prototype to enable parallel execution by annotating the generated code with CUDA directives as well as optimizing the output for execution on GPGPUS.
4. Verify the functionality of the software synthesis component through experimentation.

Phase 2 and 3 are separated in order to manage complexity and always have a component which works to some extent.

1.4 DOCUMENT OVERVIEW

The report is divided into four parts: Part I covers the problem background and necessary information in order to understand the problem; Part II describes the development and implementation of the software synthesis component; Part III closes the report by discussing future work and conclusions; and Part IV contains the appendices.

1.4.1 *Part I*

The aim of the first part is to provide the reader with enough knowledge in order to understand the problem itself and the challenges related in addressing it. Chapter 2 briefly introduces the reader to ForSyDe and how it is used to model applications. Chapter 3 describes alternative two domain-specific programming languages – Obsidian and SkePU, both with backends for code synthesis – and how they compare with ForSyDe. Chapter 4 explains GPGPUS and briefly covers how to write and optimize data parallel programs for execution on CUDA-enabled graphics cards. Chapter 5 describes the format used for representing the ForSyDe models in text files, and Chapter 6 lists the challenges in developing and implementing the software synthesis component. If the reader is already well-versed with these topics, the chapters in question can safely be skimmed or skipped entirely.

1.4.2 *Part II*

The second part concerns the design and implementation of the software synthesis component. Chapter 7 first describes the application which will be used as base for proof-of-concept and test modeling, and continues with analyzing how to transform this model into a CUDA C program. Chapter 8 explains the applied methods and algorithms, followed by Chapter 9 which covers the software synthesis component in detail. Chapter 10 lists the limitations of the implemented component, and Chapter 11 analyzes the results and performance of the synthesized CUDA C code.

1.4.3 *Part III*

The third part closes the report. Chapter 12 suggests future work by recommending non-implemented features and research topics. A summary of the report is given in Chapter 13, where the work is checked against the thesis goals. The chapter also suggests a set of new process types that should be added to the ForSyDe framework to improve and simplify modeling and synthesis of data parallel applications targeting GPGPUS.

1.4.4 *Part IV*

The fourth and last part contains the appendices. Appendix A holds the documentation of the software synthesis component and how to use it. It also describes how to maintain the component, in case future developers wish to improve upon it or add new features.

I

Understanding the Problem

CHAPTER 2

ForSyDe

This chapter introduces ForSyDe to the uninitiated. It briefly explains the methodology and how to write simple models. Note, however, that this chapter only covers ForSyDe in just-enough detail for the reader to understand the application model; for a thorough documentation of ForSyDe, please consult [40–42]. The chapter also contains code examples based on Haskell and SystemC, and only some of the language constructs will be explained. For introductory texts, the reader is advised to consult [36] and [6], respectively; covering any of these languages in detail is out of scope for this report.

The text in this chapter is primarily based on the material found in [40–43].

2.1 A BRIEF INTRODUCTION

Keutzer et al. states that in order "to be effective, a design methodology that addresses complex systems must start at high levels of abstraction" [31]. One such methodology is ForSyDe (Formal System Design), which primarily targets modeling of systems-on-a-chip and embedded systems.

The main objective of ForSyDe is "to move design refinement from the implementation into the functional domain" [42]. It does this by capturing the system functionality in a *specification model*. The specification model operates at a high level of abstraction which hides implementation-oriented details, thus allowing the designer to focus on *what* the system is supposed to do rather than *how*. Moreover, as the model is founded on formal methods, its correctness can be verified using automated tools. This reduces both cost and development time as errors detected in the post-release phase can be 10 to 25 times more expensive to correct than had they been revealed in the construction phase [34].

Once designed and tested, the specification model can be refined into an *implementation model* using *semantic-preserving transformations* and *design decisions* (the refinement process is out of scope for this thesis and will thus not be covered in any detail). Semantic-preserving means that the transformations do not change the meaning of the model. Hence, if applied on a specification model which is proven to be correct, then the resultant implementation model will also be correct, thus allowing systems to be "correct by construction" [18]. The refinement process stops when the model contains enough details to be synthesized into (or mapped onto) an implementation. The synthesis process may produce either hardware (e.g. VHDL code) or software (e.g. C code), depending on the backend support.

ForSyDe was originally implemented as a library in functional programming language Haskell. This was due to several reasons:

- The natural match between functional languages and data flow applications advocates using a functional programming language as base for a *domain-specific language* (DSL) [45].
- Haskell is based on formal semantics and is purely functional (i.e. functions produce no side-effects) and thus yields completely deterministic models [18]. This in turns aids verification.
- Haskell uses a lazy evaluation mechanism which allows convenient structures such as infinite lists to model signals [41].
- Haskell supports powerful language constructs such as pattern matching and higher-order functions [43].
- Haskell has a strong type system with automatic inference, which provides great flexibility while at the same time facilitating verification as bugs related to type matching errors can be detected at compile time [41, 43].

During the course of this thesis project, ForSyDe was also being implemented in *SystemC*. However, as its GraphML backend was not yet operational, the GraphML files used in this work were either generated from Haskell ForSyDe models and its process function arguments were manually translated into C, or the files were hand-written from scratch. Nonetheless, the goal is to have a completely automated process from model design to code synthesis.

2.2 SYSTEM MODELING

As previously stated, the ForSyDe methodology provides two kinds of system models: a specification model and an implementation model. For this work we are only interested in the former as the input to the software synthesis component will be derived from the specification model. This model will from hereon be referred to as the *ForSyDe model* or *system model*.

A system model consists mainly of *signals* and *processes*. For the purpose of this thesis, we simplify the definition of a signal $\vec{s}$ and view it as an infinite sequence of values:

$$\vec{s} = \langle v_1, v_2, \ldots \rangle$$

A sequence may also contain *absent* values denoted by $\perp$. The ability to signify absent values is needed in order to model systems where some parts operate at slower data rate speeds than others. Hence we may have signals such as

$$\vec{s} = \langle v_1, \perp, v_2, \perp, \perp, v_3, v_4, \ldots \rangle$$

A process P is defined as a functional mapping of m input signals $\vec{i}_1, \ldots, \vec{i}_m$ into n output signals $\vec{o}_1, \ldots, \vec{o}_n$:

$$P(\vec{i}_1, \ldots, \vec{i}_m) = (\vec{o}_1, \ldots, \vec{o}_n)$$

A model is thus a *hierarchical network of processes* which communicate via signals. Figure 2.1 shows an example of a process network PN, which itself is also a process and can be expressed as:

$$PN(\vec{i}) = (\vec{o})$$

where

$$\begin{aligned} (\vec{s}_1, \vec{s}_2) &= P_1(\vec{i}) \\ \vec{o} &= P_2(\vec{s}_1, \vec{s}_3) \\ \vec{s}_3 &= P_3(\vec{s}_2) \end{aligned}$$

ForSyDe provides several *models of computation* [18, 31], but in this work we will only consider the *synchronous model of computation* for which there is currently most extensive support. In this model, systems are assumed to operate under the *perfect synchrony hypothesis* [5] where processes are "infinitely fast" [42] and signal propagation takes zero time. Output values from a process, and even for the entire model, are thus produced immediately after (or as) the input values arrive. This eliminates the possibility of race

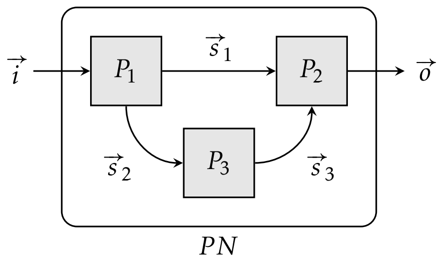
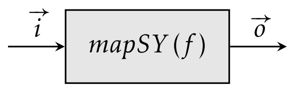


FIGURE 2.1 – Example of a ForSyDe model, which can be viewed as a network of concurrent processes. *Source: adapted from [41].*

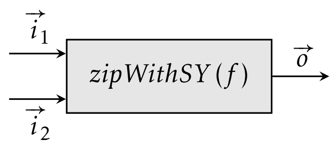
conditions (i.e. that one value will arrive before another), which reduces the number of possible states and leads to deterministic system models.

Processes are created using *process constructors* provided by the ForSyDe library. Each constructor is a higher-order function which takes *combinatorial functions* (i.e. functions with no internal state) and input signals as parameters and produces a process as output. A combinatorial function is also called a process' *function argument*. This "cleanly separates" [42] communication and computation and leads to coherent and well-defined models. In addition, all process constructors have structural *hardware and software semantics* which allow the implementation model to be synthesized into corresponding hardware or software. Figure 2.2 illustrates some of the process constructors and processes available in the ForSyDe library. The entity names (e.g. *mapSY*, *zipWithSY*) have been chosen to reflect their similarity with the Haskell library functions (*map*, *zipWith*). There is also a generic *zipWithNSY* which can take an arbitrary number of input signals. Hence, a *zipWithNSY* process with one input signal is equivalent to a *mapSY* process, and a *zipWithNSY* process with two input signals is equivalent to a *zipWithSY* process.

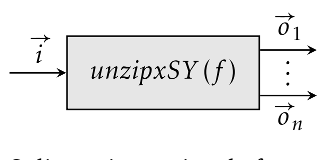
Lastly, the "SY" suffix indicates that the entities are synchronous. As we will only be dealing with the synchronous computational model, we assume from hereon that all processes and process constructors are synchronous unless otherwise stated.



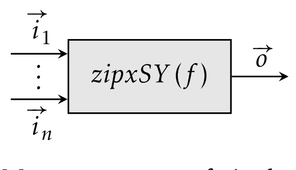
(a) Applies a combinatorial function f on every value on the input signal.



(b) Applies a combinatorial function f on every pair of values (one from each input signal).



(c) Splits an input signal of array values into an array of single-value output signals.



(d) Merges an array of single-value input signals into an output signal of array values.

FIGURE 2.2 – ForSyDe process constructors *mapSY*, *zipWithSY*, *unzipxSY*, and *zipxSY*. *Source: adapted from [42].*

2.3 MODELING IN HASKELL

Declaring a ForSyDe process is akin to declaring a function in Haskell. Listing 2.1 shows the declaration of an *adder* process which adds the values of two input signals. Using the Haskell function compositor operator (.), the output of one process can easily be connected to the input of another. For example, in Listing 2.2) three processes f, g and h are connected one after another (also see Figure 2.3). The process order may appear reversed compared to the declaration, but this is just due to the declaration using nested function calls (think of it as $h(g(f()))$, where f will be evaluated first). Also, the network is declared using point free style where the right-most function parameter may be omitted, thus allowing a more compact style of coding.

2.3.1 *Shallow and deep models*

ForSyDe provides two methods of modeling: *shallow* models and *deep* models [1]. The modeling concepts are the same for both, but differ in terms of ease-of-use and backend support. The shallow model permits usage of all Haskell language constructs but can only be simulated. Shallow models are therefore only suitable for rapid prototyping and experimentation. Deep models are more difficult to write as they are more restrictive in terms of language constructs; the programmer is limited to a small set of data types, and is not allowed to use infinite lists, pattern matching or combinatorial recursion.

```haskell
adderSY :: (Num a) => a -> a -> a
adderSY in1 in2 = zipWithSY (+) in1 in2
```

LISTING 2.1 – Declaration of a Haskell ForSyDe *adder* process.

```haskell
programSY :: a -> a
programSY = mapSY h . mapSY g . mapSY f
```

LISTING 2.2 – Connecting several Haskell ForSyDe processes with combinatorial functions f, g and f.

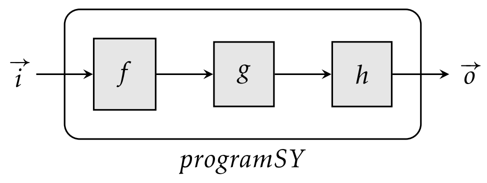


FIGURE 2.3 – Illustration of the program modeled in Listing 2.2.

However, the advantage of deep models is that they can be synthesized to VHDL or GraphML code (or whatever backend is available).

2.4 MODELING IN SYSTEMC

The theory behind application modeling in the SystemC flavor of ForSyDe is the same as in Haskell. The main difference is, obviously, the language constructs – Haskell is a purely functional language while SystemC is a template library implemented in C++, an object-oriented language. In SystemC, process constructors are implemented as classes, and processes are created by deriving the appropriate process constructor class and providing implementations for the pure virtual functions. Once declared, the processes are instantiated as objects which are used to create modules, which in turn are attached to form a network using channels. Channels are the communication elements of SystemC and can be either something as simple as a wire or as complex as a FIFO queue or a bus. Listing 2.3 shows an equivalent implementation of the *adder* process which was declared in Haskell in Listing 2.1.

```
#include "forsyde.h"

using namespace ForSyDe::SY;

class adderSY : public comb2<int, int, int> {
  public:
    adderSY(sc_module_name _name)
        : comb2<int, int, int>(_name) {}
  protected:
    int _func(int a, int b) {
      return a + b;
    }
};
```

LISTING 2.3 – Declaration of a SystemC equivalent of the Haskell *adder* process.

CHAPTER 3

Alternative DSLs

This chapter introduces the reader to two alternative domain-specific programming languages – Obsidian and SkePU – designed either completely or partly for writing applications which targets execution on GPGPUs. Both languages provide backends for synthesizing their models into optimized CUDA C code and the goal of this chapter is to analyze how they compare with ForSyDe and whether the same synthesis solutions and methods can be applied for the ForSyDe component.

3.1 Obsidian

The text in this section is primarily based on the material found in [46, 47].

Obsidian is a domain-specific programming language for programming data parallel applications for execution on GPGPUs (specifically CUDA platforms). It is a work in progress developed by the Department of Computer Science and Engineering at Chalmers University of Technology and Göteborg University with the goal to offer a tool which “encourages experimentation” by “rais[ing] the level of abstraction of GPU programming” [46].

3.1.1 *Implementation and usage*

Implemented as an embedded language in Haskell[1], Obsidian consists of:

- a collection of scalar type operations;
- arrays together with library functions to operate on these arrays; and
- a collection of operations called combinators which construct the GPGPU kernels.

[1] Obsidian has been implemented twice, the first approach based on monads and the second based on arrows.

An array is of type `Arr a` and may consists of types `IntE`, `FloatE` or `BoolE`. All arrays are of finite length and an Obsidian programmer uses these arrays to appropriately package the computation data.

Once the data arrays have been formed, the programmer uses the array library functions to construct the functionality of the GPGPU kernels. In a sense, these functions are the program building blocks; most functions are simply Obsidian versions of the Haskell library functions (e.g. *map*, *foldr*, *zip*, etc.) with the restriction that they only operate on Obsidian arrays. Listing 3.1 shows an example of where *fmap* is used to declare a function which increments each value in an array. Obsidian also provides a set of permutation functions, motivated by their usefulness in sorting networks – *rev*, *riffle*, *unriffle*, to name a few.

From the array functions, the programmer can construct the GPGPU kernels using *combinators*. In Obsidian, a kernel is represented by a data type `a :-> b` which can be interpreted as a program which inputs `a` and outputs `b`. Two of the most common combinators are *pure* and *sync*, defined as

```
pure :: (a -> b) a :-> b
```

and

```
sync :: Flatten a => Arr a :-> Arr a
```

respectively (`Flatten` declares types that can be stored in GPU memory). The *pure* combinator turns one or more array functions into a kernel, and the *sync* combinator is used between kernels to signify that the data is to be intermediately stored in shared memory (will be explained in Section 4.2.5). The *sync* also includes synchronization barriers when necessary (see Figure 3.1). For some combinators the performance of the generated CUDA code is "satisfactory" [47] and comparable with hand-optimized code, while for others the performance is not yet sufficient.

Using this approach, the programmer does not need to be concerned with figuring out the intricate array indexing schemes that naturally appear when programming GPGPUs. Instead, these are taken care of by Obsidian as part of the synthesis process. Obsidian also manages usage of the shared memory; the developer simply has to say when and where to use it.

```
incr :: Arr a -> Arr a
incr = fmap (+1)
```

LISTING 3.1 – Declaration of an Obsidian function *incr*.

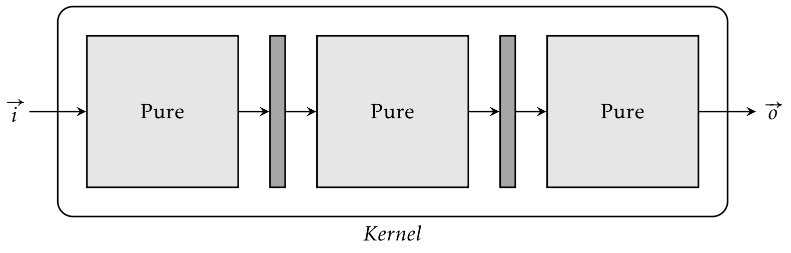


FIGURE 3.1 – Illustration of a GPGPU kernel modeled in Obsidian using *pure* and *sync* combinators. *Source: adapted from [46].*

3.1.2 *Similarities with ForSyDe*

Due to Obsidian and ForSyDe both being implemented as embedded languages in Haskell, the way of writing (or modeling) programs in each are congruent in nature, especially since both languages prohibit global state and thus requires that all functions are purely functional.

Another similarity is Obsidian's array library functions which are akin to ForSyDe's process constructors: they take higher functions as arguments; can be daisy-chained using the Haskell function composition function (.); and have software semantics which allows them to be synthesized into an implementation.

3.1.3 *Differences from ForSyDe*

A key difference between Obsidian and ForSyDe is their intention. Obsidian focuses on absolving the developer from having to deal with low-level details such as kernel invocation, thread count and block size decisions, array indexing and facilitates the process of combining kernels into larger kernels. To wit, Obsidian is a language for simplifying GPGPU programming by taking care of the mechanical work while leaving the creative bits to the programmer. In contrast, ForSyDe concentrates purely on functionality and correctness and abstracts away the underlying execution platform entirely. This means that implementation-specific details such as synchronization barriers are not provided by the programmer in the ForSyDe model, but must be analyzed and inserted appropriately by the CUDA synthesis backend.

Since Obsidian targets a specific architecture, it provides matching input/output data structures and library functions and forces the developer to adapt his algorithms to fit them. ForSyDe, being more general, simply does not have the luxury to customize its library for a particular platform. Hence a program written in Obsidian is probably more accessible for CUDA optimizations and synthesis than a ForSyDe model will be. For instance, as input and output to its kernel-generating combinators Obsidian uses arrays, which coincide with GPGPUs' method of computation. ForSyDe, on the other hand, propagates data through signals which may have to be aggregated in order to provide enough data for execution on GPGPUs to be worthwhile.

Another important difference is that Obsidian requires that the programmer explicitly inserts *sync* calls on appropriate places to trigger usage of shared memory. Hence Obsidian relies on the programmer to write programs which correctly exploits this feature. ForSyDe does not provide any such constructs, which leaves the responsibility of finding out when, where and how to capitalize on the shared memory to the software synthesis component.

3.2 SKEPU

The text in this section is primarily based on the material found in [15–17, 19, 20].

SkePU is a skeleton programming framework targeting heterogeneous platforms under development by the Department of Computer and Information Science at Linköping University. Like Obsidian, SkePU raises the abstraction level by facilitating the process of writing data parallel applications. However, it is general enough to support other architectures than CUDA and strives to make its program portable without having to rewrite the code. At the point of writing, SkePU provides code generating backends for multi-GPU CUDA, OpenCL, OpenMP and pure sequential CPU execution, and is even capable of utilizing several backends simultaneously. In addition, SkePU uses a *lazy memory copying* technique where the actual copying of data from GPU memory to CPU memory is only done when the data is needed, thereby avoiding unnecessary memory transfers.

3.2.1 *Implementation and usage*

SkePU is implemented as a C++ template library using StarPU as underlying run-time system (see [17] for information about StarPU). It applies a notion called *skeleton programming*, where the application building blocks consist of predefined *skeletons* which take higher order functions as input[2]. Currently, SkePU provides six skeletons – *Map*, *MapReduce*, *MapOverlap*, *MapArray*, and *Scan* – which each provides software semantics for the synthesis backends and thus enables portability (although functionality not provided through any combination of skeletons must be still be manually rewritten).

The skeletons are represented as singleton objects. When a skeleton is instantiated, an execution environment containing all available OpenCL and CUDA devices is created. Since all skeleton objects are singletons, this environment is shared between all instances within the program. All skeleton object

[2]The astute reader will note the similarity between how ForSyDe and SkePU models their programs; in fact, the ForSyDe process constructors were even called *skeletons* in earlier work (see Sander1999). However, since higher order functions are not supported in C++ as in Haskell, function pointers are used instead.

also overload the *operator()* construct, enabling them to behave like functions which, when invoked, selects an appropriate implementation-specific member function to execute. Which member function is selected depends on device availability and problem size together with the current execution plan (more on this later). The skeletons also perform architecture-specific optimizations, such as utilization of shared memory on CUDA platforms.

Before a skeleton can be created, a *user function* must be provided. A programmer declares a user function through preprocessor macros which expand the function into a `struct` containing the backend-specific implementations. Listing 3.2 shows the declaration and creation of a *Reduce* skeleton which implements the a+b function, with the corresponding macro expansion shown in Listing 3.3.

So far SkePU also provides two generic data containers – `Vector` and `Matrix`, based on `std::vector` – on which the skeletons operate. Apart from holding the data itself, the data containers also implicitly manage data transfers between the CPU and GPU memory and keep track of multiple copies residing in different memories. As previously noted, SkePU applies a lazy memory copying technique which delays the actual copying as long as possible. This is beneficial when applying a series of GPU computations on the same container as data will not be transferred back to the CPU memory between the invocations.

The execution time of a skeleton is affected by three factors:

- the problem size,
- the selected backend, and
- the parameter set for that backend.

In general, the CPU and OpenMP backends run faster on smaller problems compared to OpenCL and CUDA, and vice versa. Furthermore, performance of

```
BINARY_FUNC(plus, double, a, b,
  return a+b;
)

// Create a reduction skeleton from the plus operator
skepu::Reduce<plus> globalSum(new plus);

skepu::Vector<double> input(100, 10);

// Apply sum reduction on the input vector
double sum = globalSum(input);

// Apply sum reduction explicitly using the CPU backend
double sum = globalSum.CPU(input);
```

LISTING 3.2 – Skeleton creation in SkePU. *Source: [20].*

```
BINARY_FUNC(plus, double, a, b,
  return a+b;
)

// expands to

struct plus
{
  skepu::FuncType funcType;
  std::string func_CL;
  std::string funcName_CL;
  std::string datatype_CL;
  plus()
  {
    funcType = skepu::BINARY;
    funcName_CL.append("plus");
    funcName_CL.append("double");
    datatype_CL.append(
    "double␣plus(double␣a,␣double␣b)\n"
    "{\n"
    "␣␣␣return␣a+b;\n"
    "}\n");
  }
  double CPU(double a, double b)
  {
    return a+b;
  }
  __device__ double CU(double a, double b)
  {
    return a+b;
  }
}
```

Listing 3.3 – Macro expansion of user function in SkePU. *Source: [20].*

a particular backend can be affected by several parameters, such as number of threads for the OpenMP backend and grid and thread block sizes for the CUDA backend. In SkePU, these parameters are called the *parameter set*. Hence it would be beneficial if the backend and parameter set was selected at runtime. This is addressed by execution plans.

An *execution plan* maps a problem size for a particular skeleton to the backend and parameter set which offers the most optimal performance. Since all skeletons operate on vectors and matrices, the problem size is defined as the size of the input vector or matrix. The execution plans can be set manually, computed offline using machine learning and training data, or tuned at runtime.

To conclude, by applying these skeletons and data containers, the programmer is insulated from any architecture implementation details. Furthermore, backend-specific optimizations are taken care of by the skeleton and memory transfers between the CPU and GPUs are efficiently managed by the data containers provided by SkePU.

3.2.2 *Similarities with ForSyDe*

Like Obsidian and ForSyDe, SkePU wraps user-defined functionality in software semantic entities which can be understood and synthesized by the backends. Moreover, both SkePU and ForSyDe targets no specific backend and therefore must have an application modeling methodology which is general enough to implement models on all architectures. This means that the synthesis backends cannot rely on the programmer to declare operations such as synchronization barriers in the application models as this backend dependent. This allows kernel translation and optimization techniques implemented in SkePU to be analyzed and adapted to the ForSyDe software synthesis component.

3.2.3 *Differences from ForSyDe*

While SkePU was not developed targeting a specific execution platform, it does aspire to alleviate data parallelization and thus uses, like Obsidian, vectors and matrices as data containers. These containers match this purpose and thus need not be altered or adapted for the synthesis process. In ForSyDe, data may have to be aggregated across signals in order to provide enough data parallelism for offloading computations on the GPGPUs to be productive. Hence, the data containers have to be adapted to the signal data type and the pattern of the data parallelism being exploited in the model, which can only be done during synthesis. In that sense, ForSyDe operates at a higher level of abstraction compared to SkePU as it does not restrict the system designer to use a limited set of data containers. However, this means that the software synthesis component has to construct appropriate data containers which can be used efficienctly by the GPU kernels.

An important difference between SkePU and ForSyDe, even when considering the SystemC flavor, is that ForSyDe prohibits any global state (this is for good measure as that is essential for determinism). SkePU, on the other hand, capitalizes on this by making all skeleton objects into singletons. This allows the same execution environment to be shared among the skeletons, and encourages the programmer to write applications under this assumption. The same approach cannot be made directly for ForSyDe models. (It may be possible, however, that this could be applied in the synthesis process as the component will generate C code, although the given ForSyDe models may not exactly promote this approach.)

CHAPTER 4

GPGPUs and General Programming with CUDA

This chapter explains the background of GPGPUs, their architecture and a short introduction in how to write parallel C programs for NVIDIA's CUDA platform. Understanding the underlying hardware covered in Section 4.2 is essential for comprehending the code optimizations explained in Section 4.4, and consequently also for appreciating many of the challenges in implementing a software synthesis component which generates efficient CUDA C code. The chapter closes with comparing CUDA to two alternative GPGPU programming frameworks.

4.1 BACKGROUND

Propelled mainly by the gaming industry, graphics cards filled the demand of processing units needed for rendering complex, high-resolution 3D scenes at real-time [35]. Starting with the first GPU –NVIDIA's GeForce 256 released in 1999 – the number of transistors has increased by three magnitudes in little over 10 years, reaching 1.03 single-precision teraflops in 2009 [14, 32].

