Abstract of thesis entitled

# ”QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft Coarse-Grained Reconfigurable Array Overlay”

Submitted by

**Cheng LIU**

for the degree of Doctor of Philosophy

at The University of Hong Kong

in December 2015

The use of FPGAs as accelerators for compute-intensive loops has been demonstrated by numerous researchers as an effective solution to meet both the performance and energy efficiency requirements across many application domains. However, the design productivity of developing FPGA accelerators remains much lower compared to the use of a typical software development flow. Although the use of high-level design tools may partly alleviate this shortcoming, the lengthy low-level FPGA implementation process including synthesis, placing and routing as well as the complex design space exploration (DSE) dramatically limits the number of compile-debug-edit cycles per day and hinders the widespread adoption of FPGAs.

To address the design productivity problem, this disseration work has developed QuickDough, a framework that can rapidly generate the loop accelerators and their associated hardware-software interfaces. By utilizing a soft coarse-grained reconfigurable array (SCGRA) overlay as an intermediate fabric built on top of off-the-shelf FPGAs, QuickDough partitions the complex accelerator development flow into two paths. Along the rapid and common path, it transforms the loop kernel to

data flow graph (DFG), schedules the DFG to the overlay through a rapid operation scheduling and then generates the FPGA accelerator bitstream through a rapid integration of the scheduling result and a partially implemented overlay bitstream selected from a pre-built accelerator library. By employing different selection algorithms, QuickDough allows users to perform trade-off between performance and compilation time. According to the experiments, QuickDough is able to produce accelerators in the order of seconds with pre-built library while achieving up to 9X performance speedup over the execution of the same software running on a hard ARM processor.

Meanwhile, QuickDough also includes a relatively slow yet less frequent path which pre-builds an overlay based accelerator library targeting a group of applications. To expedite the library generation process, a representative set of accelerator configurations are chosen as the library and generated automatically using a template based system. In addition, intensive application-specific customization is also optional to produce accelerators with optimized performance or energy efficiency. By taking advantage of the regularity of the overlay based accelerators, the customization process can be two orders of magnitude faster compared to an exhaustive exploration while achieving similar performance.

Finally, the underlying SCGRA overlay is the backbone of QuickDough and it is critical to the performance of the resulting FPGA loop accelerators. To that end, a highly pipelined SCGRA overlay template is developed to work at high clock frequency which helps to enhance both the performance and the energy efficiency. Also it is simple, easy to be extended and scalable for the customization specifically to various compute kernels.

---

*An abstract of exactly 436 words.*

# ”QuickDough: A Rapid FPGA Loop Accelerator Design Framework Using Soft Coarse-Grained Reconfigurable Array Overlay”

by

**Cheng LIU**

B.Eng., Harbin Institute of Technology, 2007.

M.Eng., Harbin Institute of Technology, 2009.

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
at The University of Hong Kong

December 2015

# Declaration

I hereby declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this university or to any other institution for a degree diploma or other qualifications.

*Signature:* _____

Cheng Liu

# Acknowledgement

I would like to express profound gratitude to my supervisor in Department of Electrical and Electronic Engineering from The University of Hong Kong, Dr. Hayden Kwok-Hay So. Thank you for giving me numerous suggestions, encouragement and support for this QuickDough research project. Without his guide, this thesis would not be possible. I really appreciate the way you solve problems and the rule you do research. I believe these will be beneficial to the rest of my life.

I would like to thank Dr. Ngai Wong. Although he is not doing research exactly on reconfigurable computing, he gave me a lot of great general suggestions on doing research in the first semester of my PhD study when everything was completely new for me. He also spent quite a lot of efforts pushing me to the right track of doing research and encouraged me when I was confused.

I want to thank the lab mates including Colin, Candice, Mingo, Ho-Cheung and Sam for all the help on my research. Most importantly, I would like to thank you for showing me the various woderful food hiding in Hong Kong. I would like to thank Sharat who gave me a lot of suggestions on the QuickDough project as well. At the same time, I want to thank the buddies including Xiachun Chen, Fang Liu, Ya Wei, Xiaoxu Shi, Big Brother Qun, Prof.Fang, Ruihua Ye and so on for climbing the hills in HK when I was extremely discouraging on my research. It was really helpful to release the stress and resume courage on my PhD study.

Finally, I would thank my family for all the support and understanding on my study in HKU.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Recent years have witnessed a tremendous growth in the use of accelerators in computer systems to improve the systems' performance and energy efficiency [56, 77, 86, 90]. Among these accelerators, GPUs and Xeon Phi accelerators have stood out as two of the most popular choices in spite of their relatively short history as accelerators–5 of the top 10 systems on the top500 list take advantage of them [103]. On the other hand, despite the long and successful track record of FPGA accelerators with promising performance speedup and energy efficiency [8, 23, 90, 95, 100], the use of FPGA accelerators in main-stream systems remains limited and has yet to receive widespread adoption beyond highly skilled hardware engineers [30]. When compared to the wide adoption of GPUs and Xeon Phi accelerators, it is believed that the much lower design productivity in developing FPGA applications, which is generally caused by both the lengthy FPGA compilation and notorious inaccessibility to software programmers, has become one of the major obstacles that hinder the software developers using FPGAs as compute accelerators.