Recognizing this massive computing power, pioneering developers attempted to utilize the GPUs for tasks other than processing graphics [27, 35, 44]. This required, however, that the programmers expressed non-graphical computations through the graphics API and shader languages, which needless to say was less than ideal. In addition, floating-point arithmetic was dubious, if even supported at all.

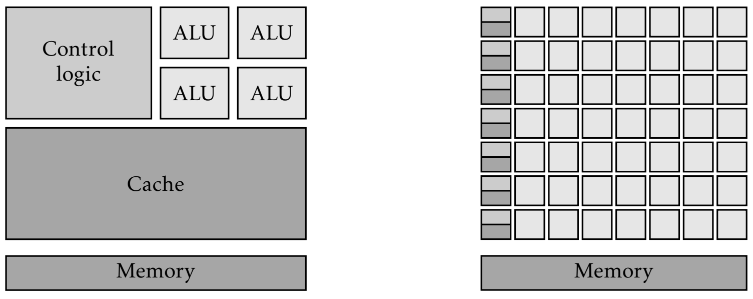
Greater demands were made on the GPU to provide more general vertex and pixel-fragment processors, which increased design complexity, area and costs. In addition to increased inefficiencies due to poor load-balancing between the two, GPU manufacturers opted that the processors be unified into a single entity [33]. Furthermore, the vector processors were switched to scalar thread processors, and instruction support for integer and IEEE 754 floating-point arithmetic, load/store memory access with byte addressing, and synchronization barriers was added. This allowed the GPU to be programmed using general-purpose languages such as C, and the advent of the GPGPU was marked in 2006 when NVIDIA released GeForce 8800 – the first unified graphics and computing GPGPU architecture [35].

4.2 ARCHITECTURE

The most widely used technique for rendering images is to use geometric primitives, such as triangles, which are then processed through several stages. At each stage, individual triangles, triangle corners or pixels are operated upon independently [23]. As there easily can be millions of pixels for a single frame, graphical computations is thus inherently a *data parallel* process and consequently yields a fundamentally different architecture compared to a CPU (see Figure 4.1) [23, 32, 35, 44]. As they have been optimized for completely different tasks, the GPU is a complement to the CPU rather than a replacement.

4.2.1 *Many simple processing units*

The CPU has been designed under the assumption that its programs will be largely sequential in nature, leading to a *latency-oriented architecture*. Traditional scalar microprocessors are optimized for single-thread execution by dedicating a considerable amount of chip area to sophisticated control logic such as out-of-order execution and speculative branch prediction in order to minimize delay.



(a) CPU (b) GPU

Figure 4.1 – An architectural comparison between a CPU and a GPU – two fundamentally different design philosophies. The CPU has been optimized for execution of sequential code, while the GPU has been optimized for execution of data parallel code. *Source: adapted from [32].*

The GPU, on the other hand, has been designed assuming that there will be plenty of parallelism and is therefore a *throughput-oriented architecture*. Such platforms strive to include as many computing cores as possible by avoiding advanced control logic, meaning that the cores typically execute their instructions in order, although in a pipelined fashion.

4.2.2 *Hardware multithreading*

Processing an image element such as a triangle or pixel generally involves launching a thread which executes a small program – usually called a *shader* [23], and in CUDA it is called a *kernel function* (these are explained in Section 4.3.2). In order to keep the processing cores busy while waiting for long latency operations such as memory accesses, GPGPUs apply *hardware multithreading* to make thread switches with extremely fine granularity, often at instruction-level [23, 32, 35]. A modern GPGPU therefore manages tens of thousands of concurrently existing threads with minimal overhead. For example, a Tesla C2050 executing a simple kernel function which increments one individual value per thread will spawn, execute and retire about 13 billion threads per second [23].

4.2.3 *SIMD execution*

Parallel processors often employ some form of *single-instruction, multiple-data (SIMD) execution* to improve throughput, either by issuing the same instruction to multiple cores or by using a vector processor which execute a single instruction over a vector of data. Such execution is attractive as more resources can be devoted to functional units rather than control logic.

Since each image rendering stage involves executing the same shader on multiple image elements, it is clear that SIMD execution will also benefit GPUs, and this is often applied. As we will see shortly in Section 4.2.6, NVIDIA's CUDA platform applies a slightly different form of SIMD execution.

4.2.4 *Calculation is cheap*

In GPGPUs, computation generally costs much less than memory transfers, in particular transfers to and from off-chip DRAM memory. A computation operation, such as an add, only takes a few cycles, while a memory read from DRAM can take several hundred cycles. Moreover, much more energy is required to transfer data between the cores and DRAM memory than required to operate a functional unit. For instance, in a 45 nm circuit, a 64-bit addition unit consumes about 1 pJ (picojoule), and a 64-bit floating point multiply-add unit requires about 100 pJ, while reading a 64-bit data value from external DRAM absorbs 2,000 pJ [23].

It may therefore be favored to recompute values such as $\sin x$ and $\cos x$ instead of looking up precomputed values in a table residing in memory, especially when there are functional units on board dedicated for computing fast approximations.

4.2.5 *The NVIDIA CUDA platform*

The text in this section is primarily based on the material found in [23, 24, 27, 32, 33, 35].

A typical CUDA GPGPU is illustrated in Figure 4.2. Roughly speaking, the GPGPU can be divided into *clusters*, each consisting of a pair of *streaming multiprocessors* (SM). Each SM in turn contains a large register file, an instruction cache, an instruction fetch/dispatch unit, a constant cache, a shared memory, and a number of streaming processors and special-function units, all connected to high bandwidth GDDR5 DRAM interfaces. Every CUDA-enabled graphics card also fulfills a certain *compute capability* which describes the GPU's features and configuration [12, 32]. The versioning starts at 1.0, and each incremental version (minor increments are 1.1, 1.2, 1.3, and major increments are 2.x, 3.x, 4.x) is a super set of the previous.

Each SM contains 8 *streaming processors* (SP), also known as *CUDA cores*. The CUDA cores are the primary thread-processing units of the GPU: they perform fundamental operations such as integer and floating-point arithmetic, and each is a fully pipelined in-order core, optimized to balance delay and area. The total number of SMs, and hence also the number of SPs, is device-dependent. At the point of writing, the most sophisticated model – Tesla C2070 – sports 448 CUDA cores [14].

The SM also contains 2 *special-function units* (SFU) which compute fast approximations to certain functions such as $\sin x$, $\cos x$, $\log_2 x$ and $1/\sqrt{x}$. The SFUs are shared among the SPs and can be used to offload the SPs when accuracy can be sacrified for increased throughput.

The *register file* holds the thread contexts (i.e. program counter, local variables, stack pointers, etc). Its size is dependent on the GPU's compute

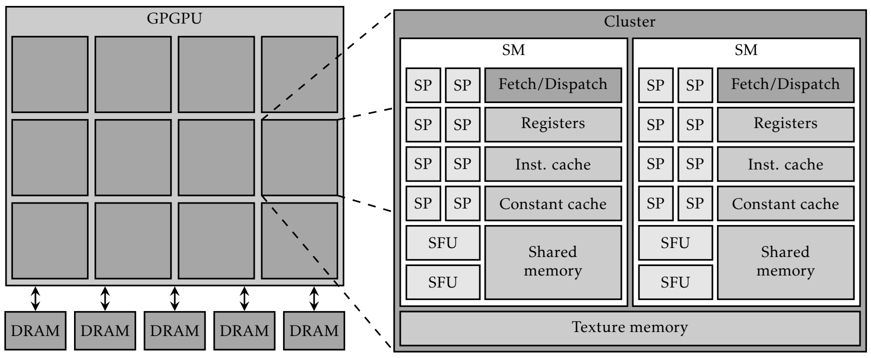


FIGURE 4.2 – Overview of NVIDIA's CUDA architecture. *Sources: adapted from [24, 33, 35].*

capability, varying between 8 KB, 16 KB and 32 KB (future version increments will probably provide even larger register files), capable of accommodating up to 768, 1024 and 1536 threads per SM, respectively. Each thread may also be allocated a small portion of the DRAM memory for registry spilling. This portion is called the *local memory*, most likely referring to its scope rather than its locality, and the DRAM memory as a whole is called the *global memory*.

The *shared memory* is a small (16 to 48 KB [12]), low-latency, multibanked on-chip memory which allows application-controlled caching of DRAM data. In many other architectures which contain similar memories this is commonly referred to as the *scratchpad memory*. Rewriting the application to make use of this memory often yields a tremendous performance gain (e.g. the throughput of a simple matrix multiplication kernel function increased from 21.6 to 345.6 gigaflops). This will be covered again but in greater detail in Section 4.4.

The CUDA GPU also contains a set of other caches for reducing memory latencies. The *constant cache* at 8 KB [12] is used for caching constant values residing in DRAM, thereby reducing memory read delays of such data. Pairs of SMs also share a *texture memory* between 6 and 8 KB [12] which cache neighbored elements in 2D matrices from the DRAM (compare this to normal CPU caches which are row major-oriented). This may improve performance if the memory accesses follow such a pattern.

Newer CUDA generations are also equipped with L1 and L2 caches (intermediate storage facilities which minimizes the number of read and write operations to global memory), but for simplicity these have not been drawn out. We will even assume that they are not present as they are transparent from the developer's point of view – the same way as we can safely ignore CPU caches in terms of functionality. Moreover, there has been little research covering the performance effects of L1 and L2 caches in GPGPUS, making it very difficult to take them into account when conducting code optimizations.

4.2.6 *Thread division and scheduling*

In order to handle the large population of threads and to achieve scalability, the threads are divided into a set of hierarchical groups – grids, thread blocks and warps [23, 32, 33, 35, 44] (see Figure 4.3).

A *grid* constitutes all threads belonging to the same kernel invocation (kernels will be covered in the next section). The grid is divided into thread blocks which are arranged in either a 1D or 2D geometry, with maximum 65,535 blocks in each dimension (compute capability 2.x even supports 3-dimensional grids). The geometry and size of the grid is configurable by the programmer in order to match the computing problem.

A *thread block* contains the threads to execute, arranged in either a 1D, 2D or 3D geometry. Similarly with grids, the thread block configuration can be tweaked by the programmer to match the structure of the input data; this simplifies memory data accesses. Each block is limited to 512, 512 and 64

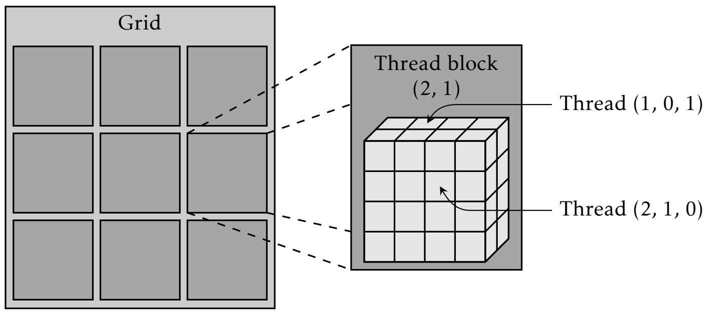


FIGURE 4.3 – Thread division and organization.

threads in each x, y and z dimension, respectively, but also the total number $x * y * z$ must not exceed 512 or 1024 threads, depending on whether the GPU supports compute capability 1.x or 2.x [12].

The thread blocks are distributed over the SMs for scheduling and execution. Each SM can handle up to 8 thread blocks and the GPU will dynamically balance the workload across the SMs. Thread blocks already allocated to an SM are not relocated, and a thread block does not relinquish its slot until all of its threads have finished their execution.

To efficiently manage this large population of threads with as little overhead as possible, threads are not scheduled individually. Instead, NVIDIA has developed a variant of SIMD technology known as *single-instruction, multiple-threads (SIMT) execution* [23, 32]. In SIMT, threads are bundled into *warps*. A warp is a set of threads that are adjacent to each other, i.e. the threads must be consecutive when ordered row major (see Figure 4.4). This obviously also requires that all threads within a warp must belong to the same thread block. A warp consists of maximum 32 threads, but may contain fewer threads (this occurs when the thread block size is not a multiple of 32).

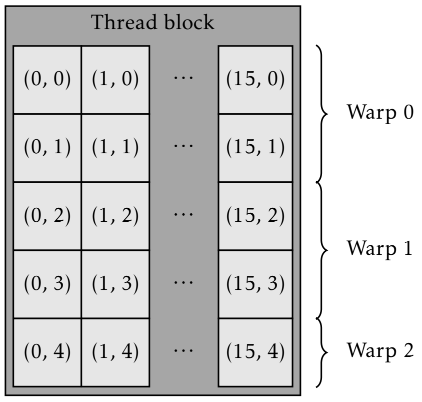


FIGURE 4.4 – Warp partitioning.

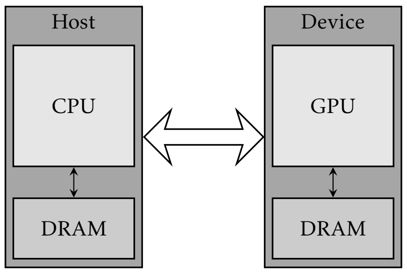


FIGURE 4.5 – Host and device separation in CUDA.

Warps eligible for execution are scheduled and executed as a whole. The next warp to be executed can belong to any thread block currently allocated to the SM, and there is no fairness between warps or thread blocks. The SM executes a warp by issuing the next instruction to all its SPS. Since there are only 8 SPS per SM, the thread execution is interleaved such that the instruction is first issued to threads 0 to 7 within the warp, then 8 to 15, etc.. Hence, if the next issued instruction takes 1 cycle to execute, then executing the entire warp will take 4 cycles. A consequence of this is that execution of warps consisting of less than 32 threads will leave some SPS to idle, thus failing to achieve full SP core utilization. One should therefore strive to achieve a thread blocks whose sizes are a multiple of 32 for optimal performance.

To allow threads within a warp to be executed in synchronous at all times, execution of data-dependent branch instructions must be serialized. What this means is that given an `if-else` statement whose expression has been evaluated, the SM will first issue all instructions within the `if` branch, and then all instructions within the `else` branch. Predicates are used to prevent threads that did take the currently executing branch from committing their result, and will thus effectively be stalled. Once all branches have been executed, the threads converge to the same execution path and thus synchronous execution of all threads is resumed. Avoiding thread divergence can therefore boost performance.

4.3 PROGRAMMING WITH CUDA C

In order to enable applications to be run on CUDA-enabled GPGPUS, NVIDIA provides an SDK consisting of a set of libraries, a CUDA C compiler called *nvcc*, and reference manuals. In principle, programming in CUDA C is quite straight-forward and often it takes only a few days effort for a skilled CUDA programmer to port appropriate C programs to CUDA C . Many applications – e.g. machine learning, database processing, bioinformatics, financial modeling, and medical images – have been accelerated with CUDA, at times yielding over a 100x speedup [24, 35].

In CUDA, the system consists of a *host* and a number of *devices*, the host being the CPU and the devices the GPGPUS (see Figure 4.5). All CUDA program generally apply the same host-device interaction pattern:

1. allocate memory on the device,
2. copy data from host to device,
3. perform data calculations on the device,
4. copy the result back, and lastly
5. free allocated memory.

This may even be iterated multiple times within the program, using the GPU to perform heavy data parallel computations and using the CPU for intense sequential algorithms which are hard to parallelize. In fact, this is exactly the key idea with GPGPUS.

To show how to write CUDA C programs, we will take a pure C program which implements matrix multiplication and then proceed by porting it to CUDA C.

4.3.1 *Starting with pure C*

For readers who have forgotten their linear algebra, matrix multiplication between a matrix A and matrix B is done by taking the dot product between a row from A and a column from B (also see Figure 4.6), i.e.

$$c_{i,j} = row_{A,i} \cdot col_{B,j}$$

The implementation in pure C is given in Listing 4.1). It iterates over the rows of matrix A and columns of matrix B, multiplying each pair-wise row-column value and calculates the sum which produces the value in the corresponding cell of matrix C. Note that the matrices are commonly stored *row major* in memory, meaning that the rows are preserved and stored one after another (see Figure 4.7).

4.3.2 *Porting to CUDA C*

The text in this section is primarily based on the material found in [32, 44].

Having a complete working C implemention of the matrix multiplication, we will now port it to CUDA C.

In normal circumstances, the device is not allowed access to the host memory. To circumvent this, we first allocate memory on the device for the matrices A, B and C:

```
cudaMalloc((void**) &Ma, size);
cudaMalloc((void**) &Mb, size);
cudaMalloc((void**) &Mc, size);
```

```
void matrixMult(int* a, int* b, int* c) {
    int i, j, k;
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            int sum = 0;
            for (k = 0; k < N; ++k) {
                sum += a[i * N + k] * b[j + k * N];
            }
            c[i * N + j] = sum;
        }
    }
}
```

LISTING 4.1 – Pure C version of matrix multiplication. *Source: [32].*

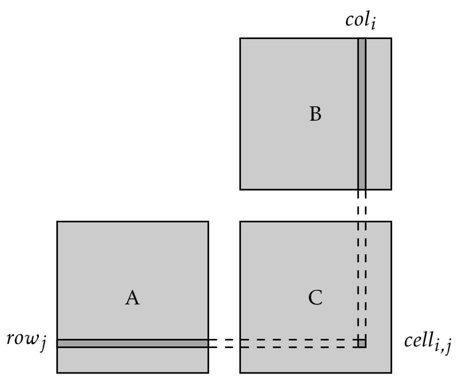


FIGURE 4.6 – Matrix multiplication. *Source: adapted from [32].*

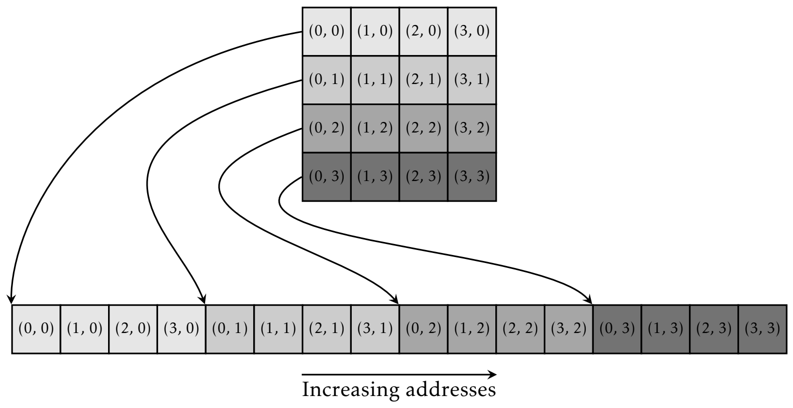


FIGURE 4.7 – Row major storing of matrices in memory. *Source: adapted from [32].*

The library function *cudaMalloc* is analogous with the C function *malloc*, but instead of allocating memory on the system it allocates memory on the device. Another difference is that *cudaMalloc* returns the pointer through a parameter instead of return values as *malloc* does. This is to be able to return status codes in case the operation should fail (for brevity we don't check the return values, but that should always be done in any production code; in fact, all CUDA library functions follow this convention).

The values of the matrices A and B are then copied to the global memory located on the device:

```
    cudaMemcpy(Ma, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Mb, b, size, cudaMemcpyHostToDevice);
```

This also entails that code executing on the device is only allowed to dereference pointers referring to the global memory on the device, and, similarly, code executing on the host may only dereference pointers to the host memory. Breaking this rule leads to undefined behavior.

Once copied, we can execute the calculations. This is done through a *kernel invocation*:

```
    dim3 gridDimension(1, 1);
    dim3 blockDimension(N, N);
    matrixMult<<<gridDimension, blockDimension>>>
        ((int*) Ma, (int*) Mb, (int*) Mc);
```

The syntax may appear a bit bizarre at first due to the introduced angular brackets. However, when ignoring these characters, a kernel invocation is nothing more than a function call – the angular brackets are there to be able to provide the grid and thread configuration of that invocation. In this example, we have chosen that the grid consists of a single 2-dimensional thread block of size $N \times N$.

But before we can invoke a kernel, it first needs to be declared:

```
__global__ void matrixMult(int* a, int* b, int* c) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Calculate dot product
    int k, sum = 0;
    for (k = 0; k < N; ++k) {
        sum += a[ty * N + k] * b[tx + k * N];
    }
    c[ty * N + tx] = sum;
}
```

Kernel functions are declared as a C function, with a few differences. First, its prototype must be preceded by the `__global__` keyword[1]. This indicates to the CUDA C compiler that this C function is a kernel function. Second, kernel functions are not allowed to make any recursive calls. This limitation is most likely caused by limited stack space, but has been lifted in later CUDA generations.

The most significant change in *matrixMult* is that its two outer loops have been removed. The kernel function will be executed by each thread within the entire grid, and although keeping the outer loops would produce the same result (ignoring race conditions), it would be a terrible waste of parallelizing potential. Instead of computing all values in matrix C, we rewrite the *matrixMult* to compute only a single value based on the values of `threadIdx.x` and `threadIdx.y`. These are predefined variables provided by the CUDA framework which specify a thread's x and y coordinates within the thread block. Having selected an appropriate thread block configuration, we can use these x and y coordinate values to directly select a corresponding row_x of matrix A and $column_y$ of matrix B to compute $value_{x,y}$ of matrix C, thus performing the calculations in parallel.

Lastly, when the device has finished executing the kernel, the results needs to copied back to the main memory and the allocated memory freed:

```
    cudaMemcpy(c, Mc, size, cudaMemcpyDeviceToHost);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
```

Since each thread block is limited to 512 or 1024 threads in total, depending on the GPU's compute capability, this program can only handle matrices up to sizes $\sqrt{512} \times \sqrt{512}$ or 32×32, respectively. This amount of data is far too small to reach full utilization of the GPU. We should therefore rewrite the kernel function to accommodate larger matrices by applying multiple thread blocks. The improved version of *matrixMult* is given in Listing 4.2 and illustrated in Figure 4.8. Assuming each thread block can contain at maximum 512 threads, this version is capable of handling matrices containing up to $(\sqrt{512} \cdot 65,535)^2$ elements (i.e. maximum grid size in each dimension with maximum thread block size). Should larger matrices need to be processed, or if the device runs out of global memory, the computations must be split into several kernel invocations.

[1] In addition, CUDA provides two more keywords – `__device__` and `__host__` – the former declaring a *device function* which may be invoked from the kernel, and the latter declaring a *host function* which may only be executed on the host. By default, all functions are declared as host functions if no keyword is given. Functions may also be declared using both `__device__` and `__host__`. This triggers the compiler to generate two versions of the function, allowing it to be run on both the host and the device.

```
__global__ void matrixMult(int* a, int* b, int* c) {
    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    // Calculate dot product
    int k, sum = 0;
    for (k = 0; k < N; ++k) {
        sum += a[row * N + k] * b[col + k * N];
    }
    c[row * N + col] = sum;
}
```

LISTING 4.2 – Improved CUDA version of the matrix multiplication program which allows larger matrices. *Source: [32].*

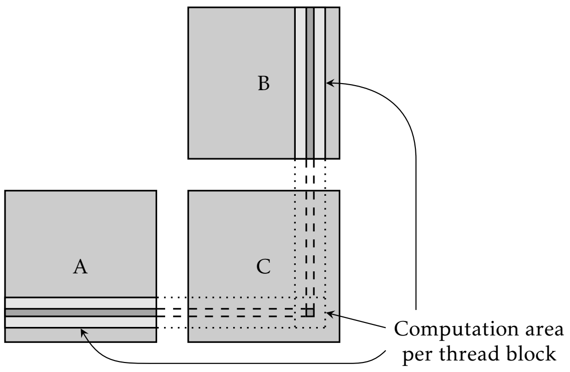


FIGURE 4.8 – Matrix multiplication using thread blocks.

4.4 CODE OPTIMIZATIONS

Simply porting an program to CUDA C rarely results in optimal performance; CUDA code is notoriously tricky to optimize as there are many factors affecting the performance, often with counter-intuitive results.

Ryoo et al. divides CUDA optimizations into six categories [37–39]:

1. memory bandwidth optimizations,
2. dynamic instruction reduction,
3. increasing thread-level parallelism,
4. increasing intra-thread parallelism,
5. work distribution, and
6. resource balancing.

Each will be covered individually in the following sections.

4.4.1 *Memory bandwidth optimizations*

The first category deals with optimizing memory bandwidth utilization, in particular of global memory. This is most often the key factor which limits speedup. To see why, let us examine the computation performance and memory bandwidth of NVIDIA Tesla C2070 [14]. It provides a theoretical single precision-floating point (SPFP) peak throughput of 1.03 teraflops and a memory bandwidth of 144 GB/sec. With one SPFP value being 4 bytes, this yields a computation-memory loading ratio of about 28 [28]. This means that for every loaded SPFP value, there need to be on average at least 28 operations per loaded value between each global memory load or else memory bandwidth will be saturated and threads will stall due to data starvation. Most algorithms simply don't exhibit such high computation-memory load ratios [28].

To reduce memory pressure, the programmer can

- cache global data in local, low-latencies memories, and
- rearrange memory access patterns to allow global read or write operations to be coalesced.

Using the shared memory, data from global memory can be *cached on-chip*, thereby allowing low-latency access if these data need to be used frequently. Revisiting the matrix multiplication example in Section 4.3, we notice that adjacent threads access the same row data in matrix A. This can be optimized, as shown in Listing 4.3.

Variables which are shared by multiple threads can be allocated in shared memory by prepending `__shared__` to the variable declaration. Using the `blockIdx` and `threadIdx` variables to drive the threads, the matrix multiplication algorithm now iterates over two steps: (i) load tile data from global memory into shared memory (see Figure 4.9), and (ii) calculate partial sum. Each step must be separated by a synchronization barrier to ensure that all threads have finished loading their data or performed their calculations before proceeding with the next tile. The barrier only encompasses all threads within the same block, but that is sufficient as the shared memory is also partitioned at thread block level. By caching the tiles in the shared memory, the number of global memory accesses is reduced by a factor of T if there are $T \times T$ tiles in total [32].

Since the shared memory is small – limited to 16 KB and 48 KB in compute capabilities 1.x and 2.x, respectively – it can easily become a limiting factor. In our example, we have selected the tiles to be 16×16 of 4-byte elements, which consumes 1 KB, i.e. each thread block demands 2 KB (one tile for each matrix) of shared memory. As each SM can manage 8 thread blocks at maximum, all tiles exactly fit and shared memory is thus not a limiting factor in our matrix multiplication application.

Memory operations issued by threads within the same warp performing memory reads or writes to adjacent addresses in global memory can be *coalesced* into a single transaction (see Figure 4.10 on page 35). In Listing 4.3 we

```
__global__ void matrixMult(int* a, int* b, int* c) {
    __shared__ int Da[TILE_SIZE][TILE_SIZE];
    __shared__ int Db[TILE_SIZE][TILE_SIZE];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;

    // Calculate dot product
    int i, sum = 0;
    for (i = 0; i < N / TILE_SIZE; ++i) {
        int k, sum_tmp = 0;

        // Load tile
        Da[ty][tx] = a[row * N + TILE_SIZE * i + tx];
        Db[ty][tx] = b[col + (TILE_SIZE * i + ty) * N];
        __synchthreads();

        // Calculate partial dot product
        for (k = 0; k < TILE_SIZE; ++k) {
            sum_tmp += Da[ty][k] * Db[k][tx];
        }
        sum += sum_tmp;
        __synchthreads();
    }
    c[row * N + col] = sum;
}
```

Listing 4.3 – Improved CUDA version of the matrix multiplication, now using shared memory to cache partial rows and columns of matrix A and B, respectively *Source: [32]*.

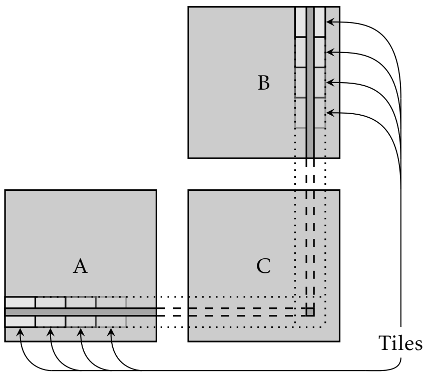


Figure 4.9 – Matrix multiplication using tiles.

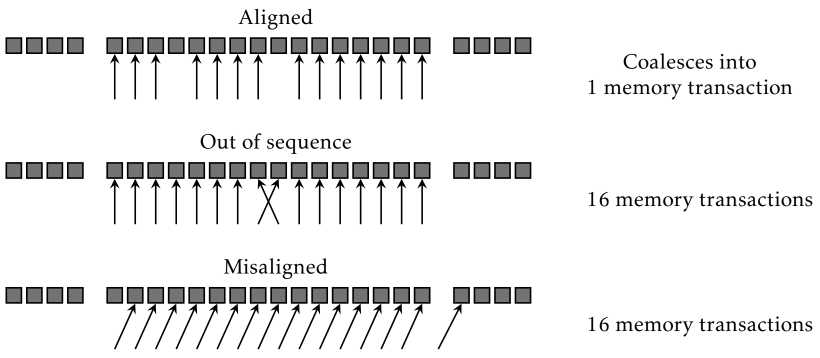


FIGURE 4.10 – Coalescing on reads performed on 4-byte memory units which are stored consecutively. Devices with compute capabilities 1.0 or 1.1 perform memory coalescing on 64- and 128-byte segments, while devices with compute capabilities 1.2 and up also perform coalescing on 32-byte segments. The examples above illustrate the capabilities of compute capabilities 1.0 and 1.1 which can only coalesce accesses which are aligned and within the 64-byte segment; devices with compute capabilities 1.2 and up are able to coalesce all these accesses into 1 memory transaction. *Source: adapted from [32].*

see that both load operations for Da and Db are already in such a pattern to allow coalescing, hence we need not to optimize the code any further.

4.4.2 *Dynamic instruction reduction*

The second category concerns increasing the instruction efficiency to increase performance. The idea is that if the same amount of work can be done using fewer instructions, then throughput must increase. This can be achieved by three independent methods:

- eliminating recurring subexpressions,
- moving loop-invariant code, and
- loop unrolling.

Recurring subexpressions, i.e. calculations which appear more than once within the code, can be removed by saving and reusing the result of the first calculation. However, this tends to require additional registers which may decrease the total number of threads which can reside in an SM.

Loop-invariant code is akin to multiplications found in mathematical summation formulas, i.e.

$$\sum_i ca_i \Leftrightarrow c \sum_i a_i$$

Operations unaffected by the loop itself can be moved from the body outside the loop, thereby avoiding redundant calculations and increasing instruction efficiency.

Loop unrolling is an optimization technique commonly applied automatically by compilers. It involves increasing the increment count used in the loop and duplicate the operations of the body. Listing 4.4 shows the simple CUDA version of matrix multiplication where the loop in the kernel function has been unrolled once.

While this actually increases the code size by adding extra instructions, it trades loop overhead for computation and hence increases the instruction efficiency. However, additional control logic may be required to handle cases where the new step count is not a multiple of the original number of iterations.

4.4.3 *Increasing thread-level parallelism*

The third category aims to increase core utilization by providing enough threads or warps to hide stalls due to long latency and blocking operations. For instance, a synchronization barrier will stall an entire warp until all warps of that thread block has reached the barrier. More independent warps can be created by decreasing the size of each thread block, thereby allowing the total number or thread blocks to increase. However, this should be done with care as this may increase memory pressure due to lessened data sharing.

Another optimization that I feel falls into this category, but which is not explicitly stated in [37–39], is minimizing thread divergence. Branch conditions, which cause some threads within a warp to execute one path and other threads a separate path, result in the execution of all threads to be serialized for each taken path. For example, let us assume the kernel contains an `if/else` statement and that half of the threads evaluate the expression to `true` and the other half evaluate the expression to `false`. Then as the threads reach this statement, the second half of the threads will be stalled as the first half executes its path. When the executing half has reached the end of its path, that half will now be stalled and the second half becomes

```
__global__ void matrixMult(int* a, int* b, int* c) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Calculate dot product
    int k, sum = 0;
    for (k = 0; k < N; k += 2) {
        sum += a[ty * N + k] * b[tx + k * N];
        sum += a[ty * N + (k + 1)] * b[tx + (k + 1) * N];
    }
    c[ty * N + tx] = sum;
}
```

LISTING 4.4 – Loop unrolled CUDA version of matrix multiplication.

ready and executes the `else` part of the statement. Once all paths have been executed, all threads converge to the same path and resume execution. Hence performance can be increased by either removing the branch entirely, or rearranging access patterns in such a way which causes entire warps to diverge instead of individual threads *within* a warp. Since warps execute independently from each other, thread stalling will be reduced. Let us look at an example.

Listing 4.5 shows the kernel function for vector sum reduction, which is also illustrated in Figure 4.11. While simple, it has several flaws: first, it waste thread resources as half of them will never even execute; and second, the algorithm causes a lot of intra-warp thread divergence. A much better approach is listed in Listing 4.6.

```
__global__ void sumReduce(int* v, int* sum) {
    __shared__ int partialSum[N];

    int tx = threadIdx.x;

    // Calculate sum reduction
    int stride, sum = 0;
    for (stride = 1; stride < N; stride <<= 1) {
        __syncthreads();
        if (tx % (2*stride) == 0) {
            partialSum[tx] += partialSum[tx + stride];
        }
    }
    *sum = partialSum[0];
}
```

LISTING 4.5 – Naïve approach to implementing *reduceSum* in CUDA. *Source: [32].*

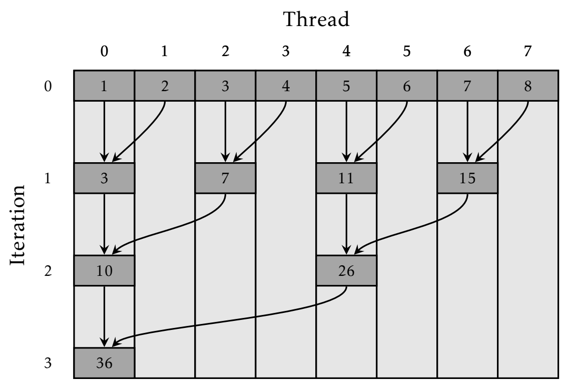


FIGURE 4.11 – Sum reduction – naïve approach. *Source: adapted from [32].*

```
__global__ void sumReduce(int* v, int* sum) {
    __shared__ int partialSum[N];

    int tx = threadIdx.x;

    // Calculate sum reduction
    int stride, sum = 0;
    for (stride = N >> 1; stride > 0; stride >>= 1) {
        __syncthreads();
        if (tx < stride) {
            partialSum[tx] += partialSum[tx + stride];
        }
    }
    *sum = partialSum[0];
}
```

Listing 4.6 – An improved CUDA implementation of *reduceSum*. *Source: [32].*

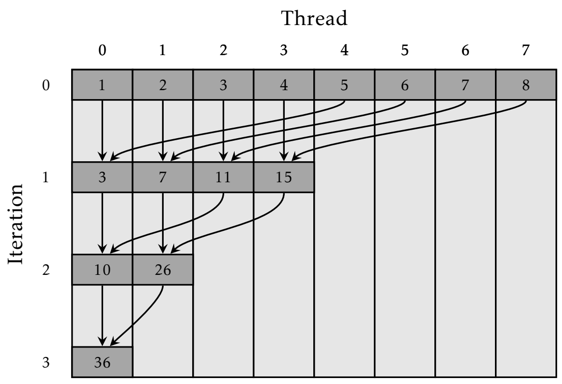


Figure 4.12 – Sum reduction – improved approach. *Source: adapted from [32].*

The new version exhibits a different pattern (see Figure 4.12) of summing elements by starting with a stride half the number of elements and then dividing it by 2 for each iteration. By doing this, `if (tx < stride)` will evaluate to either `true` or `false` for *all* threads within the same warp (at least as long as the stride is larger or equal to 32). This prevents thread divergence and hence minimizes stalling which consequently increases throughput.

4.4.4 *Increasing intra-thread parallelism*

The fourth category involves increasing the availability of independent instructions within a thread. This can be divided into two subcategories:

- instruction-level parallelism, and
- memory latency hiding.

Instruction-level parallelism (ILP) is primarily the domain of the instruction scheduler of the CUDA runtime system. According to Ryoo et al., it "appears to reschedule operations to hide intra-thread stalls, but sometimes does this to the detriment of inter-thread parallelism" [38] as rearranging instructions may increase register usage and thereby reduce the total thread count of each SM.

Memory latency hiding is a special case of ILP where computational instructions and memory operations are overlapped in order to hide these long latencies. This optimization may be performed by the compiler but can also be done explicitly by the programmer by applying a technique commonly known as *data prefetching*.

Returning to the CUDA version of *matrixMult*, we see that there is no overlap between the memory operations and the computations as these are divided by a synchronization barrier. This can be solved by prefetching the next data elements while consuming the current data elements, as shown in Listing 4.7.

4.4.5 *Work distribution*

The fifth category regards work redistribution across threads and thread blocks. Because to their nature, these optimizations are often unpredictable due to changes in register usage. For example, the tile size can be tweaked to optimize memory bandwidth usage; larger tiles maximizes the effects of caching by reducing the total number of global memory accesses. However, this also decreases scheduling flexibility since a larger percentage of threads must be stalled at synchronization barriers.

Other techniques are to increase the amount of work per thread (e.g. in matrix multiplication, each thread can calculate two values instead of one), and distribute work across multiple kernel invocations due to usage of limited sources, such as the constant cache.

4.4.6 *Resource balancing*

The sixth, and last, category concerns making changes in resource utilization to shift pressure from overused resources to underused resources. For example, registers can be proactively spilled by the programmer to global memory in order to allow more threads per SM, thus trading decreased instruction efficiency for higher thread-level parallelism. However, one must do this carefully as not to affect other optimization aspects negatively.

4.5 ALTERNATIVES TO CUDA

There are a number of alternative GPGPU-programming platforms, the most known being Khronos Group's Open Computing Language (OpenCL) – an

```
__global__ void matrixMult(int* a, int* b, int* c) {
    __shared__ int Da[TILE_SIZE][TILE_SIZE];
    __shared__ int Db[TILE_SIZE][TILE_SIZE];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    int row = by * TILE_SIZE + ty;
    int col = bx * TILE_SIZE + tx;

    // Load first tile
    int temp_a = a[row * N + tx];
    int temp_b = b[col + ty * N];

    // Calculate dot product
    int i, sum = 0;
    for (i = 1; i < N / TILE_SIZE; ++i) {
        int k, sum_tmp = 0;

        // Deposit tile into shared memory
        Da[ty][tx] = temp_a;
        Db[ty][tx] = temp_b;
        __synchthreads();

        // Load next tile
        Da[ty][tx] = a[row * N + TILE_SIZE * i + tx];
        Db[ty][tx] = b[col + (TILE_SIZE * i + ty) * N];

        // Calculate partial dot product from current tile
        for (k = 0; k < TILE_SIZE; ++k) {
            sum_tmp += Da[ty][k] * data_b[k][tx];
        }
        sum += sum_tmp;
        __synchthreads();
    }
    c[row * N + col] = sum;
}
```

LISTING 4.7 – Improved CUDA version of the matrix multiplication, using prefetching to load the tiles into shared memory. *Sources: [32, 38].*

open standard for doing general programming on heterogeneous platforms, adopted by Intel, AMD, ATI, NVIDIA and ARM – and Microsoft's DirectCompute.

OpenCL [13, 25] is very similar to CUDA – it uses the same notion of kernel functions and threads but uses a slightly different terminology and has a higher coding overhead (i.e. requires more code to achieve the same result) – and while CUDA only runs on NVIDIA CUDA-capable graphics cards, OpenCL is supported by a range of GPU vendors. An interesting difference is that the kernel functions are defined entirely as strings in OpenCL and compiled at runtime, whereas CUDA kernels are compiled beforehand along with the program. This allows OpenCL kernels to be optimized no matter which platform it will execute on. However, at the time, CUDA had more literature available and a larger user group (and thus more technical support), which is why it was decided that the software synthesis component should produce CUDA C code. Furthermore, porting a CUDA program into OpenCL should essentially only involve replacing the CUDA directives library calls with OpenCL equivalents and adding the code necessary for invoking the kernel.

DirectCompute [11, 49] is an extension of DirectX, forcing the programmer to write the applications as a special type of shader attached to the graphics pipeline. This may be convenient if the program needs to be integrated with Direct3D. However, unless the programmer is already familiar with the HLSL shading language, DirectCompute is more difficult and less accessible to use than CUDA. Moreover, DirectCompute has lower OS support as DirectX is only available on Windows platforms.

CHAPTER 5

GraphML

This chapter describes the GraphML standard, which is the format in which the ForSyDe model will be represented and given as input to the software synthesis component.

The text in this chapter is primarily based on the material found in [7, 8].

GraphML is a standardized format, developed by Ulrik Brandes et al., for specifying graphs and graph appearances. It was initiated by the Steering Committee of the Graph Drawing Symposium with the aim to overcome earlier failed attempts to devise a standard to improve tool interoperability.

5.1 BRIEF INTRODUCTION TO XML

GraphML builds upon another standardized format – XML. The XML standard allows data to be stored in a text format which can be read and understood by humans. The XML elements consists of *tags*, *attributes*, and *text* (the data to be stored). Text may only appear within a tag, which is written as

```
<tag>
  data_as_text
</tag>
```

Surrounding whitespace is ignored and does not affect the structure of the document; thus, the following would be equivalent to the code above:

```
<tag>data_as_text</tag>
```

A tag may also contain other tags, thus allowing data to be hierarchically ordered, e.g.

```
<tag>
  data

  <another_tag>
    some other data
  </another_tag>
  <tag>more data</tag>
</tag>
```

At the top of the hierarchy, there must always be a *root tag* which contains all other tags. A tag can also have one or more *attributes*, which are given within the tag as

```
<tag attribute="value">
  data
</tag>
```

For tags which contain no data, there is a short-hand notation which is written as

```
<tag />
```

We now have sufficient background knowledge to discuss the GraphML format.

5.2 THE GRAPHML FORMAT

The GraphML format specifies

- a set of XML tags and attributes which collectively define the structure of a GraphML document or file; and
- a mechanism which allows the GraphML format to be extended with application-specific graph data formats (will not be covered).

All GraphML files must start with some XML headers (these are not of interest for our purpose and will thus be ignored in this report) followed by the root tag `<graphml>`. The root tag contains the graphs specified in the file.

Graph is contained by the `<graph>` tag. A *graphs* are defined by a set of *nodes* and *edges*. An edge defines a connection between two nodes. A GraphML file may contain any number of graphs, although for this work we assume that there will be only one. A `<graph>` tag may contain any number of *core elements*, which consist of `<node>`, `<edge>`, and `<hyperedge>` tags. The order of these tags within the `<graph>` tag is insignificant to the structure of the graph. A graph can even contain subgraphs by nesting a `<graph>` tag within a core element.

A node is defined using the <node> tag. Each node has a unique identifier, given through the id attribute. A node may also have any number of ports which are defined by including <port> tags within the <node> tag. A port must have a name unique within the node, which is given through the name attribute.

An edge between two nodes is defined using the <edge> tag. The source and destination of the edge are given through the source and target attributes. If the edge is to connect to a certain port, then the ports of the source and destination are given through the sourceport and targetport attributes, respectively.

Simply defining nodes and edges is usually not enough to fully represent the intended graph. For instance, the nodes and edges may need to be attributed with additional data, like colors and weights. Such additional data can be added by including a <data> tag within whatever core element that needs to be augmented. A core element can contain any number of <data> tags. The data itself is then stored as data of the <data> (e.g., if we say that the data to be stored is green, then this is written as <data> green </data>). The <data> tag must also contain a id which specifies the *data key*. Data keys are defined using the <key> tag, which may only appear within the <graphml> tag alongside the <graph> tags. It is sufficient to think of the data key as describing what <data> tags may appear in the rest of the document.

An example of a GraphML file is shown in Listing 5.1 which defines the graph illustrated in Figure 5.1.

5.3 REPRESENTING FORSYDE MODELS IN GRAPHML

A ForSyDe model is really just a graph: the processes are nodes, and the signals are edges. Thus, we can represent a ForSyDe process in GraphML using a <node> tags and a set of <data> tags to capture the process type and any process constructur input parameter (e.g. function arguments and initial delay values). A signal is simply represented by an <edge> tag, where the source and sourceport attributes are set to the source process and output port of that process, and the target and targetport attributes are set to the destination process and input port of that process.

A complete example a ForSyDe model represented as a GraphML file is available in Listing 9.1 on page 86.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml [XML Schema references omitted] >
   <key id="desc" for="node" attr.name="node_desc"
       attr.type="string">
     <default></default>
   </key>
   <key id="weight" for="edge" attr.name="edge_weight"
       attr.type="float">
     <default>1.0</default>
   </key>
  <graph edgedefault="undirected">
    <node id="n0" />
    <node id="n1">
      <data key="desc">Power node.</data>
      <port name="south" />
    </node>
    <node id="n2">
      <graph edgedefault="undirected">
        <node id="n2_0">
          <data key="desc">This is an inner node.</data>
        </node>
        <node id="n2_1" />
        <edge source="n2_0" target="n2_1">
          <data key="weight">2.0</data>
        </edge>
      </graph>
    </node>

    <edge source="n0" target="n1">
      <data key="weight">0.5</data>
    </edge>
    <edge source="n1" target="n2" />
    <edge source="n2" target="n0" />
    <edge source="n2_1" target="n1" targetport="south">
      <data key="weight">0.5</data>
    </edge>
  </graph>
</graphml>
```

LISTING 5.1 – A GraphML example file.

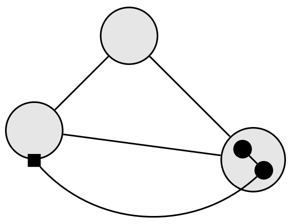


FIGURE 5.1 – Illustration of the graph specified in Listing 5.1.

CHAPTER 6

Challenges

This chapter lists the main challenges ahead of developing and implementing the software synthesis component.

From the material covered so far in this report, we can identify a set of main challenges in developing and implementing a prototype to the software synthesis component. The list below contains all identified challenges, the majority dealing with optimization problems. Each challenge has been categorized according to their nature, and prioritized as either Critical, High, Medium, or Low.

- Functionality challenges
  - *Identifying potential parallelization* (Critical)
    How to identify which parts of the model that can be parallelized and will benefit from it?
  - *Transforming ForSyDe processes into threads* (Critical)
    How to translate the functionality in the ForSyDe processes of the application model into CUDA kernel functions which can be executed on GPGPU?
  - *Avoiding recursive function calls* (Critical)
    In CUDA, recursion is currently prohibited within a kernel function. Will this cause problems when synthesizing ForSyDe models and how to avoid them?

- Memory optimization challenges
  - *Utilizing the shared memory* (High)
    How to include functionality in the synthesized code which enables utilization of the shared memory to minimize the number of global memory accesses?
  - *Maximizing memory coalescing* (Medium)
    How to optimize global memory access patterns in such a way that simultaneous memory accesses from multiple threads can be coalesced into a single memory issue?
  - *Utilizing the constant memory* (Medium)
    Data which is not modified can be cached in the constant memory, thereby reducing the number of global memory reads. How to identify situations when this memory can be used and how to apply it correctly?
  - *Optimizing shared memory accesses* (Low)
    Throughput to and from the shared memory is maximized if the threads of the same warp access separate modules or banks. How to optimize the synthesized code such that this pattern is achieved?
  - *Utilizing the texture memory* (Low)
    When data is stored in a 2-dimensional matrix, texture memory can be used to cache a cell's neighbors whenever it is accessed from global memory. Subsequent accesses to its neighbors will then be much faster as the data now resides on-chip. How to detect such memory access patterns and utilize the texture memory accordingly?
- Thread-level optimization challenges
  - *Implementing prefetching* (High)
    How to apply data prefetching in order to maximize core utilization through memory latency hiding?
  - *Minimizing thread divergence* (Medium)
    How to optimize the synthesized code such that threads of the same warp follow the same control flow as much as possible?
  - *Maximizing throughput by applying loop unrolling* (Medium)
    Unrolling loops can increase the computational-to-branch instruction ratio and thus increase throughput. How to detect when this will be beneficial, and how to apply it correctly?

  - ◇ *Removing redundant synchronization barriers* (Low)
    If threads operate on memory which will *only* be accessed by other threads within the same warp, then these accesses need no synchronization barriers. How to detect when this pattern occurs and thus eliminate the need of synchronization barriers?

  - ◇ *Trade lower accuracy for faster execution* (Low)
    By replacing operations for less accurate but faster equivalents (e.g. swapping division operators with `native_divide(x, y)`), higher instruction throughput can be achieved. How to detect when and where this is applicable?

- ■ Kernel-level optimization challenges
  - ◇ *Minimizing kernel invocations* (High)
    How to generate synthesized code which keeps the number of kernel invocations to a minimum?

  - ◇ *Utilizing more than one device* (Medium)
    How to optimize the code to use more than one GPU, if present? How to balance the work load evenly if the computational power of each GPU is not equal?

  - ◇ *Maximizing throughput by overlapping memory transfers and kernel invocations* (Low)
    On newer devices, one or more memory read and write operations to and from the GPU can be executed in parallel with a computation operation. How to optimize the synthesized code to exploit this feature?

  - ◇ *Eliminating redundant memory transfers between host and onboard device* (Low)
    When the GPU is integrated onto the motherboard, memory copying between the CPU and GPU is superfluous. How to detect this and act accordingly?

  - ◇ *Optimizing host-to-device data transfers using page-locked memory* (Low)
    Memory transfers between the CPU and the GPU can be made faster if the transactions are made over page-locked (or pinned) memory. How to detect when this is applicable, and how to apply it correctly?

As time is always a limiting factor, the main work of this thesis was dedicated to addressing all Critical-level challenges and as many of the High-level priority challenges as possible.

II

Development & Implementation

CHAPTER 7

Application Model Analysis

This chapter investigates how a ForSyDe model can be transformed into CUDA C code, concentrating on the image processing application model. It also covers some optimizations that can be performed on the model in order to improve the performance of the generated code. The idea is to only present the general theories in this chapter and postpone the intricate details to Chapter 8.

7.1 Application description

To guide the development of the software synthesis component, a particular ForSyDe model was used as proof-of-concept. This model was based on a reliability analysis program used in a printing system which imprints circuits onto paper. In order to verify the print, the system is equipped with a line scan camera which takes images of the ink drops at a very high frame rate. From these images, the application produces a Boolean matrix which indicates whether a drop was present or not in the corresponding cell. The output data can then be compared with an expected Boolean matrix, any discrepancies thus indicating a nozzle fault. From hereon, we refer to this program as the *line scan application*.

7.1.1 *ForSyDe model*

The ForSyDe model for the line scan application is illustrated in Figure 7.1. It was designed by my supervisor Christian Menne as part of his PhD project.

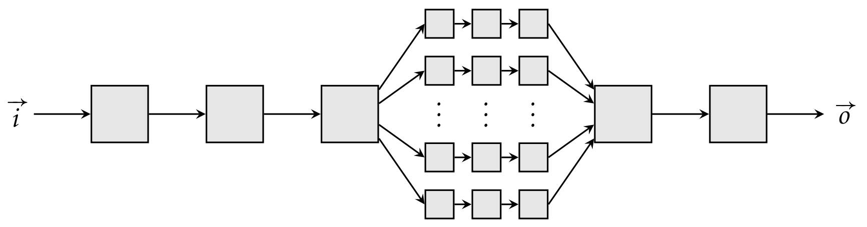


FIGURE 7.1 – ForSyDe model of the line scan application.

When image data is fed to the application, it is first run through a series of sequential processes which transform the data in order to simplify the parallelization process. The image data is then split and distributed to multiple parallel processes which analyze the data and determine whether an ink drop is present or not. Once all parallel processes have finished, the result data is aggregated and run through a last transformation process before being output. The precise functionality of the processes themselves will not be covered as not to disclose any industry secrets.

In order to avoid potential confusion, the line scan application model will be denoted as $\mathcal{LSM}$ when referred.

7.2 MODELS SUITABLE FOR SYNTHESIS TO CUDA C

As explained in Chapter 4, GPGPUS are throughput-oriented platforms designed for executing applications exhibiting massive data parallelism. Its threads execute the same kernel program but compute over separate data. A comparison of different model candidates is illustrated in Figure 7.2, from which we deduce that application model $\mathcal{LSM}$ should indeed be a good candidate for CUDA C synthesis. Inherently sequential or task parallel application models are assumed to lack enough data for execution on GPGPUS to be beneficial, and thus should be synthesized for other execution platforms such as CPUS.

Data parallel applications can be modeled in ForSyDe at two levels:

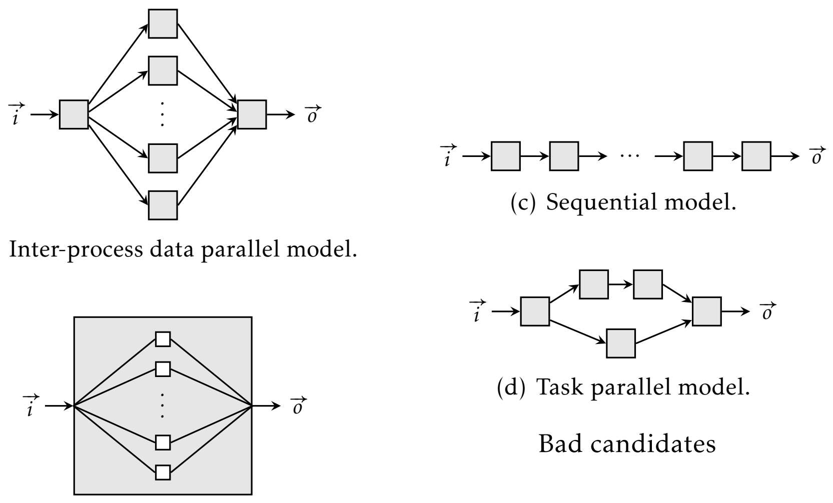
- inter-process (between several processes), and
- intra-process (within a single process).

We will look at each in turn in the subsequent sections.

7.2.1 *Inter-process data parallelism*

Using the process constructors *unzipxSY*, *mapSY* and *zipxSY*[1], inter-process data parallelism can be expressed in ForSyDe models by

[1] The names of these particular process constructors may be different in SystemC but the principle is the same as in Haskell.



(a) Inter-process data parallel model.

(b) Intra-process data parallel model.

Good candidates

(c) Sequential model.

(d) Task parallel model.

Bad candidates

FIGURE 7.2 – Comparison between good and poor candidates of ForSyDe models in terms of suitability for executing the synthesized code on GPGPUS.

1. splitting an array input signal into an array of signals through a *unzipxSY* process,
2. passing each signal through a *mapSY* process, and
3. merging back all signals into a single array signal via a *zipxSY* process.

Let us call this method of expressing data parallelism the *split-map-merge pattern*. Listing 7.1 shows an example where the method is applied, which creates the process network illustrated in (a) of Figure 7.2. This is the approach applied in application model $\mathcal{LSM}$. It should be noted that even though the *mapxSY* process constructor was used in the line scan model, the GraphML backend expanded each such instance into multiple *mapSY* processes (this will have repercussions that we will see later).