When considering the classical hardware description language (HDL) based FPGA accelerator development flow, a number of aspects lead to the FPGA design productivity challenge. First of all, the FPGA design process using HDL is fundamentally different from that of software development and requires the low-level circuit design techniques which most of the software designers are unfamiliar with. In addition, the relatively low abstraction level of hardware description languages also

limits the efficiency of the application development and thus the designers' productivity eventually [30]. Secondly, instead of spending seconds to compile and debug a quick-and-dirty proof-of-concept design, software programmers soon find that implementing even the smallest FPGA design consumes at least tens of minutes. As the size of their designs increases, the run-time of the implementation tools quickly goes up to hours or even days, greatly limiting the number of possible debug-edit-compile cycles per day [60, 63, 64, 115]. Beyond that, software designers are often put off by the large amount of design optimization options involved in both the design tools and the hardware designs. Moreover, the initial hardware design can rarely meet the requirements and the design optimization is critical to achieve the anticipated performance and energy-efficiency [20, 47, 54, 61, 70, 91, 111, 117].

Although both the industry and academia have made significant progress in increasing the productivity of the FPGA designers by improving the EDA tools for shorter run time [42, 82, 83, 107] and providing a large amount of IPs for design reuse [33, 34], the typical FPGA compilation remains two orders of magnitudes slower than software compilation and the FPGA accelerator design remains a discipline mostly carried out by highly trained specialists. The full potential of FPGAs to the software programmers hasn't been fully unleashed and there is still a long way to drive towards software programmable FPGA allowing high-productivity FPGA development [39, 102].

## 1.1 High-level Synthesis

To bridge the design productivity gap between software and hardware development, many people have turned to the use of high-level synthesis (HLS) techniques in the past decades [30]. Figure 1.1 presents both the FPGA design flow using conventional hardware description language (HDL) and HLS. It is clear that the HDL based FPGA design flow takes HDL as the design entry and thus requires the users to be experienced on low-level circuit design to develop a register-transfer level (RTL) FPGA accelerator for an application. By raising the abstraction level of the phys-

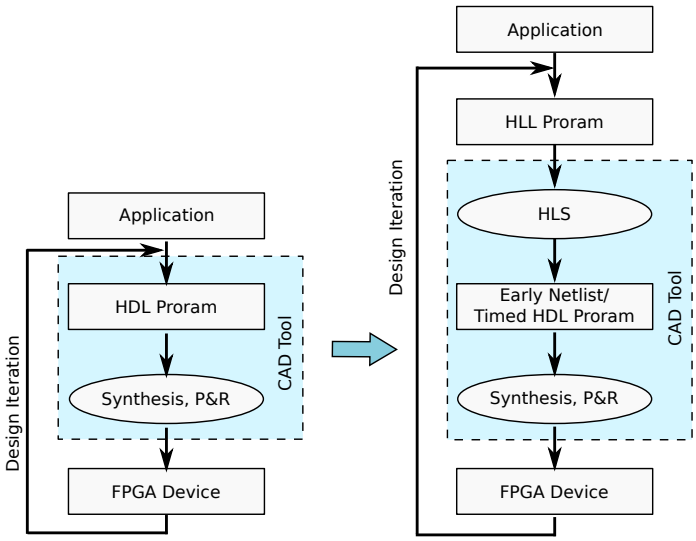



Figure 1.1: FPGA Design Flow Using Conventional HDL and HLS.

ical hardware, HLS based design flow allows designers to develop hardware using familiar high-level, software-like description languages such as C, C++, C-like languages, Matlab and Python [55, 78, 80, 84, 88, 107]. The low-level hardware implementations are then left to the conventional design tools to synthesize and optimize. With decades of efforts, some early results in HLS have already found their ways into FPGA vendors' commercial tools in recent years [25, 55, 80, 107, 116].

Unfortunately, when considering the overall design productivity of developing hybrid hardware/software applications, the raised abstraction provided by HLS is only addressing part of the problem. While the high-level abstraction makes expressing complex functionality on FPGA easier, the lengthy compilation spent in synthesis, placing and routing remains a bottleneck to the overall design productivity for designers who are accustomed to the high speed software compilation. As shown in Figure 1.1, whenever the target application changes to a similar one or even the single application changes slightly during its design iterations, the lengthy compilation process must be repeated. Therefore, the lengthy compilation is dramatically impacting the possible debug-edit-compile cycles achievable per day and thus the design productivity of a designer.