```
inc :: (Num a) => a -> a
inc x = x + 1

programSY :: Signal (Vector a) -> Signal (Vector a)
programSY v = zipxSY .
              mapV (mapSY inc) .
              unzipxSY v
```

LISTING 7.1 – Example of a ForSyDe model exhibiting inter-process data parallelism. The `mapV (mapSY inc)` construct could also be replaced by `mapxSY inc`; both are equivalent.

Models which makes extensive use of *mapSY* processes may contain a great amount of potential data parallelism to exploit. Moreover, using this approach, there is no data sharing between any two *mapSY* processes and thus requires no synchronization barriers which can potentially inhibit performance. Remember also that a *mapSY* process is equivalent to a *ZipWithNSY* process with one input signal.

7.2.2 *Intra-process data parallelism*

An extension of Haskell, called Data Parallel Haskell (DPH) [9, 29, 30], enables data parallelism by providing a set of arrays, or vectors, and functions which operate on those constructs. More than just signaling to the compiler that "these parts can be executed in parallel", DPH is also capable of handling nested data parallelism, i.e. where the data set consists of vectors of vectors of variable lengths.

A parallel vector in DPH is expressed as

```
[: type :]
```

The functions are often parallel siblings of those operating on regular lists and are thus named accordingly:

```
mapP     :: (a -> b) -> [:a:]-> [:b:]
zipWithP :: (a -> b -> c) -> [:a:] -> [:b:] -> [:c:]
sumP     :: Num a => [:a:] -> a
filterP  :: (a -> Bool) -> [:a:] -> [:a:]
etc.
```

Using these constructs, a intra-process equivalent of Listing 7.1 is given in Listing 7.2.

Intra-process data parallelism will not be considered in this work, although DPH is, in theory, supported in the shallow version of ForSyDe Haskell. However, it is not supported in the deep version and certainly not in ForSyDe SystemC. A further discussion will be done in the next section.

```
inc :: (Num a) => [:a:] -> [:a:]
inc = mapP (\x -> x + 1)

programSY :: Signal [:a:] -> Signal [:a:]
programSY v = mapSY inc v
```

LISTING 7.2 – Example of a ForSyDe model exhibiting intra-process data parallelism.

7.3 DISCOVERING AND EXPLOITING DATA PARALLELISM

The problem of discovering and exploiting potential data parallelism within a model can be approached in several ways:

1. Let the *software synthesis component* figure out where and when potential data parallelism can be exploited in the model and *always* execute those parts on a GPGPU.
2. Same as in 1 but *only* execute parts on a GPGPU when there is sufficient data for such execution to be beneficial (and leave this decision to the component).
3. Let the *model developer* figure out where and when potential data parallelism can be exploited in the model and *always* execute those parts on a GPGPU.
4. Same as in 3 but *only* execute parts on a GPGPU when there is sufficient data for such execution to be beneficial (and leave this decision to the component).

Much research has been conducted in finding efficient methods for automated parallelization, the latest involving transformations that translate the concerning code into a *polyhedral*, also called a *polytope*. Paul Feautrier describes one such process in [21]. The polyhedral model can be analyzed using mathematical methods to perform automatic partitioning of computation data and space allocation for frequently accessed elements onto fast on-chip memories. Muthu Manikandan Baskaran et al. propose such an algorithm in [4], along with a method for estimating when such allocations are beneficial.

We will apply approach 1 to discover inter-process data parallelism in ForSyDe models. This can be done by scanning the model for *unzipxSY* and *zipxSY* processes, checking that a *unzipxSY-zipxSY* pair forms a contained data flow, and that all processes in between are created using the *mapSY* process constructor. We will call such regions *data parallel sections*, and an algorithm for finding such sections is given in Section 8.1.

Deciding whether to execute the data parallel regions on a GPGPU or some other execution platform, however, is very difficult as it requires sophisticated static analysis of the combinatorial functions. Hence, approach 2 was abandoned in this thesis project.

Intra-process data parallelism is not considered in this thesis for several reasons:

1. it is not supported in either deep version of ForSyDe Haskell or ForSyDe SystemC;
2. such parallelism allows the possibility of *nested data parallelism*, which is much more difficult to exploit compared to flat structures [9]; and
3. intra-process data parallelism can often be rewritten to inter-process data parallelism.

Thus, to keep the work effort within reasonable boundaries, the scope of this thesis has been reduced to only analyze the split-map-merge pattern of expressing data parallelism and then always execute such regions on the GPGPU.

7.4 TRANSFORMING FORSYDE MODELS TO CUDA C

A reasonable approach on transforming a ForSyDe model to CUDA C code is to retain the notion of processes and trigger them according to a schedule which preserves the semantics of the model. For processes which have a function argument, the combinatorial functions can simple be invoked, passing the input signal values via the function parameters, and writing the result to the output signal. For processes which do not have function arguments, the values only need to be propagated from one signal to another.

7.4.1 *Mapping process function arguments to C functions*

Whatever the format the model is represented in, it must be possible to extract the process type and whatever parameters, such as function arguments, it may have. Once retrieved, they can be analyzed and translated into corresponding C functions which can be invoked by the scheduler. In this case, the representation is a GraphML file based on ForSyDe SystemC. As SystemC is implemented in C++, the function arguments can be copied directly.

There are, however, some discrepancies between C and C++ which hinders this method from being applicable in all general cases. For instance, C++ provides additional language constructs such as `bool`, `new` and `delete` which are not available in C . Hence, either these constructs are automatically recognized and converted into appropriate C equivalents by the component, or they are simply forbidden. In this work, we will use the latter and leave it to the programmer to ensure that the C++ functions used in the models are also syntactically and semantically correct in C.

To be understood by the software synthesis component, the function argument must also adhere to a few rules. A function must always accept its input data through the first parameter and return its produced value using either the `return` clause, or through the final parameter, but not both. The latter is necessary when the function produces an array as result since C functions can only return a single value[2]. Furthermore, when the input data is an array, the input parameter must be declared as `const` to prevent the function from accidentally modifying the input data and causing incorrect model behavior.

Data parallel processes can be mapped using the same principle but require more work as they will be executed on the GPGPU. This entails wrapping

[2]Another solution is to let the function allocate a new array and return its address as value, but that could potentially lead to memory leaks. Moreover, the C++-provided `new` clause is not available in C.

the functions into a CUDA kernel, configuring the grid and thread block dimensions as well as managing memory transfers between the host and the device. This will be discussed in greater detail in Section 8.6.

7.4.2 *Scheduling processes*

In the synchronous ForSyDe model we assume that processes are "infinitely fast" [42], an assumption which no longer applies once it has been synthesized into C . However, the generated code must still yield the result as if the perfect synchronous hypothesis was still in effect. This requires that the processes be *scheduled* and executed in an order that produces the correct output. There can be more than one correct schedule, except for purely sequential models where there is only one schedule which is easy to find (simply follow the chain of processes through the signals using depth-first search[3]). The problem is finding a schedule for models exhibiting task or data parallelism, various approaches were considered, and as one method lead to another, we will cover each in the order they were evaluated.

The first method was based intuitively on data flow analysis. It introduces the notion of execution paths and relies on those to find the schedule. Let us call this method SFM-EP (*Schedule Finding Method using Execution Paths*).

We begin with a few definitions.

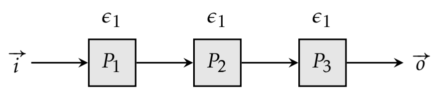
DEFINITION 7.1 – The set of input and output signals of a process P is denoted by $\textsc{in}(P)$ and $\textsc{out}(P)$, respectively.

DEFINITION 7.2 – An *execution path* is an ordered set denoted as ϵ_k, where $k \in \mathbb{N}_1$ and is unique for each separate execution path within a model, consisting of processes and other execution paths. For all execution paths it holds that an element $\alpha_i \in \epsilon_k$, where $1 \leq i \leq |\epsilon_k|$, can only be executed after all preceding elements α_j, where $1 \leq j < i$, have been executed.

An element α belongs to ϵ_k if:

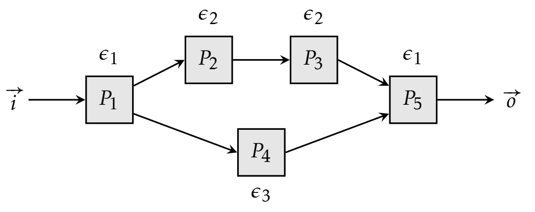
- α is a process P_1 and $\textsc{in}(P_1) = \textsc{out}(P_2)$, where P_2 is another process and $P_2 \in \epsilon_k$; or
- α is a process which receives all of its input signals from a set of processes $P_1, \ldots, P_n$, where $P_i \in \epsilon_m$, and $\epsilon_m \in \epsilon_k$; or
- α is an execution path caused by diverging output signals from a process P, where $P \in \epsilon_k$.

[3]There are two common ways of doing search in a graph: *breadth-first search* (BFS) and *depth-first search* (DFS) [10]. Breadth-first search means that all neighbors of the current node are visited first before proceeding. It can be thought of as traversing the graph in a circular fashion, and is usually implemented using a FIFO queue from which the next node to visit is popped from the head and its neighbors pushed to the back. Depth-first search works in the exact opposite and traverses down a path as far as possible, like shooting an arrow across the graph, and then backtracks to the next branch until there are no more nodes to visit. It is usually implemented using recursion.



$$\epsilon_1 = \{P_1, P_2, P_3\}$$

(a) Sequential model.



$$\epsilon_1 = \{P_1, \epsilon_2, \epsilon_3, P_5\}$$
$$\epsilon_2 = \{P_2, P_3\}$$
$$\epsilon_3 = \{P_4\}$$

(b) Task parallel model.

FIGURE 7.3 – Execution paths in ForSyDe models.

To illustrate the notion of execution paths, we turn to an example. Figure 7.3 shows the execution paths for a sequential and a task parallel model. In the sequential model we see clearly that the processes P_1, P_2 and P_3 all belong to the same execution path (ϵ_1). For the task parallel model, things become a bit more difficult. From P_1 the execution path ϵ_1 diverges into two separate execution paths ϵ_2 (containing processes P_2 and P_3) and ϵ_3 (containing only P_4). Since ϵ_2 and ϵ_3 both belong to ϵ_1, then P_5 must also belong to the same execution path as P_1.

The idea is that, if the execution paths can be found and ordered to form a single set E, then a schedule can be found by iterating over all elements in E and do the following for each element $\alpha \in E$:

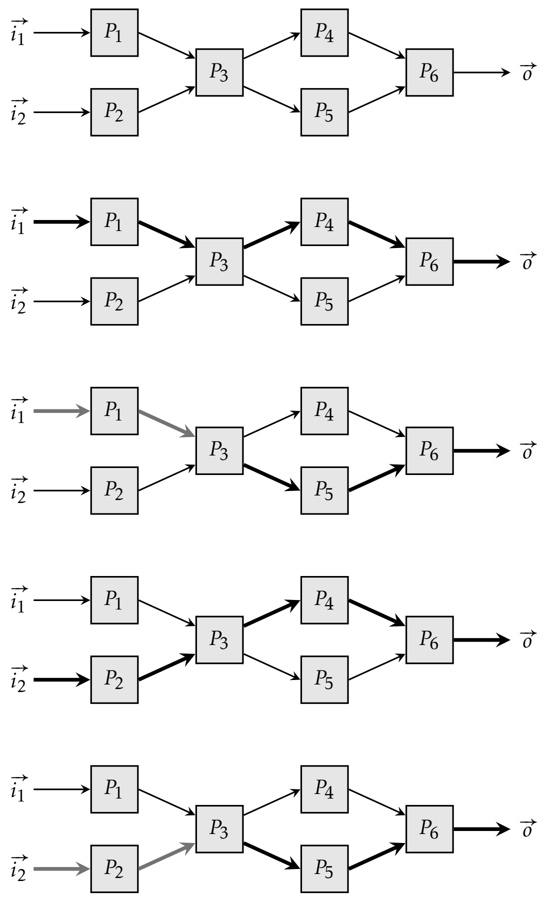
- If α is a process, add it to the schedule.
- If α is an execution path, iterate over all its elements and apply the same method.

The problem, however, is finding the execution paths. Furthermore, the execution paths need to be ordered together with the processes to form an ordered set where one element is not allowed to be executed unless all processes or execution paths prior to it in the set have been executed. Thus, building this set E is really nothing but finding a schedule and only pushes the same problem to another domain. Hence new methods were sought.

Realizing that attempting to find an execution order between processes from the model is just as difficult as finding a schedule, such ambitions were discarded entirely. However, since finding a schedule for a sequential model is so simple using DFS, could the strategy not be augmented to accommodate more complex models?

Indeed, it turns out that it can. By following the chain of processes from an input signal to an output signal, we first build a schedule containing all processes but with multiple invocations, and then prune away any duplicates

Step 1: Visit all processes



Schedule
–

Schedule
$\boldsymbol{P_1}$ $\boldsymbol{P_3}$ $\boldsymbol{P_4}$ $\boldsymbol{P_6}$

Schedule
P_1 P_3 P_4 P_6 $\boldsymbol{P_5}$ $\boldsymbol{P_6}$

Schedule
P_1 P_3 P_4 P_6 P_5 P_6 $\boldsymbol{P_2}$ $\boldsymbol{P_3}$ $\boldsymbol{P_4}$ $\boldsymbol{P_6}$

Schedule
P_1 P_3 P_4 P_6 P_5 P_6 P_2 P_3 P_4 P_6 $\boldsymbol{P_5}$
$\boldsymbol{P_6}$

Step 2: Prune away duplicates, starting from the end

Schedule
P_1 $\cancel{P_3}$ $\cancel{P_4}$ $\cancel{P_6}$ $\cancel{P_5}$ $\cancel{P_6}$ P_2 P_3 P_4 $\cancel{P_6}$ P_5 P_6 $\Rightarrow$ Schedule
P_1 P_2 P_3 P_4 P_5 P_6

Figure 7.4 – An example of how sfm-fps can be used to find a process schedule for ForSyDe models. Thick black arrows indicate paths taken during the current search from an input to an output signal. Grey arrows indicate paths which lead to the current search.

starting from the end of the schedule (a formal proof will not be given in this report). We call this method sfm-fps (*Schedule Finding Method using Forward Process Search*) and an example of how it works is illustrated in Figure 7.4.

This method is much more elegant than sfm-ep; the notion of execution paths is gone entirely and it relies on a node visiting method which is easy to implement. However, its simplicity has a price. Whenever a process has multiple output signals, the method must visits all following processes, potentially making numerous revisits (hence the need to prune the schedule afterwards). If the following part is shaped like a completely connected tree with n nodes, then the method will make n^2 visits. Thus, its time and space complexity[4] is $O(mn^2)$, where m is the number of input signals to the network and n is the number of processes. Furthermore, if the model contains *cycles*, also known as *feedback loops*, then an implementation of this algorithm will never terminate, which is completely unacceptable as such loops may appear in ForSyDe models. This method also does not work for models which have no inputs.

Fortunately, we can do better. First we avoid revisiting process nodes by maintaining a set of visited processes. This minimizes the number of visited processes, reduces the size of the intermediate schedule, and addresses the termination problem. However, this also raises a problem when a process requires more than one input. As all its inputs must have been produced before that process can be executed, we cannot add that process to the process until all preceding processes have been scheduled. This means that using the simple DFS can no longer be applied directly. One feasible solution is to backtrack along each input signal, but that seems unnecessarily complicated. A better approach is to *reverse* the search, i.e. to start at the output signals and visit the processes in a backwards fashion. Then, whenever a process requires multiple inputs, we can recursively call the method to produce partial schedules and then concatenate the schedules once all processes along those input signals have been visited. We call this method SFM-BPS (*Schedule Finding Method using Backward Process Search*) and this is the method used in the component for scheduling the processes. A simplified version is illustrated in Figure 7.5 using the same example model as in Figure 7.4, and a detailed description of SFM-BPS is given in Section 8.5.1.

7.4.3 *Scheduling data parallel processes*

Since data parallel processes are not executed individually on the GPGPU, certain changes to the model may be required in order to be manageable by the process scheduler. As covered in Section 7.2.1, inter-process data parallelism consists of three constructs: *unzipxSY*, *mapSY*, and *zipxSY* processes. On a GPGPU, these are actually executed in tandem, and hence need to be merged into a single process through a technique which we will call *data parallel*

[4]Algorithms are analyzed in terms of *complexity* [10], most often *worst case complexity* which is expressed using *big O* notation and written as $O(\alpha)$. Big O notation basically states that the worst case execution time of an algorithm for a given input is proportional to the expression α. For example, $O(n^2)$, where n is the size of the input, indicates that the worst case run time is proportional to the square of the input size. There is also *best case complexity*, expressed using *big Omega* notation and written as $\Omega(\alpha)$, and *average case complexity*, expressed using *big Theta* notation and written as $\Theta(\alpha)$. Throughout this report, it is assumed that we are always talking about worst case unless specified differently.

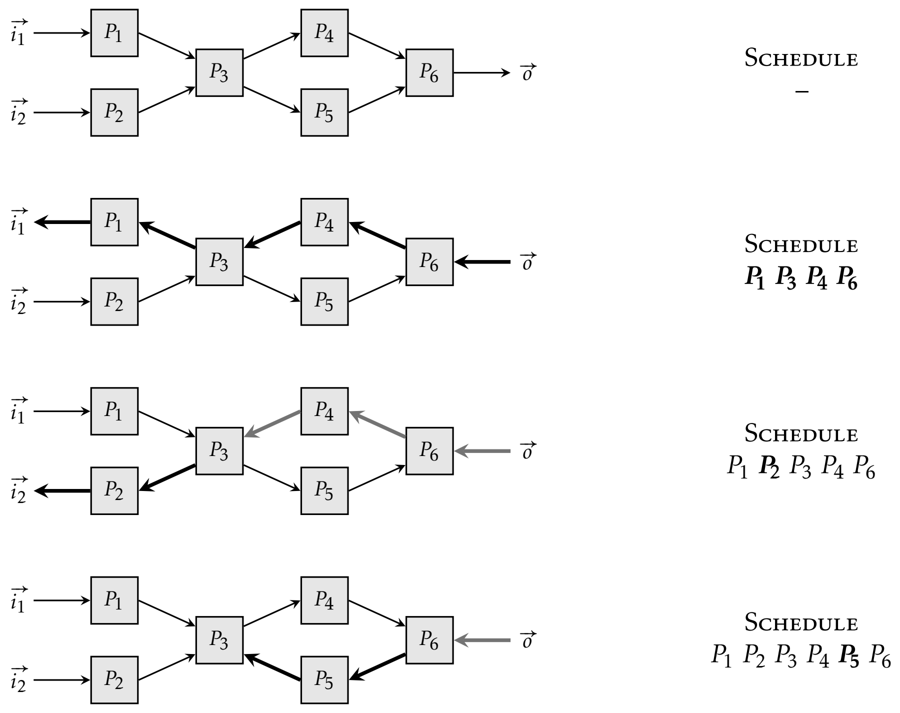


Figure 7.5 – A simplified example of how SFM-BPS can be used to find a process schedule for ForSyDe models. Thick black arrows indicate paths taken during the current search from an output to an input signal. Grey arrows indicate paths which lead to the current search.

section fusing (see Figure 7.6; note that is different from *process coalescing* which will be covered in Section 7.5.1). Let us denote this new process type *parallelMapSY*, which entails the functionality of all three process types that it replaces. The algorithm for this method is described in Section 8.3.

However, this cannot be done for data parallel sections which consists of more than one segment. A *segment* is a column of processes within a data parallel section (e.g., the model in Figure 7.1 on page 54 has three such segments). Thus, prior to fusing the data parallel section, we must either:

1. inject a *zipxSY* followed by a *unzipxSY* between each data parallel segment (this is known as *data parallel segment splitting* and is covered in Section 8.2); or
2. merge the segments into a single segment through process coalescing.

Once performed, the data parallel sections can be treated like any other process by the scheduler.

Thus, assuming we have a vector signal, a data parallel section can be expressed in two ways, either through:

1. a network of *unzipxSY*, *mapSY*, and *zipxSY* processes and interconnecting signals; or
2. a *parallelMapSY* process.

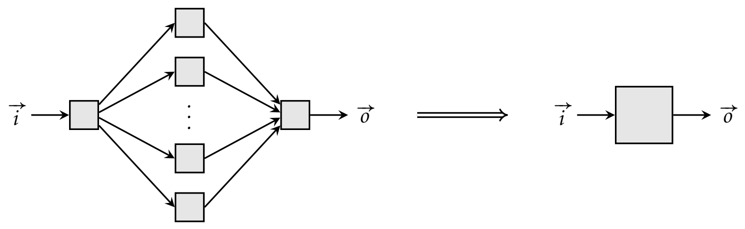


FIGURE 7.6 – Data parallel section fusing.

Although the former requires that a separate *mapSY* be declared for every element in the vector, it is easy to express in Haskell via the entityNamemapxSY process constructor which is based on Haskell's *mapV* function. However, that functionality disappears as the system model transitions from the Haskell domain to the GraphML domain, forcing all data parallel *mapSY* processes to be declared explicitly. This becomes impractical when the input vector grows large (remember that a GPGPU usually requires several thousands of threads to achieve full utilization). This problem could be solved if the *mapxSY* process was retained in the GraphML file, which would absolve the need to introduce a new process constructor.

7.4.4 *Mapping signals to appropriate data containers*

Since we are only considering the synchronous – the most restrictive – computational model of ForSyDe, processes are only allowed to consume and produce one token at a time. This enables signals to be represented using single-value data containers such as simple C variables. However, the exact nature of how these data containers are implemented is highly dependent on the functionality of the scheduler. We therefore defer the discussion about signals-to-intermediate storage until Chapter 8, where the scheduler is described in detail.

7.4.5 *Managing multiple port connections*

In ForSyDe, values produced by a process P can be used as input to multiple other processes. When generating the GraphML files for such models, there will be multiple connections to the out port of process P. Operating on such models is more difficult than models where every port is only allowed to be connected to a single other port.

To break such multiple port connections, we introduce a new process type called *copySY* which copies the value of its input signal to all of its output signals. Models exhibiting multiple port connections can then be modified by inserting a new *copySY* for each multiple connection point (see Figure 7.7).

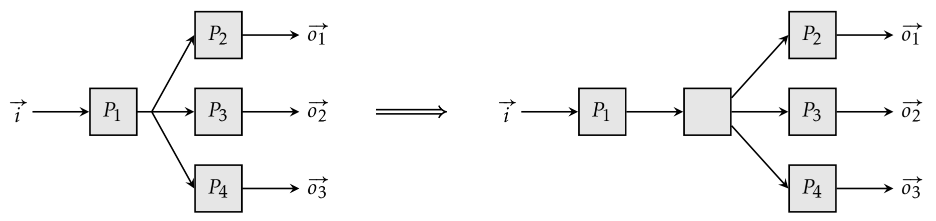


FIGURE 7.7 – Breaking multiple port connections by inserting an intermediate *copySY* process.

Although this simplifies model representation and processing, it also introduces additional signal propagation which will decrease performance if not dealt with in the later stages of the software synthesis process.

7.5 MODEL-LEVEL OPTIMIZATIONS

In a ForSyDe model, each process usually embodies a single functionality. This encourages modularity and component reuse. However, this is not optimal from a CUDA implementation and performance perspective as it may lead to excessive memory transfers and kernel invocations. Hence, it may be necessary to make semantic-preserving modifications to the model to improve the performance of the synthesized code.

7.5.1 *Process coalescing*

In Section 4.4.1, we discussed the limitations of global memory bandwidth and how it is essential to perform as many calculations as possible on each data loaded from global memory. As signals between processes imply transfer of data, we would like to minimize the number of signals in the model since that reduces the number of memory operations. In inter-process data parallel sections of the model, we can reduce the number of intermediate signals by merging processes together. This is called *process coalescing* and is illustrated in Figure 7.8. While the technique may appear to be simple, its implementation can be quite complex and all details are thus deferred until Section 8.4.

We begin with a definition, and then use that definition to formalize the optimization into a theorem.

DEFINITION 7.3 – The combinatorial function of a process P is denoted by FUNC(P), and $\boxplus$ denotes concatenation of one function to another (i.e. $f(x) \boxplus g(x)$ is equivalent to $g(f(x))$).

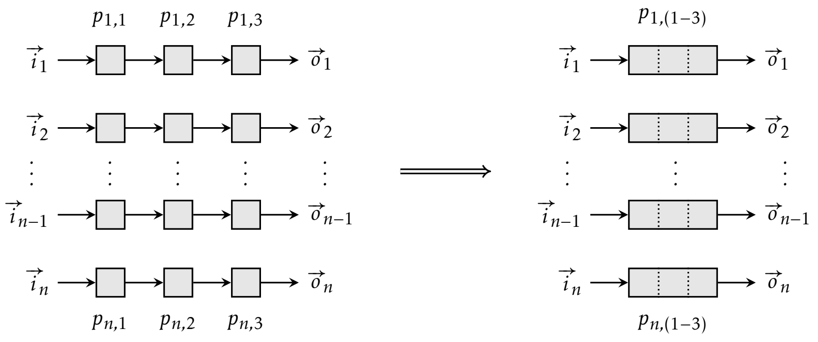


FIGURE 7.8 – Process coalescing.

THEOREM 7.1 – If OUT(P_1) = IN(P_2) holds for two processes P_1 and P_2, and neither is nor contains a delay process[5], then P_1 and P_2 can be replaced by a new process $P_{1,2}$ where IN($P_{1,2}$) = IN(P_1), OUT($P_{1,2}$) = OUT(P_1) and FUNC($P_{1,2}$) = FUNC(P_1) $\boxplus$ FUNC(P_2). After coalescing, the signals between P_1 and P_2 can be removed.

Proof. The condition OUT(P_1) = IN(P_2) simply states that all input signals of process P_2 must be exactly the same as the output signals of P_1. If this holds, then we know that the output of P_1 is not directed to any other process but P_2, and P_2 receives no input from any other process but P_1. This requires, however, that the output from P_2 is directly based on the input to P_1, which is assured if there are no delay elements in between P_1 and P_2. Thus it is safe to coalesce these two without breaking the semantics of the model. □

By combining the functionality of several subsequent processes into a single process, the number of kernel invocations needed to perform the calculations is minimized. This also increases the computation-memory load ratio as more computations are executed for each data read from and written to the global memory. Hence, process coalescing seems to be a promising method for increasing performance of the CUDA C code synthesized from data parallel ForSyDe models.

[5]Such processes can be created using the process constructor *delaySY$_k$*, which delays a signal by k events.

CHAPTER 8

Methods and Algorithms

This chapter covers all methods and algorithms applied in the implementation of the software synthesis component. The functionality of each algorithm is described in detail and its time and space complexity is also analyzed when applicable. Many of the techniques have already been briefly covered in Chapter 7 to illustrate the general approach while deferring the technicalities to this chapter.

8.1 finding data parallel sections

The first step to exploiting inter-process data parallelism within a model is to find where the data parallel sections are. One method for doing this is given in Listing 8.1. Although it explicitly relies on *mapSY* processes, the method can also be applied on models which use one-input *ZipWithNSY* processes by first converting these into *mapSY* processes.

First, the algorithm computes a list of contained sections in the model. A section is denoted by a start and end process and is *contained* if all signals emerging from the start process converges at the end process, and vice versa. The contained sections are found by first doing a dfs over all processes in the model, starting from the model outputs. To fix the termination problem when the model contains cycles, a set of visited processes is maintained. When an unvisited process of type *zipxSY* is found, the method attempts to find the nearest *unzipxSY* process along the reversed data flow direction. If such a process is found, the next step is to check if the data flow between these two points is contained (i.e. denotes a contained section). A method for checking data flow containment between two processes is given in Listing 8.2 on page 69.

```
function FindDPSections(M)
  returns data parallel sections within M
  inputs: M, a ForSyDe model

  sections ← FindContainedSections(M)
  for each section in sections do
    if not isSectionDP(section) then remove section from sections

function FindContainedSections(M)
  returns list of contained sections within M
  inputs: M, a ForSyDe model

  sections ← empty list
  visited ← empty set
  for each output port port in M do
    schedule ← schedule + FindContainedSections(process of port, visited)
  return schedule

function FindContainedSections(P, visited)
  returns list of contained sections found when starting the search from process P
  inputs: P, a process starting point
          visited, set of visited processes

  sections ← empty list
  if $P \notin visited$ then
    visited ← $visited \cup P$
    if P is of type zipxSY then
      start_point ← FindNearestUnzipxSY(P)
      if start_point was found then
        if the data flow is contained between start_point and P then
          sections ← sections + {start_point, P} +
            FindContainedSections(endpoint, visited)
        else goto continue_search
      else return sections
    continue_search:
    for each connected out port port in P do
      sections ← sections + FindContainedSections(process of port, visited)
  return sections

function isSectionDP(S)
  returns true if the section S is data parallel
  inputs: S, a contained section

  chains ← all chains of processes between start and end point of section
  if not all chains in chains are of equal length then return false
  if not chains consists of only mapSY processes then return false
  for each segment in section do
    if not all processes in segment have identical function arguments) then
      return false
  return true
```

LISTING 8.1 – Algorithm for finding data parallel sections. The rest of the implementation continues in Listing 8.2.

```
function FindNearestUnzipxSY(begin)
  returns nearest unzipxSY process from a given process point
  inputs: begin, a process search starting point

  if begin is a unzipxSY then
    return begin
  else
    for each in port port in begin do
      if port is connected then
      unzipxsy ← FindNearestUnzipxSY(process of other end of port)
      if unzipxsy was found then
        return unzipxsy
  return indication that no such process was found

function CheckDataFlow(start, end, direction)
  returns true if all data flows begin or end at the start or end points
  inputs: start, a process starting point
          end, a process end point
          direction, data flow direction to check

  if start = end then return true
  if direction = forward then
    for each out port port of start do
      if not CheckDataFlow(process of port, end, forward) then
        return false
  else
    for each in port port in end do
      if not CheckDataFlow(start, process of port, backward) then
        return false
  return true
```

Listing 8.2 – Algorithms for finding the nearest *unzipxSY* process and for checking data flow containment.

Once all contained sections have been found, the algorithm runs some checks on the process chains within each section (a *process chain* is the chain of processes between the *unzipxSY* and *zipxSY* processes). If all process chains are of equal length, consists of only processes of type *mapSY*, and each segment applies the same combinatorial function, then the section is a data parallel section. If not, then the section is removed from the list.

In most circumstances the algorithm will only visit each process in a model once in its search for data parallel sections. This takes $O(n)$ time, where n is the number of processes within the model. Checking data flow containment requires visiting each process within the section twice – once in the forward check and once in the backward check – thus also taking $O(n)$ time (except that n is now the number of processes within the section and not the entire model). The same applies for for checking if a contained section is also a data parallel section. The last significant factor to consider is the time complexity for checking whether a process exists in the visited set. Fortunately, such sets can be implemented using *hash tables*. A hash table is a vector or array of lists where the index is calculated using a *hash function*. Using the element as input data, the hash function computes a seemingly random yet deterministic value, which is not necessarily unique for every input data. This value is used as an index in the table, and the corresponding list is then traversed. Although it is theoretically possible to reach time complexity $O(n)$ in worst case, using a large enough table and good hash functions will effectively bring this down to $O(1)$ [10]. Hence, the time complexity for the entire algorithm appears to be linear (i.e. $O(n)$). However, if the model consists of a complete tree of *unzipxSY* and *zipxSY* processes, then the entire remaining part of the network may be visited for every process. This would bring the time complexity to $O(n^2)$, but such networks are hopefully very unusual.

8.2 SPLITTING DATA PARALLEL SEGMENTS

In Section 7.4.3, we talked about why the segments in a data parallel section may need to be separated by injecting a *zipxSY* and *unzipxSY* process. Although the procedure for doing this is simple, the algorithm is given in Listing 8.3 for completeness.

First, it finds the data parallel sections within the model. For each section which has process chains longer than 1 unit, the method then:

1. traverses over each intersection between two process segments,
2. creates a new *zipxSY* and *unzipxSY* process,
3. connects the output of the *zipxSY* process to the input of the *unzipxSY*,

```
function SplitDPSegments($M$)
  returns modified model
  inputs: $M$, a ForSyDe model

  $sections \leftarrow$ FindDPSections($M$)
  for each $section$ where its chains are longer than 1 in $sections$ do
    $chains \leftarrow$ convert $section$ into vector of vector of processes
    for $segment \leftarrow 1$ to size of $chains[0] - 1$ do
      $zip \leftarrow$ new $zipxSY$ process
      $unzip \leftarrow$ new $unzipxSY$ process
      connect output of $zip$ to input of $zip$
      for $i \leftarrow 0$ to size of $chains$ - 1 do
        connect output of $chains[i][segment - 1]$ to input at index $i$ of $zip$
        connect input of $chains[i][segment]$ to output at index $i$ of $unzip$
```

Listing 8.3 – Algorithm for splitting data parallel segments.

4. connects the outputs from the previous segment to the inputs to the *zipxSY* process, and
5. connects the inputs of the next segment to the outputs of the *zipxSY* process.

Its time complexity is primarily dominated by FINDDPSECTIONS, i.e. $O(n^2)$.

8.3 FUSING UNZIPXSY, MAPSY, AND ZIPXSY PROCESSES

Prior to scheduling, the data parallel sections need to be fused into a single process through a process called *data parallel section fusing*. This insulates the scheduler from having to concern itself with whether the data parallel section is executed on the CPU or GPGPU. Although trivial, the algorithm is given in Listing 8.4 and described for completeness.

This method also starts with finding the data parallel sections within the model. It then creates a new *parallelMapSY* process to replace the the *zipxSY*, *mapSY*, and *unzipxSY* processes in the section. The function argument of the *mapSY* processes (any one will do as they are the same at this point) is copied and the inputs and outputs of the section redirected to the new process. Finally, the obsolete processes and signals are removed from the model. This method also takes $O(n^2)$ time to execute for the same reason as with the SPLITDPSEGMENTS algorithm.

8.4 COALESCING MAPSY PROCESSES

In Section 7.5.1, we discussed how certain processes could be coalesced, or merged, into a single equivalent. This minimizes global memory transfers in the synthesized CUDA code. While the principle appears to be simple, initially its implementation proved to be the opposite.

```
function FUSEDPSECTIONS(M)
  returns modified model
  inputs: M, a ForSyDe model

  sections ← FINDDPSECTIONS(M)
  for each section in sections do
    new_process ← new specialized process
    copy function argument from mapSY processes in section to new_process
    redirect inputs to section to inputs to new_process
    redirect outputs from section to outputs from new_process
    remove entire section from M
```

LISTING 8.4 – Algorithm for fusing data parallel sections.

```
function CoalesceMapSYProcesses($M$)
  returns modified model
  inputs: $M$, a ForSyDe model

  sections ← FindDPSections($M$)
  chains ← all chains in sections
  for each chain in chains do
    new_process ← new specialized process
    new_function ← new empty function
    input_param ← function input parameter with the same data type as the input
      parameter to the function of first process in chain
    add input_param to new_function
    set return value of new_function to the same data type as the return value of the
      function of last process in chain
    source_variable ← input_param
    dest_variable ← empty
    for each process in chain do
      function ← function of process
      dest_variable ← new unique variable with same data type as return value of
        function
      add code to new_function which adds dest_variable as local variable
      add code to new_function which set the value of dest_variable by invoking function
        with source_variable as input
      source_variable ← dest_variable
    add code to new_function which returns value of dest_variable
    redirect input to first process in chain to input to new_process
    redirect output from last process in chain to output from new_process
    remove entire chain from $M$
```

Listing 8.5 – Algorithm for coalescing *mapSY* processes.

Coalescing *mapSY* processes is equivalent to merging their process function arguments into a single function. The first considered approach entailed inlining one function onto another. However, this typically requires that the functions are first converted into *abstract syntax trees* (AST) [2, 3], a process demanding syntax analysis of a language which is infamous for its difficulty to parse. Then the tree of one function must be appended to another by replacing occurrences of `return` statements in the tree. Furthermore, the local variables within a function must be renamed in order to avoid name clashes.

Fortunately, a much simpler approach was discovered. Instead of modifying the existing functions, a new function is created which calls each function in a sequential manner. This is relatively straight-forward and can be done without having to analyze the functions[1]. The algorithm is given in Listing 8.5, and its principle can also be applied for coalescing *parallelMapSY* processes.

The algorithm first looks for chains of *mapSY* processes within the data

[1]The prototypes must still be examined in order to discover the data types of its return value and parameters, but that can be done easily through a series of string searches.

parallel sections of the model. Then, or each chain, it constructs a new process which contains a new function as well as the functions of the other *mapSY* processes. To allow this, a new process type *coalescedMapSY* was devised. The new function has the same input parameters as the function in the first process in the chain, and returns a value of the same data type as the value returned from the function in the last process. The algorithm proceeds with iterating over all function arguments in the chain and generates code which sequentially invokes each function of the following *mapSY* processes. The intermediate results are stored in a new local variable (optimization of the register use is left to the C compiler). Lastly, the input and output signals to the chain are redirected to the new process and the entire chain is removed from the model. Iterating over all chains take $O(n)$ time, but finding the chains may take $O(n^2)$ at most, thereby yielding a total time complexity of $O(n^2)$.

8.5 THE PROCESS SCHEDULER

The process scheduler is responsible for scheduling the processes such that the perfect synchrony hypothesis is obeyed and managing the signals which transfer the data from one process to another.

We begin by covering the algorithm used for finding a process schedule.

8.5.1 *SFM-BPS*

As described in Section 7.4.2, several approaches were considered in finding schedule for a ForSyDe model. The first method – SFM-EP – proved incomplete, and the second method – SFM-FPS – exhibited a higher time and size complexity than necessary. This, among other limitations, was fixed in the third method – SFM-BPS[2]. A brief sketch of SFM-BPS illustrating its approach was also given in Section 7.4.2 while postponing its details which will be covered in this section.

The algorithm for SFM-BPS is given in Listing 8.6, and a simplified execution on an example model is illustrated in Figure 7.4 on page 61. First it builds a queue of *starting points* containing all processes that are directly attached to the model outputs. For each process in the queue, it builds a partial schedule which is appended to the complete schedule. This process is repeated until the queue is empty.

The partial schedule is generated by FINDPARTIALSCHEDULE. For a given input process P, the function recursively calls itself on each process P' which is located at the other end of every input port of P. This allows the algorithm to visit every process that is involved in generating the model outputs and thus must be part of the schedule. The recursion stops when either

[2] As a reminder, SFM stands for Schedule Finding Method, EP for Execution Paths, FPS for Forward Process Search, and BPS for Backward Process Search.

```
function FindSchedule(M)
  returns process schedule for M
  inputs: M, a ForSyDe model

  schedule ← empty list
  $visited_G$ ← empty set
  $visited_L$ ← empty set
  queue ← empty queue
  for each output port port in M do
    add process of port to head of queue
  while queue is not empty do
    process ← head of queue
    remove head from queue
    $visited_L$ ← empty set
    partial_schedule ← findPartialSchedule(process, $visited_G$, $visited_L$, queue)
    if insertion point of partial_schedule is at beginning then
      insert partial_schedule before head of schedule
    else
      insert partial_schedule after insertion point in schedule
    $visited_G$ ← $visited_G \cup visited_L$
    return schedule

function FindPartialSchedule(P, $visited_G$, $visited_L$, queue)
  returns schedule for processes visited along the path from P to a model input port
  inputs: P, a process starting point
          $visited_G$, set of globally visited processes
          $visited_L$, set of locally visited processes
          queue, queue of starting points

  schedule ← empty list
  if $P \notin visited_G$ then
    return schedule with P as insertion point
  if P is a delay element then
    add process at other end of in port of P to end of queue
    return P with at beginning as insertion point
  ip ← at beginning
  if $P \notin visited_L$ then
    $visited_L$ ← $visited_L \cup \{P\}$
    for each in port port in P do
      partial_schedule ← FindPartialSchedule(process at other end of port, $visited_G$,
        $visited_L$, queue)
      if insertion point of partial_schedule is not at beginning then
        ip ← insertion point of partial_schedule
    schedule ← schedule + P
  return schedule with ip as insertion point
```

Listing 8.6 – The SFM-BPS algorithm.

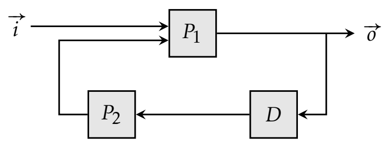
1. an input signal is reached,
2. an already visited process is reached, or
3. a delay element is reached.

The first two conditions are simple to understand: an input signal has no process at its output port, and halting at an already visited process avoids redundant search – a problem which was evident in SFM-FPS. More importantly, it also provides a termination condition for when the model contains feedback loops. However, this alone is not enough to generate a correct schedule for such models.

To see why the first two halting conditions are not sufficient, we turn to an example. In Figure 8.1, we see two slightly different models containing feedback loops, along with the correct schedule and the schedule generated by SFM-BPS if only the first two conditions were applied. In reality, the exact placements of the delay elements in the schedule is not important as long as all the other processes are correctly scheduled (to see why, see Section 9.4). Hence, in (a), the generated schedule is acceptable, while the schedule generated in (b) is clearly not correct.

To fix this problem, we need the third halting condition. However, instead of returning an empty list, the algorithm returns a partial schedule containing only the delay element D that was reached. Furthermore, the process P', which precedes process D in the model, is added to the end of the starting point queue. We also augment the structure returned by FINDPARTIALSCHEDULE to include an *insertion point*. The insertion point can assume one of the following two values:

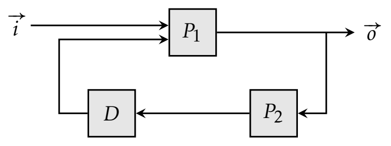
- If the recursion stopped due to reaching an already visited process P, then the insertion point for the partial schedule is *after process P* in the complete schedule.
- Otherwise the insertion point is *at the beginning*.



GENERATED SCHEDULE
$D\ P_2\ P_1$

CORRECT SCHEDULE
$P_2\ P_1\ D$

(a) Acceptable



GENERATED SCHEDULE
$P_2\ D\ P_1$

CORRECT SCHEDULE
$D\ P_1\ P_2$

(b) Incorrect

FIGURE 8.1 – Schedules generated for two example models containing feedback loops if the algorithm only applies the two first halting conditions. The D process is a delay element.

With this the algorithm can properly handle feedback loops as described in Figure 8.1. However, if the feedback loop itself contains feedback loops, then the result will not be as expected. We fix this by dividing the set of already visited processes into two sets – one set of *globally visited* processes and one set of *locally visited* processes. The idea is that the inner feedback loop is considered *local* in the scope of the entire model, and should thus be contained in the partial schedule. Upon return from FindPartialSchedule, the processes in the locally visited set are added to the globally visited set and then emptied before the initiating the next partial schedule search. Lastly, we slightly change the decision of the insertion point by requiring the visited process P to have been *globally visited* in order for the insertion point to become P. In all other cases, the insertion point is *at the beginning*.

Having devised a method which *appears* to be more efficient than its predecessor, let us now analyze whether it actually *is*. By maintaining a set of already visited processes, the method effectively only visits each process once, thus bringing its time and size complexity to $O(n)$ where n is the total number of processes in the model[3]. Using hash tables allows the visited sets to be searched and added to in constant time. Another significant factor to consider is search of the schedule for an insertion point, which in worst case may take $O(n)$ if the entire schedule needs to be traversed. In fact, if this is done for every partial schedule, the total number of traversals may become $O(n^2)$. Fortunately, since we are only looking for a particular element in the schedule, we can maintain a hash table to allow the sought element to be found in constant time. Regarding space, no process appears in any schedule – partial or complete – or any visited set more than once. Hence, the total time and space complexity for SFM-BPS appears to be $O(n)$ – a great improvement from SFM-FPS' $O(mn^2)$.

8.5.2 *Managing the signals*

Signal management entails two tasks: (i) transferring data from one process to another, and (ii) storing intermediate results, if necessary, until all processes have been executed. Output data from a process may not necessarily be of the same type as its input data: the amount of data may change (from array to scalar or from scalar to array); or the type itself may differ (e.g. from `int` to `float`). Hence the signal handling must be versatile enough to accommodate such data conversions.

An initial idea was to inline the function calls of the preceding processors into the function parameters of the following process. By doing this for the entire model, the need for intermediate value containers is dissolved and is instead taken care of by the C compiler. The disadvantage, however, is that processes may be invoked multiple times for a single input data. While this

[3]The pedantic will note that n is actually the number of processes which can be reached from a model output port, but we will assume that all input models are completely connected.

does not affect the final output from the model, the total execution time may increase dramatically. This approach is also inefficient when operating on arrays, and was thus quickly abandoned.

The remaining option was to use C variables, either local or global, to store the intermediate results. It was decided to use local variables as this will only consume memory when the model is invoked instead of continuously retaining memory as it would with global variables. The usage of local variables can be done in two ways: either a separate variable is used for each signal, or variables are reused or even skipped entirely when allowed[4]. The former is easier to implement but demands more memory. The latter is more memory-efficient but requires sophisticated tools such as liveness analysis (which is also used in register allocation) [2]. To limit the implementation effort, the first method was chosen.

Since a signal is only declared as an edge between two processes, its data type is not immediately available. In fact, the only place where the data type is explicitly stated in the model is in the prototype of the function arguments to the *mapSY* processes. Hence a method was implemented to traverse the model and propagate the data types from the *mapSY* and *parallelMapSY* processes to the signals. An method for doing this is available in Listing 8.7.

The algorithm is simple. Starting from the one of the processes associated with the signal, the network is recursively scanned using DFS until a *mapSY* process is found. Once found, the process' function argument is analyzed. Depending on the search direction, either the data type of the input parameter or the data type of the return value or output parameter is returned. Searching the network in both directions ensures that the entire network is covered. If no data type can be found, then the model cannot be synthesized and is rejected. This algorithm takes linear time to execute for each signal, hence $O(n^2)$ for all signals. The total execution time can be reduced by improving the algorithm to save the data type for a signal once it has been found. This means the method will not only search for the nearest *mapSY* process but also the nearest signal for which a data type has already been discovered.

However, just propagating the data type is not enough. Along the data flow, the data type may change into an array data type or grow in size. For instance, if a *mapSY* process consumes a value of data type `int` which has originated from a *unzipxSY* process, then the data type of the values flowing into the *unzipxSY* process must be of data type `int[]`. The size of the array is then dependent on the number of output signals from the *unzipxSY* process, which must be discovered separately. The same principle can be applied to *zipxSY* processes. Fortunately, discovering and propagating the array sizes can be done using the exact same method as for the data types themselves.

[4]This is possible for instance when a *mapSY* gets its input from an array signal which has passed through an *unzipxSY* and its output is directed to a *zipxSY*. In such instances the input parameter can be taken directly from the input array and its return value written directly to the output array.

```
function FindSignalDataType($S$)
  returns data type for $S$
  inputs: $S$, a signal between an out port of process $P_1$ and an in port of process $P_2$

  data_type ← FindSignalDataTypeFS($S$)
  if data_type was found then
    return data_type
  else
    data_type ← FindSignalDataTypeBS($S$)
    if data_type was found then
      return data_type
    else
      indicate error

function FindSignalDataTypeFS($S$)
  returns data type for $S$ using forward search
  inputs: $S$, a signal between two processes $P_1$ and $P_2$

  if $S$ is a model output signal then
    indicate that data type was not found
  else
    if $P_2$ is a mapSY then
      return data type of input parameter to function argument of $P_2$
    else
      for each out_port in out ports of $P_2$ do
        next_signal ← signal between out_port some in port of another process
        data_type ← FindSignalDataTypeFS(next_signal)
        if data_type was found then
          return data_type
      indicate that data type was not found

function FindSignalDataTypeBS($S$)
  returns data type for $S$ using backward search
  inputs: $S$, a signal between two processes $P_1$ and $P_2$

  if $S$ is a model input signal then
    indicate that data type was not found
  else
    if $P_1$ is a mapSY then
      return data type of return value or output parameter of function argument of $P_1$
    else
      for each in_port in in ports of $P_1$ do
        prev_signal ← signal between some out port of another process and in_port
        data_type ← FindSignalDataTypeBS(next_signal)
        if data_type was found then
          return data_type
      indicate that data type was not found
```

Listing 8.7 – Algorithm for discovering the signal data types.

8.6 GENERATING THE CUDA KERNEL

In Chapter 4, we learned that all GPGPU programs generally follow the same pattern, namely:

1. allocate memory on the device,
2. copy data from host to device,
3. invoke the kernel, which will
    a. access the global memory using the index calculated from the `blockIdx` and `threadIdx` values, and then
    b. perform the data calculations,
4. copy the result back, and lastly
5. free allocated memory.

Remember that all data parallel sections which will be executed on a GPGPU will at this stage been fused into *parallelMapSY* processes. For each such process we generate two new functions: one function is of course the CUDA kernel, which must be defined as a C function prefixed with a `__global__` directive; and another function which takes care of setting up the GPGPU, transferring the data, and invoking the kernel with appropriate parameters. By setting the second function as function argument, the scheduler can manage and execute the *parallelMapSY* processes like any other *mapSY* process. This simplifies the functionality of the scheduler.

Excluding usage of shared memory, generating code for the CUDA kernel is trivial. The code for performing the data calculations has already been provided by the developer as part of the model. Hence the software synthesis component need only generate code which calculates the global memory index for fetching input data and storing output data, and executes the data calculation function. As we expect the input data to be arranged as a 1-dimensional array, the appropriate format of the grid and thread blocks is also 1-dimensional. If i_t denotes a thread's index within the thread block, s_{tb} denotes the thread block size, and i_{tb} denotes the thread block's index within the grid, then the indices for accessing the input and output arrays can be expressed as

$$i_g = i_{tb}s_{tb} + i_t$$

$$i_i = i_g s_{is} \tag{8.1}$$

$$i_o = i_g s_{os} \tag{8.2}$$

where s_{is} and s_{os} are the number of elements consumed and produced by the thread. Lastly, for reasons which will soon become apparent, the call to the data calculation function must be wrapped with an `if` clause which prevents it from being executed if the global index i_g is larger than or equal to the number of data parallel processes in the section.

The other function – let us call it the *kernel wrapper function* – is also relatively simple to implement despite containing more functionality (again excluding usage of shared memory and another issue that we will ignore for now). Device memory management and data copying are all performed with simple CUDA commands, and the grid and thread block configuration can be found using the following equation

$$n_b = \left\lceil \frac{n_p}{s_b} \right\rceil \tag{8.3}$$

where n_p is the number of processes, s_b is the thread block size, and n_b is thus the minimum number of necessary thread blocks. As we want as many schedulable threads as possible per SM, we set s_b to be the maximum thread block size supported by the device. However, this means that we may generate more threads than the number of data parallel processes, which is why we need the `if` clause within the kernel function to prevent the additional threads from executing. If we did not include the guard, then these threads would access invalid memory and potentially result in invalid behavior.

Unfortunately, many NVIDIA CUDA-enabled GPGPUs set a kernel execution time out of about 5–10 seconds. If the kernel has not finished within that time, its execution is aborted. This can be worked around by splitting the work load into many, smaller kernel invocations which each can finish within the time frame. As we cannot modify the amount of work per thread, our only option left is lowering the number of threads issued per kernel invocation. However, we still wish to achieve full utilization of the GPGPU. We devise the following equation

$$n_{full\ utilization} = s_{tb} n_{sm} f \tag{8.4}$$

where s_{tb} is the maximum thread block size supported by the device, n_{sm} is the number of SMs on the device, and f is the number of thread blocks to create per SM (remember that each SM can schedule at most 8 thread blocks; hence $1 \leq f \leq 8$). The exact value of f is dependent on the execution time per thread – kernels with work-intensive or stalling threads which take long time to finish need a lower f value to prevent time out, while kernels with fast-executing threads require a higher f value to reach full utilization. To be on the safe side, f was fixed to 1 in this implementation.

It is possible to lift the time limit, but it must be done in the operating system environment and is thus out of reach from the application. Hence, the most user-friendly solution is to detect at runtime whether the time out is in force and act accordingly.

8.6.1 *Utilizing shared memory*

The shared memory is used to reduce the number of global memory data transfers, e.g. when multiple threads read the same data. Although data sharing is not possible with the ForSyDe constructs covered in this work, shared memory can still be used to potentially achieve higher performance for the ForSyDe models that *are* supported.

Usually, the primary bottleneck of GPGPU kernel performance is the global memory. Hence it is generally advantageous to store data read from global memory in local variables to allow reuse, thus avoiding having to issue more than one read transaction. However, as we learned in Section 4.4, using too many registers per thread may lower the number of thread blocks that can simultaneously reside in an SM. This lowers the amount of available intra-thread parallelism and thus decreases performance.

Since the latency for read and write operations from and to the shared memory is the same as for the local registers[5], the shared memory can be thought of as “additional shared registers”. By first copying the input data from global memory to the shared memory, the extra local variables are no longer necessary. This reduces the register footprint per thread and thus potentially allows more thread blocks per SM. Shared memory can also be used to reduce global memory write operations by allocating storage for the output data, but such functionality is not yet provided by the software synthesis component. However, as the shared memory is evenly divided among the thread blocks, using too much shared memory will lower the number of thread blocks that can be scheduled per SM, thus decreasing performance. These models will therefore only benefit from shared memory usage when thread block schedulability is already constrained by other factors.

As the required amount is decided by the number of threads, there is an upper limit on the thread block size. Furthermore, we want to use the thread block size which minimizes the amount of unused shared memory per SM. This is a special case of the classical *bin packing problem* as it involves packing as many unit size 1-dimensional boxes into a fixed size 1-dimensional box such that the amount of slack is minimized.

An approximation algorithm for calculating the optimal thread block size for this situation is given in Listing 8.8. Starting from the largest allowed thread block size, it iteratively reduces the size until either an optimal configuration is found or until there are no better solutions. An important point to make about this algorithm is that it ignores the register usage per thread and assumes it is low enough to not affect the performance. For a truly optimal grid and thread block configuration, this value must be taken into account. However, it is extremely difficult to get this value as it requires sophisticated analysis of the kernel code, unless it can be output by the CUDA C compiler.

8.6.2 *Data prefetching*

At the beginning of this work, it was assumed that data prefetching had to be applied as shared memory and data prefetching is often used in tandem. However, during the implementation phase, it was discovered that data prefetching is only useful when CUDA cores stall due to a synchronization

[5]This is not entirely true. The shared memory has a limited number of read and write ports which may cause some CUDA cores to stall.

```
function BestKernelConfiguration($N_t$, $S_{tb}$, $S_{sm,u}$, $S_{sm,a}$)
  returns optimal thread block size
  inputs: $N_t$, desired total number of threads
          $S_{tb}$, maximum thread block size
          $S_{sm,u}$, amount of shared memory used per thread
          $S_{sm,a}$, total amount of shared memory available per SM

  $best\_tb\_size \leftarrow 0$
  $lowest\_unused\_sm \leftarrow S_{sm,a}$
  $tb\_size \leftarrow S_{tb}$
  while true do
    $num\_tb\_per\_sm \leftarrow \lfloor S_{tb} / (tb\_size * S_{sm,u}) \rfloor$
    if $num\_tb\_per\_sm = 0$ then
      continue
    $total\_sm\_used \leftarrow num\_tb\_per\_sm * tb\_size * S_{sm,u}$
    $unused\_sm \leftarrow S_{sm,a} - total\_sm\_used$
    if $unused\_sm < lowest\_unused\_sm$ then
      $best\_tb\_size \leftarrow tb\_size$
      $lowest\_unused\_sm \leftarrow unused\_sm$
    if $unused\_sm = 0 \vee num\_tb\_per\_sm > 8$ then
      return $best\_tb\_size$
```

Listing 8.8 – Algorithm for calculating the optimal thread block size when using shared memory.

barrier. As there is no data sharing between the threads, no such barriers are needed and thus we can safely ignore data prefetching without losing performance for the models discussed in this report.

CHAPTER 9

Component Implementation

This chapter discusses the internal mechanisms of the software synthesis component: model parsing and internal representation, data flow, and the synthesis process. The chapter also briefly describes the expected format of the GraphML files which are given as input to the component.

9.1 Overview of the component data flow

The synthesis process consists of three stages: parsing, model modifications, and synthesis (also see Figure 9.1). Parsing involves reading the input file and transforming the model represented in GraphML format to an internal model representation object. This will be covered in greater detail in Section 9.3). Next, the model object is subjected to series of model modifications. The nature of these modifications depend on the target platform for the generated code, but their purpose is to facilitate the latter synthesis process (see Section 7.4.3 for an example of such a modification). The last stage will be covered in Section 9.4.

9.2 Expected input format

Apart from requiring the input file to adhere to the GraphML format, additional specifications are necessary. For instance, each process node requires supplementary data which specifies its process type. The GraphML format allows such data to be annotated to nodes via `<data>` tag (see Section 5.2). By

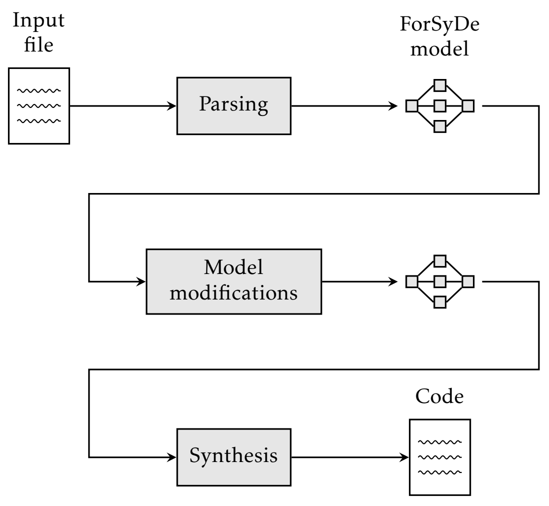


Figure 9.1 – Overview of the component data flow.

examining the `id` attribute, the data can then be interpreted to supplement the processes with whatever additional data they may require. Table 9.1 contains the expected node data along with a brief description.

The input file must also contain two specialized processes – *InPort* and *OutPort*. These are used to indicate the model inputs and outputs, the model inputs being represented as out ports from the *InPort* process, and model outputs as in ports to the *OutPort* process. There must be at most one *InPort* and exactly one *OutPort* process in every input model to the software synthesis component[1]. These are only used to ensure that the model is completely connected; once the model has passed all sanity checks, they are removed from the internal representation.

Whenever a function argument either consumes an array as input or produces an array as output, the array size must be specified. The value is given in the appropriate port of its corresponding process through a `<data>` tag (see Table 9.1). If the function argument both consumes and produces an array, the array sizes need not be equal.

An example of a complete GraphML file is given in Listing 9.1 on page 86 and its represented model is illustrated in Figure 9.2 on page 87.

9.3 MODEL PARSING AND INTERNAL REPRESENTATION

The primary part of synthesizing a ForSyDe model into target code is parsing the model from file into an *internal representation*. The input file is first parsed into an XML document object using an XML parser library called TinyXML++

[1]Note that this does not limit the model to a single model input or output as those are represented by the *ports* of the *InPort* and *OutPort*, not by the *processes themselves*.

ID	Used in	Description
`process_type`	All processes	Specifies the process' type, i.e. which process constructor is used to create this process. Typical values are `mapSY`, `unzipxSY`, and `delaySY`. The value is case-insensitive.

`procfun_arg`	*mapSY* and *parallelmapSY*	Specifies the process' function argument. The value must be such that it declares and defines a C function, i.e. consists of a function prototype and a function body. For example, the function argument to a process which increments the input value by 1 could be declared as follows:
		 `int _func(int x) {`
		 `  return x + 1;`
		 `}`
		 If the code contains one-line comments, then the code must be surrounded by `CDATA` clauses.

`num_processes`	*parallelmapSY*	Specifies the number of data parallel *mapSY* processes that the *parallelmapSY* process represent.
`initial_value`	*delaySY*	Specifies the initial delay value. If the value is a string or character, the it must be declared with surrounding quotes (") or ticks ('), respectively.
`array_size`	Ports of *mapSY* and *parallelmapSY*	Specifies the array size of either the input or output array, depending on whether the port is an in port or an out port.

TABLE 9.1 – List of key IDs for `<data>` tags.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
  <graph id="test" edgedefault="directed">
    <node id="in">
      <data key="process_type">InPort</data>
      <port name="out" />
    </node>
    <node id="out">
      <data key="process_type">OutPort</data>
      <port name="in" />
    </node>
    <node id="div">
      <data key="process_type">mapSY</data>
      <data key="procfun_arg">
        void func(const int* input, float* output) {
          output[0] = (float) input[0] / (float) input[1];
          output[1] = (float) input[2] / (float) input[3];
        }
      </data>
      <port name="in">
        <data key="array_size">4</data>
      </port>
      <port name="out">
        <data key="array_size">2</data>
      </port>
    </node>
    <node id="add">
      <data key="process_type">mapSY</data>
      <data key="procfun_arg">
        float func(const int* input) {
          return input[0] + input[1];
        }
      </data>
      <port name="in">
        <data key="array_size">2</data>
      </port>
      <port name="out" />
    </node>

    <edge source="in" sourceport="out" target="div"
          targetport="in" />
    <edge source="div" sourceport="out" target="add"
          targetport="in" />
    <edge source="add" sourceport="out" target="out"
          targetport="in" />
  </graph>
</graphml>
```

LISTING 9.1 – An example of a GraphML input file. The model consists of two *mapSY* processes, the first accepting a 4-element array and producing a 2-element array, and the second consumes the 2-element array and produces a single value.

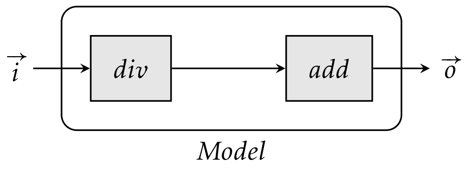


Figure 9.2 – Illustration of the model specified in Listing 9.1.

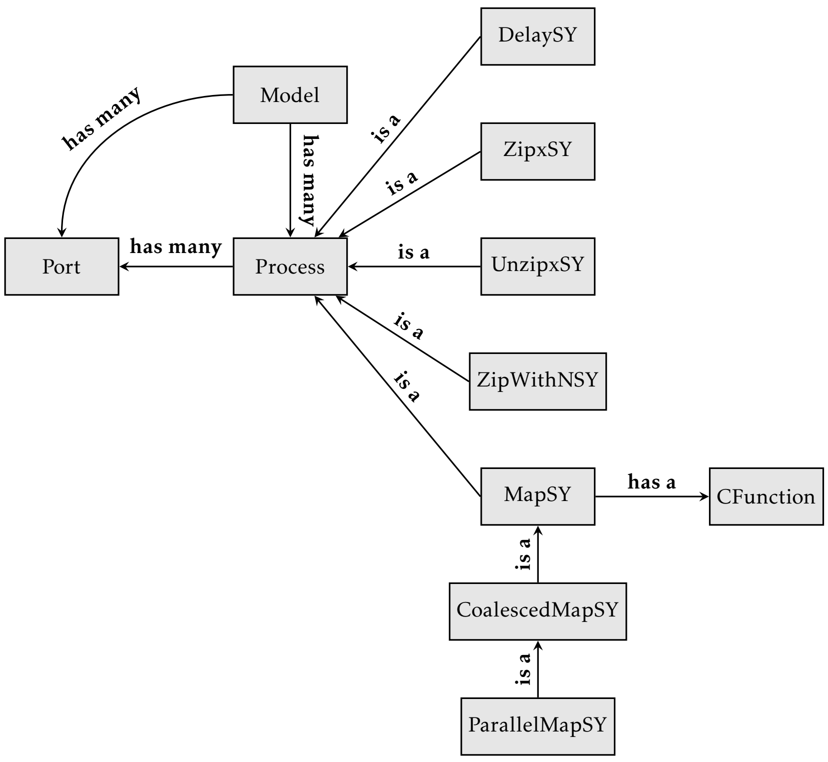


Figure 9.3 – The classes used in the internal representation of ForSyDe models. The labels shall be read along the direction of the arrow, e.g. "`Model` has many `Processes`", "`MapSY` is a `Process`", "`MapSy` has a `Function`", etc.

[48], originally developed by Lee Thomason. The XML structure is then traversed in two sweeps: the first sweep builds the ForSyDe processes from the `<node>` tags; and the second sweep creates the port connections between the processes from the `<edge>` tags. Once composed, the model is run through a sanity check which ensures that the model is valid from a ForSyDe perspective. This also simplifies the latter parts of the synthesis process as less error handling is needed.

The model itself is represented using a collection of C++ classes (see Figure 9.3). In general, the model is nothing more than a directed graph. The `Process` class is the most central entity of the model. It maintains two sets of `Port` objects, which represent the process' in and out ports. All process types derive this class in order to provide uniformity. The `Model` class in turn contains a set of `Process` objects as well as two sets of pointers to the `Port` objects to signify the model's in- and outputs. All `Port` object belongs to a `Process` object and can be connected to another `Port` object, thus establishing a connection between the two processes. Through these connections, the entire model can be traversed.

9.4 THE SYNTHESIS PROCESS

The synthesizer module adopts the following pattern:

1. Find a schedule.
2. Rename function arguments to avoid name clashes and remove duplicates.
3. Create wrapper functions for *coalescedMapSY* processes.
4. Create CUDA kernel functions for *parallelMapSY* processes. If pure C code is desired, wrapper functions are created instead.
5. Create signals for each connection between all pairwise connections.
6. Discover data types for the signals.
7. Propagate array sizes between the signals.
8. Generate header and implementation file, containing the definition of all function arguments and the model execution itself.

All methods and algorithms applied are explained in Chapter 8 except for the final step which will be briefly described in this section.

Defining the function arguments in the implementation file is trivial. One simply needs to iterate over all processes in the schedule and copy the function arguments directly. However, in C a function must be defined or at least declared before it can be invoked, meaning that the function arguments must be copied in reversed order as they appear in the process (which is why wrapper functions are inserted as first function argument once created).

The code for executing the model is declared inside a single function. This function is the public interface that allows the developer to invoke the model. The model input and output data is passed through the function parameters. This allows a model to have multiple outputs as C functions can only return a single value)[2].

For each signal, a local *signal variable* is created inside the model execution function. These act as the intermediate storage containers for the data propagating across the model network as processes are executed. Scalar values are allocated on the stack, while arrays are always allocated on the heap[3].

[2]Attempts were made to return the output values using a `struct`, but this resulted in excessive memory traffic which hampered performance.

[3]The initial approach was to allocated all variables on the stack to simplify memory management. However, this caused segmentation faults for models with large input data as the maximum allowed stack frame size was exceeded.

Moreover, for each *delaySY* process, a `static` *delay variable* is also created. In C, `static` variables declared inside a function retain their values between function calls. To implement the desired behavior, the *delaySY* processes need to be executed in two steps. The first step simply writes the value of its delay variable to the process' output signal. This is invoked for all delay elements prior to executing the schedule. The second step writes the value of the process' input signal to its delay variable, and is executed after all processes have been executed. A salutary effect of this is that the order of the delay elements within the schedule does not matter, as long as they are present (this allows us to use the schedule generated by SFM-BPS; see Section 8.5.1). Note that if signal variables were reused and shared between multiple signals, this approach would have to be modified as the value to store may have been overwritten by another process after all processes have been executed.

The next step is to generate the code which execute the processes. This is done by iterating over the schedule, identifying the process type, and then invoking the appropriate code generator function (currently, the selection is done through a `switch` statement). Letting `CoalescedMapSY` and `ParallelMapSY` derive from `MapSY` allows the synthesizer module to apply the same execution schematics for *coalescedMapSY* and *parallelMapSY* processes as for *mapSY* processes. In this context, the term "execute" is not restricted to triggering function: it also entails transferring values from one signal to another (e.g. as with *unzipxSY* and *zipxSY* processes).

Following this schema allows any model consisting of the supported process types to be synthesized into correct CUDA C or just pure C code without any need for manual modifications.

CHAPTER 10

Component Limitations

This chapter discusses the extent to which data parallelism can be exploited in a model, the supported models and process types, and features which have not been implemented in this work. It also covers some related performance-improving techniques, such as pipelining, and why these are not supported.

10.1 Supported models, process types and data types

At the point of writing, the software synthesis component accepts models containing processes of the following types:

- *delaySY*
- *mapSY*
- *parallelMapSY*
- *unzipxSY*
- *zipWithNSY*
- *zipxSY*

The component is capable to synthesize any process configuration consisting of these types. The maximum number of model inputs and outputs is limited by the number of parameters that a C function may have, which is compiler-specific. The maximum array size of each input or output is equal to the maximum value of an `int` + 1. Models containing unrecognized process types will be rejected.

The software synthesis component also requires that the process function arguments use primary C data types only as parameters or return value, i.e.:

- `char`,
- `unsigned char`,

- `short int` (or just `short`),
- `unsigned short int` (or just `unsigned short`),
- `int`,
- `unsigned int`,
- `long int`, (or just `long`),
- `unsigned long int` (or just `unsigned long`),
- `float`,
- `double`, and
- `long double`.

Function arguments containing any other data type in its parameters are rejected. However, the function arguments may use any data types internally.

Another limitation is that the size of the input data must remain constant. If the size of the input data changes, the model needs to be modified, resynthesized, and the application recompiled. Hence there is no way of changing the input data size at runtime unless the synthesized code is manually altered to allow such functionality. However, this limitation is inherited from the ForSyDe framework, not from the implementation of the software synthesis component.

10.2 NO AUTOMATIC EVALUATION OF BEST EXECUTION PLATFORM

Although the software synthesis component successfully recognizes interprocess data parallelism and generates code to offload execution of those processes on a GPGPU, it makes no consideration whether this will actually improve the performance. For instance, if the input data size is too small, then the cost of the overhead of additional data transfers and kernel invocations may be greater than the gain of using the GPGPU for parallel execution instead of executing the processes sequentially on the CPU (SkePU addresses this problem through execution plans; see Section 3.2). Therefore, before deployment, the designer should generate at least two versions of his model – one C version and one CUDA version – and run tests to compare their performance. In addition, the performance of the CUDA version may also be affected by usage of shared memory, which may require testing of even more versions.

10.3 UNEXPLOITED TYPES OF DATA PARALLELISM

There are many types of data parallelism. Through the *mapSY* processes, the software synthesis component is capable of synthesizing CUDA C code from models which are known as *embarrassingly parallel*, i.e. each computation is done using data which is only used by that task. However, there are other types of data parallelism which have many applications that currently cannot be exploited by the component.

10.3.1 *Reduction data parallelism*

Reduction is the task of computing a single value from a multi-value data set. A common example is sum reduction which has already been covered in Section 4.4.3. Figure 10.1 illustrates a model which exhibits reduction data parallelism. Each level is a data parallel section consisting of processes which operate on data independently from one another.

Although it displays many similarities with the data parallelism described in Section 7.2.1, exploiting reduction data parallelism requires more work. First, its structure can no longer be expressed using single-input *mapSY* processes, but requires *zipWith* or *zipWithN* processes which are capable of accepting two input signals.

Second, ForSyDe lacks a process constructor which denotes this internal structure, forcing the software synthesis component to identify such regions within the model. This demands a much more complex detection algorithm compared to the one proposed in Listing 8.1 on page 68. In comparison, SkePU provides a skeleton called *MapReduce* which allows the entire substructure to be expressed using a single model entity [15], thus absolving the need of any detection algorithm. Doing the same for ForSyDe should not be simple, and a suggestion is provided in Section 13.2.

Third, the kernel function for executing a reduction data parallel region is more complex. The kernel function requires additional input parameters. Also, with new data being computed at each level, the threads must now do multiple sweeps across the input data. As we saw with the sum reduction example in Section 4.4.3, doing this while avoiding thread divergence is not trivial but feasible.

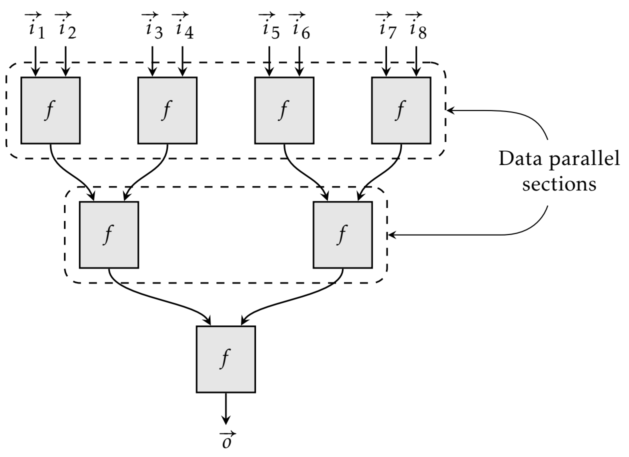

$\vec{i}_1$ $\vec{i}_2$ $\vec{i}_3$ $\vec{i}_4$ $\vec{i}_5$ $\vec{i}_6$ $\vec{i}_7$ $\vec{i}_8$
f
Data parallel sections
$\vec{o}$

FIGURE 10.1 – A model exhibiting reduction data parallelism.

10.3.2 *Data parallelism with partial overlap*

Another type of data parallelism is *partial overlap data parallelism*. Unlike reduction data parallelism, its computational part can be expressed using simple single-input processes such as *mapSY* processes. However, the processes no longer operate on completely independent data; instead, parts of the input data are shared among the processes (see Figure 10.2 for an example). This causes two problems.

First, it is uncertain whether such data-shared data flow can be expressed in ForSyDe. If it could, it would most certainly involve a convoluted pattern of interconnected *unzipxSY*, *zipxSY* and signal value copying, thereby making it impossible to detect this pattern within a model. In comparison, this is trivial to achieve in SkePU using the *MapOverlap* skeleton.

Second, executing such models efficiently on a GPGPU requires a much more competent usage of the shared memory together with data prefetching (see Section 4.4.1 and Section 4.4.4, respectively). Otherwise the application runs a risk of exhausting the global memory bandwidth at the GPGPU, which may limit performance.

10.4 PIPELINING

Another type of parallelism not exploited by the software synthesis component is *pipelining*, a performance-enhancing technique commonly used in many areas of microelectronics. It involves dividing a task into separate stages of computation that can then be executed simultaneously but on different data. Figure 10.4 shows how the model illustrated in Figure 10.3 can be executed with or without pipelining. As the same pattern is present in sequential ForSyDe models, one may argue the following: Since a GPGPU consists of multiple processing units, would it not be possible to allocate a portion of the cores to execute each process, and then forward the data from one portion to another? The idea is also illustrated in Figure 10.5.

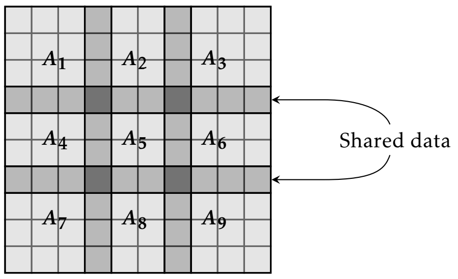


FIGURE 10.2 – A 2-dimensional input data set, where each set consists of a 4×4 submatrix that partially overlap with its neighboring data sets.

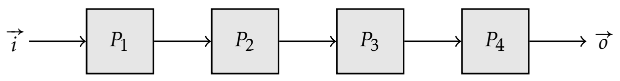
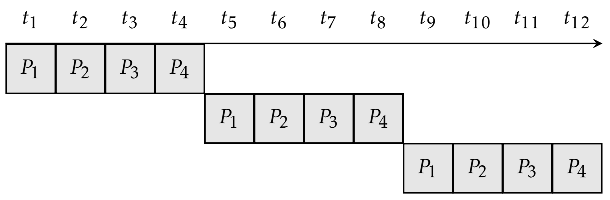
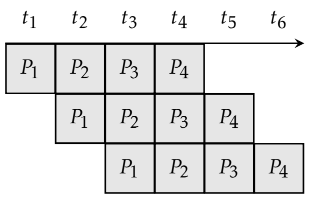


Figure 10.3 – Sequential ForSyDe model of 4 processes.


t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
P_1 P_2 P_3 P_4
P_1 P_2 P_3 P_4
P_1 P_2 P_3 P_4

(a) Time line of the ForSyDe model in Figure 10.3 executed without pipelining.



(b) Time line of the same model executed with pipelining.

Figure 10.4 – Process execution time line of the sequential 4-process ForSyDe model.

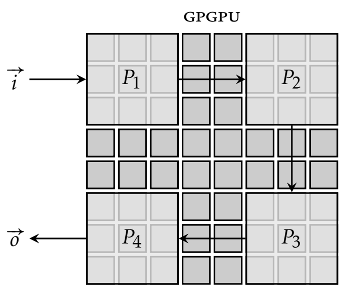


Figure 10.5 – GPGPU resource distribution for pipelining the 4-process ForSyDe model.

Technically, it is possible to include multiple tasks within the same kernel function: an `if-else` statement, together with the thread indices, can be used to control which threads execute which tasks (see Listing 10.1). A similar approach has been tried to merge independent kernel functions into a single kernel in order to reduce the kernel launch overhead and maximize GPGPU utilization [26]. However, this is a naïve solution with two problems.

The first problem is thread divergence. If threads within the same warp execute different paths, they will be interchangeably stalled until all branches have been executed, thus limiting performance. Since warps never cross the thread block boundaries, this problem can be solved by dividing the tasks, not on *thread* level, but on *thread block* level [26].

The second problem is that this solution can only be applied directly if no tasks depend on data produced by another task. If they do, then such tasks must be synchronized in such a way that they are not allowed to proceed until the data is available. With the first generation of NVIDIA CUDA graphics cards, the only way to synchronize threads across thread blocks was to split the steps into multiple kernel invocations [22], which increased kernel overhead. Wu-chun Feng and Shucai Xiao made an attempt to circumvent this problem, but their solution ran a theoretical risk of producing incorrect results [22]. Newer generations with compute capabilities 2.2 and later include functionality which addresses that risk, but it suffered from such poor performance that in the end there was no gain to do this kind of synchronization from within the kernel.

Lastly, it must be stressed that GPGPU are primarily data-driven entities. So even if the problems above *were* addressed and inflicted no performance penalty whatsoever, there is still the overhead of transferring the data back

```
__global__
void kernel(int* input, int* result1, ..., int* output) {
    int index = threadIdx.x;
    if (index >= 0 && index < 10) {
        executeTaskA(input, result1);
    }
    else if (index >= 10 && index < 20) {
        executeTaskB(result1, result2);
    }
    ...
    else {
        executeTaskN(resultN, output);
    }
}
```

LISTING 10.1 – Example of a naïve CUDA kernel function which executes more than one task.

and forth between the CPU and the GPGPU. Due to the way DRAM memories work, the amortization rate increases with larger data transfers (although with diminishing returns). This means that a chain of tasks will only benefit from being executed on a GPGPU when:

1. the input data sets and the number of computations within the tasks are large enough to offset the costs of transferring the data between the CPU and GPGPU and launching the kernels; or
2. each task is so computation-intense that they alone offset the cost of the memory transfers and kernel launches even for single input data entities.

However, such tasks benefit more from being executed on processing units optimized for sequential computations, e.g. CPUs. Moreover, for pipelining to be efficient, each task needs to take about the same amount of time to execute.

10.5 POTENTIAL PERFORMANCE PROBLEM WITH PROCESS COALESCING

Although process coalescing (see Section 7.5.1) appears to be a promising technique to minimize kernel invocations and eliminate redundant memory transfers, it does also introduce another problem.

Data between processes still need to be transferred, coalesced or not. In the algorithm presented Section 8.4, this data is intermediately stored in local variables, thus potentially increasing the register footprint. If the register usage per thread becomes too large, then either the number of thread blocks that can be scheduled on an SM decreases, or variables will be spilled to the thread's local memory which resides in global memory. Both results in a performance decrease.

For such instances, it could be better to leave the processes separate, thus introducing multiple kernel invocations, but at the same time avoiding the redundant data transfers between the invocations. The reasoning is that, while each kernel invocation does introduce some overhead, the cost may be lower than that of the additional global memory operations caused by the process coalescing.

CHAPTER 11

Results and Measurements

This chapter presents the performance results from the models of two test applications that have been synthesized using the software synthesis component. The first application is a Mandelbrot generator, and the second is the line scan application described in Section 7.1.

11.1 PERFORMANCE TESTS

To assess its effectiveness, the software synthesis component was applied on models derived from two applications – a Mandelbrot generator, and the line scan application that was described in Section 7.1 on page 53. We are mainly interested in two measurements:

1. the correctness of the synthesized code, and
2. its performance.

To compare performance, a hand-written single-threaded C version was written for each application to use as a benchmark baseline.

The models had to be designed twice. In the first attempt, the line scan model was designed in Haskell and then written to a GraphML file using the GraphML backend. This meant that the data parallel sections were expressed using *unzipxSY*, *mapSY* and *zipxSY* processes. The file was then manually modified to replace the Haskell functions used as process function arguments with C function equivalents. The Mandelbrot model was designed to follow the same approach as the line scan model, even though the GraphML file was written manually. This produced models where the number of processes and signals was proportional to the size of the model input. As a consequence, the GraphML files could be huge; for example, the GraphML file for representing a

$1,000 \times 1,000$ Mandelbrot generator model required over 850 MB. Even with an Intel Core i7 8 KB L2 at 2.80 GHZ and 8 GB DDR3 RAM at 1333 MHZ, the synthesis process ran out of memory already when the GraphML file exceeded half a gigabyte, and took over an hour for the larger Mandelbrot models it did manage to synthesize successfully.

Since the data parallel sections are internally converted into *parallelMapSY* processes after parsing, it was decided to allow this process type to be used already in the GraphML file. The models were then redesigned using *parallelMapSY* processes. As this disconnected the model size from its input data size, the size of the GraphML files reduced dramatically; in case of the $1,000 \times 1,000$ Mandelbrot generator, the file size shrank from 878 MB to 1.2 KB.

Several variations of the models were generated in order to measure the performance over a ranging amount of input data. The models were synthesized into C and CUDA C code and compiled using *g++* version 4.4.3 and *nvcc* release 3.0 version 0.2.1221, respectively, with default optimization flags. The test cases were executed on two machine: one equipped with an Intel Core i7 8 KB L2 at 2.80 GHZ, 8 GB DDR3 RAM at 1333 MHZ, and an NVIDIA Quadro NVS 290 with 16 CUDA cores, 256 MB DDR2 RAM; and another with similar CPU and DRAM hardware and an NVIDIA Quadro 600 with 96 CUDA cores, 1 GB DDR3 RAM.

To avoid timeout, the data parallel computations were divided into multiple kernel invocations (this functionality was already available in the synthesized CUDA C code produced by the software synthesis component). Each test case was run 10 times and then an arithmetic mean average was calculated from the results.

11.1.1 *Mandelbrot tests*

Generating Mandelbrot images is a classic embarrassing data parallel problem. Each pixel coordinate is converted into corresponding coordinate within a rectangular coordinate window in the complex plane. From this complex coordinate an integer value is computed which corresponds to whether the coordinate is part of the Mandelbrot set. In these tests, the window was bounded by $(-1,-1)$ and $(1,1)$.

Its ForSyDe model consists entirely of a data parallel segment. When this segment is expressed using *parallelMapSY*, the model shrinks to a single process.

The performance results are given in Table 11.1, and the speedup of the synthesized code compared to the hand-written single-threaded C version is shown in Figure 11.1. We see that the synthesized C code performs equally with the hand-written version. As expected, both synthesized CUDA C versions surpass the C versions in all tests. For small input data sizes, the speedup is hindered by the restricted amount of computations which can be offloaded to the GPGPU. As the input data size increases, so does the extent to which

Problem size (pixels)	Execution time (s)			
	C version		Synth. CUDA version	
	HW	Synth.	w/o SM	w SM
10,000	0.54	0.53	0.16	0.20
40,000	2.11	2.11	0.36	0.51
90,000	4.73	4.73	0.69	1.00
160,000	8.40	8.39	1.17	1.68
250,000	13.12	13.12	1.68	2.53
360,000	18.91	18.89	2.38	3.49
490,000	25.72	25.72	3.40	4.64
640,000	33.58	33.58	4.72	6.16
810,000	42.53	42.50	6.18	7.98
1,000,000	52.48	52.45	7.84	9.86

TABLE 11.1 – Performance results from the Mandelbrot tests – coordinate window bounded by $(-1,-1)$ and $(1,1)$, on 16 CUDA cores. Maximum measured standard deviation: 4.48%.

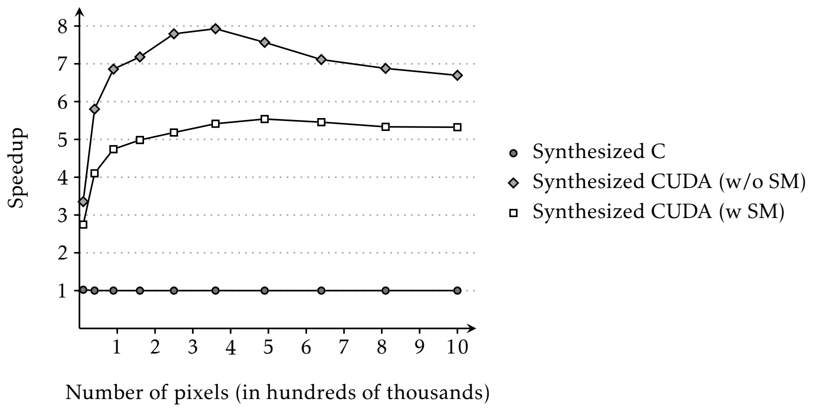


FIGURE 11.1 – Relative speedup of the Mandelbrot tests – large coordinate window, on 16 CUDA cores – compared to the hand-written single-treaded C version.

the GPGPU overhead can be amortized. Using shared memory also appears to greatly reduce the GPGPU utilization rate. This is most likely due to reduced thread-level parallelism as the size of the thread blocks decreases.

However, after reaching a peak of a factor of 8, the results also show that the speedup starts to *decrease* for larger input data. Figure 11.2 shows that this performance decrease is a result of increasing computation time per pixel when executed on the GPGPU. A reasonable explanation for this phenomenon is yet to be found, although a promising theory arose when analyzing how each pixel is processed. Figure 11.3 shows an example of a rendered Mandelbrot image. Coordinates outside the Mandelbrot set only take 2–5 iterations to compute, while coordinates inside the Mandelbrot set require the maximum iteration count. Computations for coordinates on the border land somewhere in between. By visual measurements we see that the coordinate window bounded by $(-1,-1)$ and $(1,1)$ appears to contain slightly more white-colored pixels than black-colored. This would mean that the average number of computations per pixel decreases as the total number of pixels increases. In terms of Amdahl's Law, the parallel section of the total work becomes relatively smaller compared to the sequential section. To make the workload more uniform across the pixels, the window was shrunk to $(-\frac{1}{2},-\frac{1}{2})$ and $(\frac{1}{2},\frac{1}{2})$. The performance results from rerunning the tests on the smaller window are shown in Table 11.2 on page 104, and the relative speedup is shown in Figure 11.4 on page 104. Although the overall speedup has increased, the trend of continuously reduced speedup after 360,000 pixels remains. However, the rate at which the speedup decreases has diminished, which indicates that the hypothesis of increasingly smaller parallel-to-sequential ratio within the larger coordinate window was not entirely wrong and that there are other factors at work.

Lastly, for some coordinates, the integer value produced by the GPGPU differed slightly from that produced by the CPU. Presumably, this discrepancy is caused by differing architectures of the floating point units between the CPU and the CUDA cores.

This space was left blank intentionally.

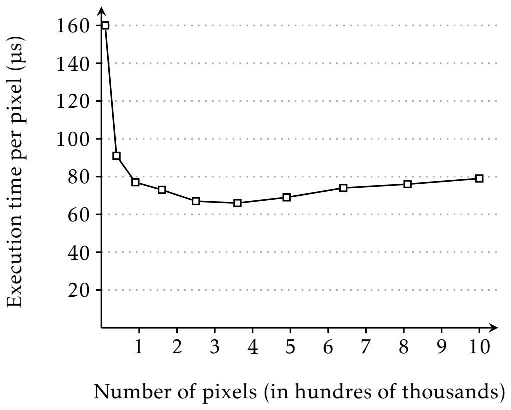


FIGURE 11.2 – Computation time per pixel for the *synthesized* CUDA *(w* PC, *w/o* SM) Mandelbrot-generating code, large coordinate window.

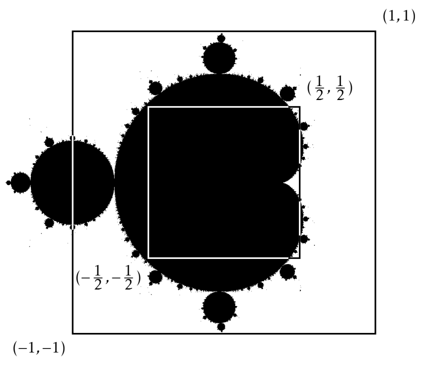


FIGURE 11.3 – A rendered Mandelbrot image, marked with the windows containing the coordinates of interest.

Problem size (pixels)	Execution time (s)			
	C version		Synth. CUDA version	
	HW	Synth.	w/o SM	w SM
10,000	1.29	1.30	0.20	0.27
40,000	5.14	5.14	0.57	0.85
90,000	11.56	11.55	1.17	1.78
160,000	20.53	20.52	1.98	3.05
250,000	32.06	32.05	3.04	4.66
360,000	46.16	46.17	4.32	6.60
490,000	62.81	62.82	5.92	8.87
640,000	82.06	82.06	7.73	11.56
810,000	103.85	103.84	10.05	14.55
1,000,000	128.23	128.14	12.58	17.90

TABLE 11.2 – Performance results from the Mandelbrot tests – coordinate window bounded by $(-\frac{1}{2},-\frac{1}{2})$ and $(\frac{1}{2},\frac{1}{2})$, on 16 CUDA cores. Maximum measured standard deviation: 3.52%.

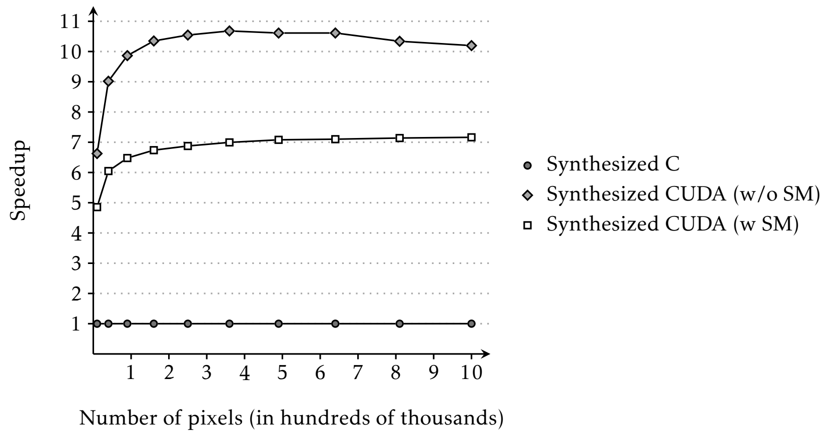


FIGURE 11.4 – Relative speedup of the Mandelbrot tests – small coordinate window, on 16 CUDA cores – compared to the hand-written single-treaded C version.

11.1.2 *Line scan tests*

The performance results are given in Table 11.3, and the speedup of the synthesized line scan code compared to the hand-written single-threaded C version is shown in Figure 11.5. Again, we see that the synthesized C code is on par with the hand-written version. However, the synthesized CUDA C versions show nowhere near the same positive results as with the Mandelbrot tests: although the speedup increases with larger input data size, the GPGPU only provides a mere 31% performance increase at best. In addition, the best-performing CUDA C code also exhibit the same trend of decreasing speedup for the largest problem size that was present in the Mandelbrot tests. Perhaps even more disturbing, the CUDA C code which has been synthesized without process coalescing performs even worse than any of the C versions. If the application's ForSyDe model was evaluated to be such a good candidate for GPGPU execution (see Section 7.2), why then does its synthesized CUDA code perform so poorly?

It turns out that there is another factor affecting a model's suitability for GPGPU execution than just its structure – namely its computation complexity. Through static analysis, the computational code per pixel was estimated to consist of less than 100 instructions. Due to heavy usage of `if-else` statements and `for` loops, the number of actually executed instructions would be even less. We pose the following hypothesis: If the amount of computations executed per thread is small, then the amortization rate of the overhead of transferring the data to and from the GPGPU and launching the kernels can be increased by (i) raising the total number of threads, or (ii) adding more computations per thread. From the performance measurements we see clearly that increasing the number of pixels boosts the speedup (at least for the process-coalesced CUDA C versions), with the break-even point at around one and a half million pixels. As more pixels means more threads to execute, this is an indicator that the first approach is applicable. We saw the same event with the Mandelbrot tests.

To test the second approach, the line scan model was augmented such that the computations for each pixel also contained a loop which incremented a counter 10,000 times, thus increasing the number of computations made per thread. The new performance results, given in Table 11.4 on page 107 and Figure 11.6 on page 107, clearly indicate that the original line scan model suffered from a lack of computational complexity. The line scan tests with original functionality were also rerun on a more powerful graphics card. The results, given in Table 11.5 on page 108 and Figure 11.7 on page 108, show better performance, as expected. However, the speedup was still disappointing when compared to the number of CUDA cores on the GPU.

We also see that, unlike the Mandelbrot tests and the first batch of line scan tests, the performance of the synthesized code for this model does not appear to deteriorate after a certain input data size.

Problem size (7-pixel rows)	Execution time (s)				
	C version		Synth. CUDA version		
	HW	Synth.	w/o PC w/o SM	w PC w/o SM	w PC w SM
1,000,000	0.26	0.27	0.52	0.26	0.26
2,000,000	0.53	0.54	0.98	0.45	0.46
3,000,000	0.79	0.81	1.48	0.65	0.66
4,000,000	1.06	1.08	1.95	0.84	0.86
5,000,000	1.32	1.36	2.34	1.04	1.06
6,000,000	1.59	1.63	2.80	1.23	1.26
7,000,000	1.85	1.90	3.26	1.43	1.46
8,000,000	2.11	2.17	3.82	1.61	1.65
9,000,000	2.38	2.44	—	1.80	1.85
10,000,000	2.64	2.71	—	2.01	2.05

TABLE 11.3 – Performance results from the line scan tests – original functionality, on 16 CUDA cores. The *synthesized CUDA (w/o PC, w/o SM)* version failed when the input data exceeded 800,000 7-pixel rows. Maximum measured standard deviation: 3.50%.

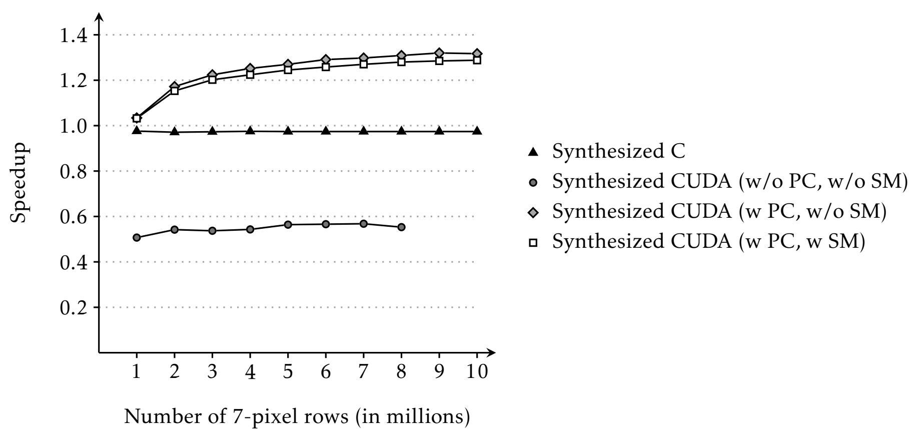


FIGURE 11.5 – Relative speedup of the line scan tests – original functionality, on 16 CUDA cores – compared to the hand-written single-treaded C version.

Problem size (7-pixel rows)	Execution time (s)				
	C version		Synth. CUDA version		
	HW	Synth.	w/o PC w/o SM	w PC w/o SM	w PC w SM
100,000	1.99	1.93	0.37	0.33	0.42
200,000	3.98	3.85	0.68	0.59	0.78
300,000	5.97	5.78	0.99	0.86	1.14
400,000	7.95	7.71	1.30	1.12	1.50
500,000	9.95	9.63	1.61	1.39	1.85
600,000	11.93	11.56	1.92	1.66	2.21
700,000	13.92	13.48	2.23	1.92	2.57
800,000	15.91	15.43	2.52	2.19	2.93
900,000	17.92	17.40	2.83	2.45	3.28
1,000,000	19.89	19.26	3.14	2.72	3.64

TABLE 11.4 – Performance results from the line scan tests – augmented functionality, on 16 CUDA cores. Maximum measured standard deviation: 4.50%.

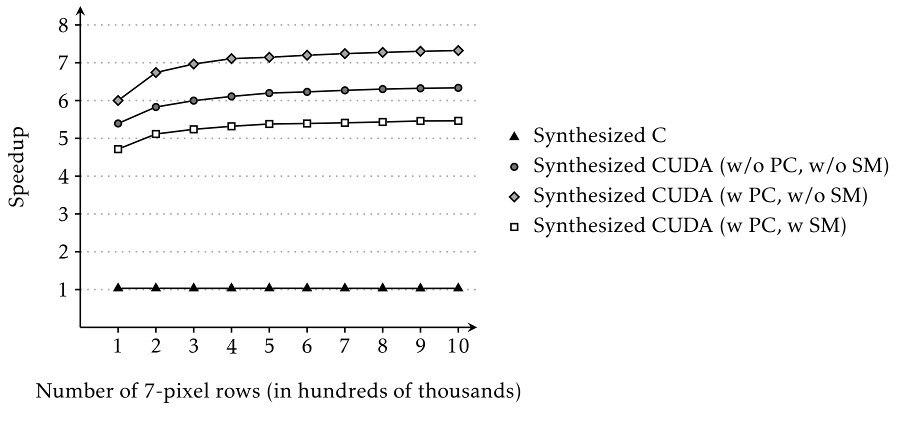


FIGURE 11.6 – Relative speedup of the line scan tests – augmented functionality, on 16 CUDA cores – compared to the hand-written single-treaded C version.

Problem size (7-pixel rows)	Execution time (s)				
	C version		Synth. CUDA version		
	HW	Synth.	w/o PC w/o SM	w PC w/o SM	w PC w SM
1,000,000	0.19	0.21	0.07	0.06	0.06
2,000,000	0.38	0.41	0.09	0.07	0.08
3,000,000	0.56	0.62	0.12	0.08	0.09
4,000,000	0.75	0.82	0.14	0.09	0.10
5,000,000	0.94	1.03	0.16	0.10	0.11
6,000,000	1.12	1.23	0.18	0.11	0.13
7,000,000	1.31	1.44	0.20	0.12	0.14
8,000,000	1.50	1.64	0.22	0.13	0.15
9,000,000	1.68	1.85	0.24	0.14	0.17
10,000,000	1.87	2.05	0.26	0.15	0.18

Table 11.5 – Performance results from the line scan tests – original functionality, on 96 CUDA cores. Maximum measured standard deviation: 6.21%.

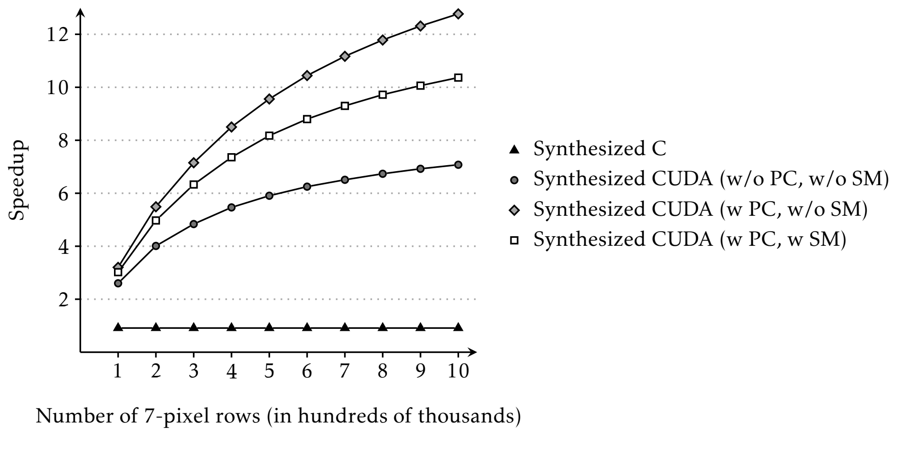


Figure 11.7 – Relative speedup of the line scan tests – original functionality, on 96 CUDA cores – compared to the hand-written single-treaded C version.

Another fact worth noticing is that, even though the code would benefit from decreased global memory traffic, the cost of reducing thread block size exceeds the gain from using shared memory to minimize the number of global memory reads.

In all tests, the produced output was identical for all code versions.

11.2 EVALUATION

The results from the synthesized Mandelbrot and (augmented) line scan models show that the software synthesis component successfully produces correct and efficient C and CUDA C code.

However, with the exception of one test batch, the relative speedup of the synthesized CUDA C compared to the hand-written sequential C code showed a steady decrease after reaching a certain problem size. Although the performance is still acceptable, this is nevertheless a disturbing problem. Three plausible explanations come to mind:

1. The character of the computations is such that the parallel-to-sequential portion ratio decreases with larger problem sizes.
2. The underlying architecture performs worse as the problem size increases.
3. The problem lies in the behavior of the synthesized code.

As this problem is evident in both the Mandelbrot tests and the line scan tests, it is dubious that this is caused by the computations themselves as the Mandelbrot and the line scan tests are completely different in nature. Furthermore, shrinking the coordinate window failed to remove this problem for the Mandelbrot tests, although it did diminish the rate of speedup decrease. It also seems unlikely that the software synthesis component is entirely at fault as the trend did not appear in the tests for the augmented line scan model. My suspicion is therefore that the problem is caused by the underlying architecture, which may or may not be fixed by making appropriate changes in the synthesized CUDA code. To determine the exact cause, however, more research is needed.

The problem above notwithstanding, the synthesized CUDA code still shows impressive speedup compared to its sequential counterpart. When the performance of the CUDA code *was* appalling, the problem was proven to be related to the input model itself. Once addressed, the synthesized CUDA code outperformed both the hand-written single-threaded C code and the synthesized C code, at times with nearly a factor of 11. This is very good considering that the GPGPU was equipped with 16 CUDA cores. The results also showed that process coalescing is an important factor for boosting performance.

Another interesting observation that can be made from these test results is the factors which appear to affect the speedup of the synthesized CUDA code. In the Mandelbrot tests, although the number of input data elements was relatively smaller (in the order of hundreds of thousands) compared to that used in the line scan tests (in the order of millions), the *size* of each input data element was larger (12 bytes compared to 7 bytes). This incurs a higher data transfer overhead per input data element for the Mandelbrot tests compared to the line scan tests. However, due to its higher computation complexity per input data element, the amortization rate is greater in the Mandelbrot tests, thus yielding a greater speedup.

This leads us to the following conclusion: The speedup gained (or lost) when offloading execution of parallel processes on the GPGPU, instead of executing them sequentially on the CPU, appears to depend primarily on

1. the number of input data elements,
2. the size of each input data element,
3. the amount of work performed on each input data element, and
4. the number of CUDA cores.

Weighing in the cost of data transfers and kernel invocation overheads, and of course also the computational power of the CPU, one should be able to devise a formula which can estimate the amount of change in performance when choosing the GPGPU over the CPU. This would allow the software synthesizer component to insert code that will make a runtime decision of which execution platform to use in order to achieve the best performance. Moreover, hardware upgrades would not require the model to be resynthesized, provided that some factors, such as overhead costs, remain unchanged. Unfortunately, lack of time prevented me from investigating this further.

Lastly, we also notice that in none of the tests cases was it beneficial to use shared memory – in fact, it greatly decreases performance. The reason is that doing so lowers the number of thread blocks that can be scheduled per SM; fewer thread blocks means fewer threads to switch in to hide global memory latencies. Hence, it is better to leave shared memory usage for instances where data is shared between multiple threads (e.g., such as in data parallelism with partial overlap; see Section 10.3.2).

III

Final Remarks

CHAPTER 12

Future Work

This chapter suggests future work that would expand the work of this thesis and improve the capabilities of the software synthesis component.

12.1 New software synthesis component features

Despite having already invested a lot of work and time in this thesis project, even more work remains to be done. For instance, the process type support needs to be extended, and the performance of the CUDA C code can be improved. Here follows some suggestions (this list is by no means exhaustive).

12.1.1 *Extending the process type support*

Although the small number of already supported process types was sufficient to test the functionality of the component, it would be beneficial if the support could be extended to include all process types provided by the ForSyDe framework. Consult the ForSyDe literature for a list of currently unsupported process types.

12.1.2 *Avoiding redundant GPU memory transfers between kernel invocations*

SkePU used a notion of lazy GPU memory copying to avoid redundant memory transfers between kernel invocations (see Section 3.2). This greatly boosted the performance of SkePU models, and the same methods should be applicable to the software synthesis component. This would complement the process coalescing technique (see Section 7.5.1), which minimizes kernel invocations but potentially also introduces register spilling (see Section 10.5 for a discussion). Most importantly, this would increase performance when two kernel invocations are performed consecutively and process coalescing is not applicable (e.g. when one data parallel pattern in the model is followed by another).

12.1.3 *Adding C++ support*

Later generations of CUDA-enabled graphics card apparently have support of executing C++ code instead of just pure C code. Allowing C++ code to be used in the process function arguments could potentially allow more powerful applications to be modeled, or simplify already-existing modeling capabilities.

12.1.4 *Supporting dynamic input data sizes*

Currently, the synthesized code requires that the input data size to the model remains static. If input data of ranging sizes need to be processed, then a separate model must be constructed and synthesized for each specific size. A more user-friendly approach would be to allow designers to leave the model's input data size unspecified (although a lower and upper bound may be required). Upon synthesis, the resultant code would be capable of process input data whose size is decided at runtime.

12.2 RESEARCH TOPICS

There are also a number of research topics, whose outcome could be implemented in the component. The topics involve exploitation of more types of data parallelism to more efficient use of the GPGPUS.

12.2.1 *Exploiting reduction data parallelism*

Reduction data parallelism is a common pattern in data parallel applications. To be able to exploit these patterns, methods need to be found for discovering them in a ForSyDe model and generating efficient CUDA code that minimizes thread divergence.

12.2.2 *Exploiting data parallelism with overlap*

Data parallelism with overlap is another common pattern in data parallel applications. It is usually found in image processing and physics algorithms, most often operating on 2D matrices (which means that the process type *overlapMapSY* suggested in Section 13.2 cannot be applied since it only works on 1D arrays). Exploiting these patterns requires methods for identifying them in a ForSyDe model, and competent use of shared memory and data prefetching in order to achieve efficient execution on a GPGPU.

12.2.3 *Determining when GPGPU offloading is beneficial*

Even though a model may exhibit one data parallel pattern or another, it is not always beneficial to use the GPGPU instead of the CPU to execute those sections (we have seen this already with the line scan model; see Section 11.1.2). In order to make this decision, the computation complexity within the data parallel sections needs to be analyzed, which requires static code analysis of the process function arguments. Furthermore, the factors that affect speedup – what they are and how they relate – need to be analyzed (some are identified in Section 11.1.2).

12.2.4 *Exploiting unused memories on the GPGPU*

The GPGPU contains several additional types of memories than just shared memory. If used correctly, and in the right circumstances, they can increase the performance of the application. Research involves finding out what these circumstances are, and how the memories can be exploited.

12.2.5 *Utilizing more than one GPGPU*

If the machine has more than one GPGPU installed, the kernel execution could be split up into multiple invocations, one executed on each GPGPU. However, if the GPGPUs are not equally powerful, the workload must be divided and allocated according to the computation capabilities of each GPGPU in order to achieve maximum speedup. This involves developing an algorithm which performs this division with acceptable accuracy.

CHAPTER 13

Conclusions

This chapter summarizes the thesis work. It briefly reiterates what has been achieved and the test results. It also checks how the outcome matched the objectives set out in Chapter 1, and its contribution to software programming and academic community. The chapter also suggests a set of new process types that should be added to the ForSyDe framework to improve and simplify modeling and synthesis of data parallel applications targeting GPGPUs.

13.1 Summary

In regard to the objectives set out in Chapter 1, this thesis project has been a success:

- an extensive literature study has been performed;
- the minimum set of language constructs and operations that has to be supported has been determined;
- the main challenges have been identified, prioritized, and solved;
- a working prototype of the software synthesis component has been implemented that takes a ForSyDe model representation and produces correct and efficient CUDA C code that can be executed on a GPGPU;
- the software synthesis component has been extensively tested and evaluated; and
- the project was completed within the set time frame.

In this report, we have learned how applications can be modeled at a very high level of abstraction using ForSyDe. However, the framework lacked both a backend for software synthesis into C as well as a means of exploiting data parallelism within the model for execution on a GPGPU – a data parallel execution platform possessing massive computation capabilities. Moreover, GPGPUS are notoriously difficult to program due to the way data is accessed and processed, and many interconnected factors affect the performance of the program. This makes it an exceptionally challenging task to write correct and high-performing applications for GPGPUS.

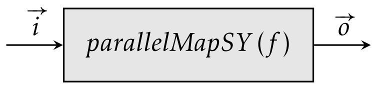
The work in this thesis aimed to address these concerns. It proposed a software synthesis process capable of discovering one type of potential data parallelism in a model and synthesizing either pure C or CUDA C code. To verify its ability, a prototype of the software synthesis component was implemented and tested on models derived from two applications – a Mandelbrot generator and an industrial-scale image processor. The test results showed that the synthesized code was both correct and efficient, except for the image processing model. In that case, although the model was indeed a suitable candidate for parallel execution in terms for structure (it applied the "split-map-merge" pattern; see Section 7.2.1), it lacked sufficient computation complexity to yield speedup when offloaded on the GPGPU. Empirical attempts also showed that models need to be expressed using specialized process types in order to be manageable by the software synthesis component.

To conclude, the contribution of this work to the software programming and academic community is a process which allows applications exhibiting one type of data parallelism to be modeled at a high level of abstraction. Such models can then be synthesized into correct and high-performing CUDA C code using an automated tool. Through this process, the model designer can make use of the massive computational power that the GPGPUS provide, with no or little prerequisite knowledge about the their intricate details.

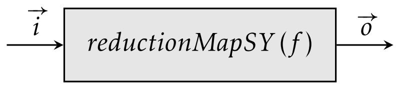
Although currently only allowing a limited set of data parallelism to be exploited, this work provides a good platform onto which to build further support. With more work and research, the process can be extended to include more complex types of data parallelism, thus allowing developers to take advantage of the high level-of-abstraction software methodology to model even more complicated applications.

13.2 SUGGESTED IMPROVEMENTS TO THE FORSYDE FRAMEWORK

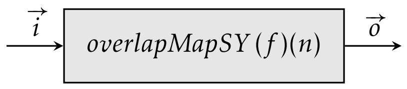
In order to make maximum use of the GPGPU and sufficiently offset the GPGPU overhead, the input data to the model need to be large. For example, the tested Mandelbrot application required 360,000 pixels to achieve maximum speedup on a graphics card with 16 CUDA cores. With each pixel requiring its own *mapSY* process, expressing the data parallel sections through a network of *unzipxSY*, *mapSY*, and *zipxSY* processes yielded extremely large models which proved to be both awkward and time-consuming to synthesize; at a certain point the model become so large that the machine ran out of memory. And for the largest models that the component *did* manage to synthesize, the



(a) Applies a combinatorial function f on every value on the input vector signal.



(b) Applies a combinatorial function f on every pair of values in the input vector signal. The same function is then applied to every pair of produced values until a single value remains.



(c) Applies a combinatorial function f on every value set $\bigcup_{j=i-n}^{i+n} v_j$, where i is the index in the input vector signal.

Figure 13.1 – New combinatorial process constructors *parallelMapSY* and *reductionMapSY*, and *overlapMapSY*.

process took more than an hour even on an Intel Core i7 at 2.80 GHZ.

To address this issue, a new process type – *parallelMapSY* – should be introduced to the ForSyDe framework. With this new process type, the functionality of the entire split-map-merge pattern can be encompassed within a single process, thus disconnecting the model size from being proportional to its input size. This dramatically decreases both file size and processing time and allows applications which process very large amounts of data to be modeled in an elegant fashion. Furthermore, additional process types should also be added to simplify modeling of reduction data parallelism and data parallelism with partial overlap (see Section 10.3.1 and Section 10.3.2, respectively). The suggested improvements are illustrated in Figure 13.1.

IV

Appendices

APPENDIX A

Component Documentation

This appendix describes how to build and use the software synthesis component. It also explains how to maintain the component, in case future developers wish to extend the process type support, add new features, or in some other way improve upon it.

A.1 BUILDING

The software synthesis component was developed and tested on Ubuntu 11.10, but it should work on any Linux distribution[1]. The source code comes with a `makefile` which builds the entire component and requires no additional tools apart from *make*, *g++* and the standard command-line UNIX tools. Hence, to build the component, simply execute

```
$ make
```

This produces the binary `f2cc` (ForSyDe-To-CUDA C) and puts it in the `bin` subfolder along with all generated object files and libraries. To clean the build, execute

```
$ make clean
```

[1] The component could probably also be compiled and executed under Windows, but no warrants are made and most likely requires source code modifications.

The API documentation is available in the docs subfolder, but it can also be generated from the source code by executing

```
$ make docs
```

The newly generated docs, however, are not placed in docs but in the api_tmp subfolder.

A.2 USAGE

The component runs entirely from the command prompt and is controlled through command-line arguments. To synthesize a model file with default options, execute

```
$ ./f2cc file
```

This produces a header file and a CUDA file with the same file names but different extensions than the input file. The output file name can be controlled through the -o switch.

All options must be placed before the input file, but need not follow any internal order. Here follows a list of all available options:

`-o FILE, -output-file=FILE`
: Specifies the output files. Default file names are the same as the input file but with different file extensions.

`-tp PLATFORM, -target-platform=PLATFORM`
: Specifies the target platform which will affect the kind of code generated. Valid options are `C` and `CUDA`.

`-no-pc, -no-process-coalescing`
: CUDA ONLY. Specifies that the tool should not coalesce processes, even when it is possible to do so for the given input model.

`-use-sm-i, -use-shared-memory-for-input`
: CUDA ONLY. Specifies that the synthesized code should make use of shared memory for the input data.

`-lf FILE, -log-file=FILE`
: Specifies the path to the log file. Default setting is `output.log`.

`-ll=LEVEL, -log-file=LEVEL`
: Specifies the log level. This affects how verbose the tool is in its logging and prompt output. Valid options are `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`. Default setting is `INFO`.

`-v, -version`
: Prints the version.

`-h, -help`
Prints the help menu.

As an example, to synthesize the GraphML file `model.graphml` into C code, with the log level set to `ERROR`, execute

```
$ ./f2cc -tp C -ll ERROR model.graphml
```

A.3 MAINTENANCE

It is strongly advised that the reader has read and fully understands the material covered in Chapter 9 as this section is heavily dependent on it.

The component is built from a set of loosely connected modules (see also Figure A.1):

- Config
- Exceptions
- Forsyde
- Frontend
- Language
- Logger
- Synthesizer
- Ticpp
- Tools

The names should be self-explanatory, but a brief description is given anyway.

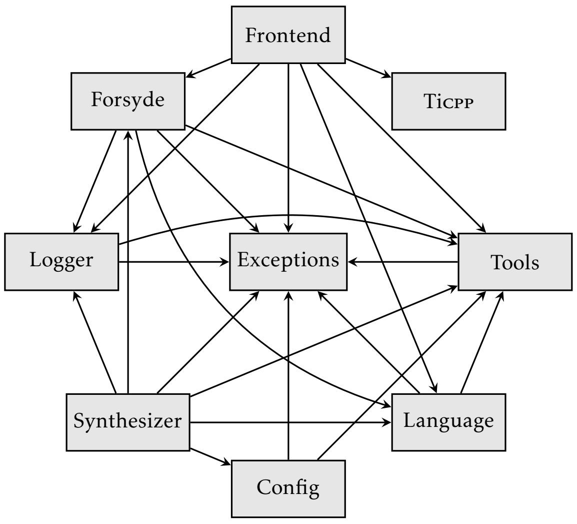


Figure A.1 – Component modules, and how they are connected.

CONFIG Contains user-specified settings for the current program invocation. If new options are to be included, this is where they belong.

EXCEPTIONS Defines all exception classes used throughout the program. New exceptions should always be put in this module, even if the exception is only used within some other module or class.

FORSYDE Defines the module and process type classes, as well as the class for performing the model modifications. When support is added for a new process type, the process type class is put here.

FRONTEND Contains the frontend definitions. Currently this only contains the frontend interface and the GraphML parser. Needless to say, new frontends go here.

LANGUAGE Defines the classes needed for defining the process C function arguments and C data types. This should not need to be modified unless the function argument support is extended.

LOGGER Contains the program logger. This should never need to be modified.

SYNTHESIZER Contains the software synthesizer class, which generates the code output. This is probably where most of the modifications will be done.

TICPP Contains the XML parser. It is highly recommended that this module is left untouched as it is provided by a third party and highly dependent upon by the GraphML parser.

TOOLS Contains common miscellaneous tools used by the other modules. If the developer discovers that the same method is defined across multiple classes, it should be declared once within this module and the affected classes refactored accordingly.

Each module has its own `makefile` which is called when it is built. The file needs to be augmented whenever new CPP files are added to a module, but other than that they should never need to be modified.

A.3.1 *How to extend the process type support*

For a new process type to be supported, three modules must be modified:

- Frontend,
- Forsyde, and
- Synthesizer.

First, a new process type class must be created. The new class must inherit from the `Process` base class and implement the necessary methods (consult some already defined process types for examples).

Once implemented, the frontends must be changed such that the new process type is recognized and parsed correctly. How this is done depends on the frontend implementation, but for the GraphML parser it is sufficient to add an additional `case` to the `switch` statement in the *generateProcess* method (if the process type requires some additional parameters, the change is more complex).

The last modification is done in the synthesizer module. In the *generateProcessExecutionCode* method, add an additional `else-if` statement with the appropriate Boolean check. Although the process execution code could be added directly in the *generateProcessExecutionCode*, it is recommended that the same code convention is followed. Hence a new process execution function should be created, named appropriately (e.g., if the process type is named *NewProcSY*, the function is named *generateProcessExecutionCodeForNewProcSY*). Consult some already defined process execution function for more information.

Once all these steps have been performed, the component need to be recompiled. It is highly recommended that the component is cleaned first and built from scratch.

Bibliography

[1] Alfonso Acosta. *ForSyDe Tutorial*. Stockholm, Sweden: School of ICT, KTH, Sept. 20, 2008. URL: `http://www.ict.kth.se/forsyde/files/tutorial/tutorial.html` (visited on 02/11/2012).

[2] Alfred V. Aho et al. *Compilers. Principles, Techniques, & Tools*. 2nd ed. Addison-Wesley, 2007. ISBN: 0-321-48681-1.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java*. 2nd ed. Cambridge University Press, 2002. ISBN: 0-521-82060-X.

[4] Muthu Manikandan Baskaran et al. "Automatic Data Movement and Computation Mapping for Multi-Level Parallel Architectures with Explicitly Managed Memories". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2008, pp. 1–10.

[5] Albert Benveniste and Gérard Berry. "The Synchronous Approach to Reactive and Real-Time Systems". In: *Proceedings of the IEEE*. Vol. 79. 9. Sept. 1991, pp. 1270–1280.

[6] David C. Black. *SystemC: From the Ground Up*. 2nd ed. Springer, 2008. ISBN: 978-0387699578.

[7] Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner. *GraphML Primer*. June 1, 2004. URL: `http://graphml.graphdrawing.org/primer/graphml-primer.html` (visited on 08/23/2011).

[8] Ulrik Brandes et al. "GraphML Progress Report. Structural Layer Proposal". In: *Proceedings of the 9th International Symposium on Graph Drawing*. GD'01. 2001, pp. 501–512.

[9] Manuel M. T. Chakravarty et al. "Data Parallel Haskell. A Status Report". In: *Proceedings of the Workshop on Declarative Aspects of Multicore Programming*. DAMP'07. Nice, France, Jan. 16, 2007, pp. 10–18.

[10] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009, pp. 43–65, 253–286, 594–612. ISBN: 978-0-262-03384-8.

[11] Nvidia Corp. *DirectCompute Programming Guide*. Version 3.2. Dec. 2010. url: http://developer.download.nvidia.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf (visited on 08/29/2011).

[12] Nvidia Corp. *Nvidia Cuda C Programming Guide*. Version 4.0. May 6, 2011. url: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (visited on 12/21/2011).

[13] Nvidia Corp. *OpenCL Programming Guide for the Cuda Architecture*. Version 3.2. Aug. 16, 2010. url: http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf (visited on 08/29/2011).

[14] Nvidia Corp. *Tesla™ C2050/C2070 Gpu Computing Processor. Supercomputing at 1/10th the cost*. 2010. url: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf (visited on 09/09/2011).

[15] Usman Dastgeer. "Skeleton Programming for Heterogeneous Gpu-Based Systems". Thesis No. 1504. Licentiate thesis. Linköing, Sweden: Pelab – Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, 2011. isbn: 978-91-7393-066-6.

[16] Usman Dastgeer, Johan Enmyren, and Christoph W. Kessler. "Autotuning Skepu: A Multi-Backend Skeleton Programming Framework for Multi-Gpu Systems". In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. Iwmse'11. Hawaii, usa, 2011-05-21, pp. 25–32.

[17] Usman Dastgeer, Christoph W. Kessler, and Samuel Thibault. "Flexible Runtime Support for Efficient Skeleton Programming on Hybrid Systems". In: *Proceedings of the International Conference on Parallel Programming*. ParCo'11. Heraklion, Greece, 2011.

[18] Stephen Edwards et al. "Design of Embedded Systems: Formal Models, Validation, and Synthesis". In: *Proceedings of the Ieee*. Vol. 85. 3. Mar. 1997, pp. 366–387.

[19] Johan Enmyren, Usman Dastgeer, and Christoph W. Kessler. "Towards a Tunable Multi-Backend Skeleton Programming Framework for Multi-Gpu Systems". In: *Proceedings of the 3rd Swedish Workshop on Multicore Computing*. Mcc'10. Gothenburg, Sweden, 2010-11.

[20] Johan Enmyren and Christoph W. Kessler. “SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems”. In: *Proceedings of the 4th International Workshop on High-Level Parallel Programming and Applications*. HLPP’10. ACM. Baltimore, USA, 2010-09, pp. 5–14.

[21] Paul Feautrier. “Automatic Parallelization in the Polytope Model”. In: *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*. Springer-Verlag, 1996, pp. 79–103. ISBN: 3-540-61736-1.

[22] Wu-chun Feng and Shucai Xiao. “To Gpu Synchronize or Not Gpu Synchronize?” In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. ISCAS’10. Paris, France, May 30–June 2, 2010, pp. 3801–3804. ISBN: 978-1-4244-5308-5.

[23] Michael Garland and David B. Kirk. “Understanding Throughput-Oriented Architectures”. In: *Communications of the ACM* 53.11 (Nov. 2010), pp. 58–66.

[24] Michael Garland et al. “Parallel Computation Experiences with CUDA”. In: *Micro, IEEE* 28 (4 July–Aug. 2008), pp. 13–27. ISSN: 0272-1732.

[25] Khronos Group. *OpenCL*. URL: http://www.khronos.org/opencl/ (visited on 02/11/2012).

[26] Marisabel Guevara et al. “Enabling Task Parallelism in the CUDA scheduler”. In: *Proceedings of the Workshop on Programming Models for Emerging Architectures*. PMEA’09. Raleigh, NC, USA, Sept. 2009, pp. 69–76.

[27] Tom R. Halfhill. “Parallel Processing with CUDA”. In: *Microprocessor Report. The Insider’s Guide to Microprocessor Hardware* (Jan. 28, 2008).

[28] Wen-mei W. Hwu. *Champaign: Meeting the Multicore Parallel Programming Scalability Challenge*. Seminar. Stockholm, Sweden: SICS Multicore Day, Sept. 15, 2011.

[29] Simon Peyton Jones and Satnam Sing. “A Tutorial on Parallel and Concurrent Programming in Haskell”. In: *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[30] Simon Peyton Jones et al. “Harnessing the Multicores. Nested Data Parallelism in Haskell”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Ed. by R. Hariharan, M. Mukund, and V. Vinay. FSTTCS’08. IBFI, Schloss Dagstuhl, 2008.

[31] Kurt Keutzer et al. “System-Level Design: Orthogonalization of Concerns and Platform-Based Design”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.12 (Dec. 2000), pp. 1523–1543.

[32] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors. A Hands-on Approach*. Morgan Kaufmann Publishers, 2010. ISBN: 978-0-12-381472-2.

[33] Erik Lindholm et al. "Nvidia Tesla. A Unified Graphics and Computing Architecture". In: *Micro, IEEE* 30 (2 Mar.–Apr. 2010), pp. 39–55. ISSN: 0272-1732.

[34] Steve McConnell. *Code Complete*. 2nd ed. Microsoft Press, 2004, p. 29. ISBN: 978-0735619678.

[35] John Nickolls and William J. Dally. "The Gpu Computing Era". In: *Micro, IEEE* 30 (2 Mar.–Apr. 2010), pp. 56–69. ISSN: 0272-1732.

[36] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly, 2009. ISBN: 978-0-596-51498-3.

[37] Shane Ryoo et al. "Optimization Principles and Application Performance Evaluation of a Multithreaded Gpu Using Cuda". In: *Proceedings of the 13th Acm SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2008, pp. 73–82.

[38] Shane Ryoo et al. "Program Optimization Carving for Gpu Computing". In: *Journal of Parallel and Distributed Computing* 68 (10 Oct. 2008), pp. 1389–1401.

[39] Shane Ryoo et al. "Program Optimization Space Pruning for a Multithreaded Gpu ". In: *Proceedings of the 6th annual Ieee/Acm International Symposium on Code Generation and Optimization*. 2008, pp. 195–204.

[40] Ingo Sander. *Getting Started with ForSyDe*. Tutorial. Stockholm, Sweden: School of ICT, KTH, May 4, 2011. URL: `http://web.it.kth.se/~ingo/forsyde-phd-course-2010/docs/ForSyDe-GettingStarted.pdf` (visited on 02/11/2012).

[41] Ingo Sander. "System Modeling and Design Refinement in ForSyDe". PhD thesis. Stockholm, Sweden: School of ICT, KTH, 2003. ISBN: 91-7283-501-X.

[42] Ingo Sander and Axel Jantsch. "System Modeling and Transformational Design Refinement in ForSyDe". In: *Ieee Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.1 (Jan. 2004), pp. 17–32.

[43] Ingo Sander and Axel Jantsch. "System Synthesis Based on a Formal Computational Model and Skeletons". In: *Proceedings of the Ieee Workshop on Vlsi*. Apr. 1999, pp. 32–39.

[44] Jason Sanders and Edward Kandrot. *CUDA by Example. An Introduction to General-Purpose Gpu Programming*. Addison-Wesley, 2011. ISBN: 978-0-13-138768-3.

[45] Mary Sheeran. “Hardware Design and Functional Programming. A Perfect Match”. In: *Journal of Universal Computer Science* 11.7 (2005), pp. 1135–1158.

[46] Jeol Svensson. “Obsidian: GPU Kernel Programming in Haskell”. Technical Report 77L, ISSN: 1652-876X. Licentiate thesis. Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, 2011.

[47] Joel Svensson, Koen Claessen, and Mary Sheeran. “GPGPU Kernel Implementation and Refinement Using Obsidian”. In: *Proceedings of the International Conference on Computational Science*. Vol. 1. ICCS’10. May 31–June 2, 2010, pp. 2065–2074.

[48] Lee Thomason, Ryan Pusztai, and Ryan Mulder. *TinyXML++*. URL: http://code.google.com/p/ticpp/.

[49] Wikipedia. *DirectCompute*. Last modified at 19:57. Feb. 19, 2011. URL: http://en.wikipedia.org/wiki/DirectCompute (visited on 09/11/2011).