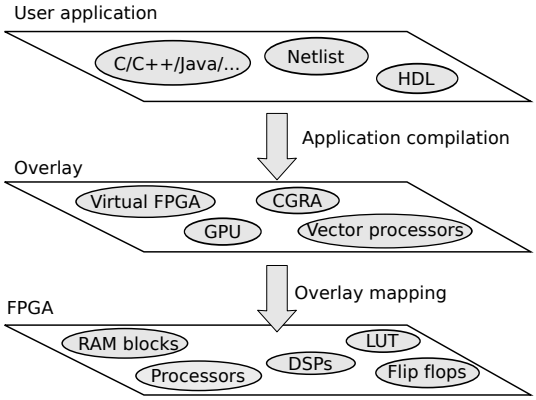



Figure 1.2: Overlay Based 2-layer Approach to FPGA Application Development. Overlay may be designed as a virtual FPGA, or it may implement an entirely different compute architecture such as a coarse-grained reconfigurable array (CGRA), vector processor, multi-core processor, or even a GPU.

## 1.2 Overlay Architecture

Recently there has been an increased interest in employing the concept of overlay architectures which are essentially virtual architectures (VA) overlaying on top of physical architectures (PA) i.e. FPGAs as a way to approach this design productivity challenge [15, 18, 31, 32, 36, 43, 46, 52, 57–59, 67–69]. VA overlay locates between the user application and the underlying physical FPGA similar to that shown in Figure 1.2. With this intermediate VA layer, user applications will be targeted toward the overlay architecture regardless of what the physical FPGA may be. A separate step will subsequently translate this VA overlay, together with the application that runs on VA, on to the PA layer. On the other hand, the overlay built on top of PA layer can be reused for design iterations of the application development or different applications that are mapped to the same PA layer, which helps to amortize the overlay building time and improve the design productivity eventually.

Taking advantage of FPGA's general-purpose configurability, many researchers have demonstrated the benefits of diverse overlays such as virtual FPGAs [15, 31, 43], soft processors [4, 93, 108], massive parallel processing arrays (MPPAs) [13, 45, 58], many-core processor system [67], coarse-grained reconfigurable arrays (CGRAs)[18, 52, 68] and even graphic processing units (GPUs) [3]. With the overlays especially the ones that have coarser granularity, software programmers are

now able to utilize the FPGAs as accelerators simply by writing programs that target a familiar computing architecture instead of working with unfamiliar hardware-centric FPGA programming paradigms. In general, one significant benefit of using FPGA overlay is to bridge the gap between the software programmers and the low-level FPGA hardware fabric. In addition, the overlay architecture helps to hide the physical FPGA details from the application. Therefore the user designs targeting the overlay are portable to FPGAs of different parts and from different vendors.

Of course, if an application is ready to be accelerated on a advanced computing architecture overlay implemented using FPGAs, then it is understandable to come up the question: why not simply run the application on such a high-performance hard computing architecture instead. The answer to this question is actually the key of the overlay research. Not only does the overlay offer desirable software-programmable fabrics on top of physical FPGAs, it also takes advantage of the inherent programmability of the physical FPGA allowing the overlay to be specifically customized to an application or a group of applications concerned for the sake of both performance and energy efficiency. Although the overlay inevitably introduces additional performance penalty and hardware resource cost, a good overlay must ensure that the performance of the resulting overlay based design remains competitive for the accelerator system to be worthwhile while the increasingly larger FPGAs with millions of programmable elements [110] are also able to afford the overlay resource overhead.

## 1.3 Soft CGRA Overlay Based FPGA Accelerator

This dissertation work aims to provide a rapid FPGA accelerator design framework by using an overlay pre-built on top of physical FPGAs while maintaining competitive performance of the resulting system at the same time. The general purpose processor remains a better choice for complex applications such as OS and GUI environment while FPGA is preferable for compute-intensive loop kernels which are usually performance bottleneck in many applications such as signal processing,

image processing, scientific computing and financial computing [14, 97, 101, 105]. Thereby, this work targets a hybrid CPU-FPGA computing system and followed the common wisdom offloading the compute intensive loop kernels to the FPGA accelerator while leaving the rest part of the applications on a general purpose processor [9, 17]. In particular, instead of adopting general FPGAs for random application acceleration, a soft CGRA (SCGRA) overlay is developed for rapid FPGA accelerator design targeting only simple nested loop kernels as demonstrated by numerous CGRA designs on application specific integrated circuit (ASIC) [29, 99].

The proposed FPGA loop accelerator design framework is named QuickDough as shown in Figure 1.3. It takes high-level user-designated loop kernels as input and then maps the loop kernels to the SCGRA overlay based FPGA accelerators. As the SCGRA overlay has coarser configuration granularity than the physical FPGAs, mapping high-level loop kernels to the SCGRA overlay is much faster compared to the direct mapping from the equivalent HDL program to the FPGA directly. In addition, to reduce the lengthy accelerator implementation (i.e. synthesis, placing and routing) time, a representative SCGRA overlay based FPGA accelerator library is pre-built. As the library can be reused by a domain of applications or design iterations of an application, the lengthy FPGA accelerator implementation can be amortized allowing rapid FPGA loop accelerator generation. Meanwhile, QuickDough also produces a communication interface between the host processor and the resulting accelerator, which can be easily called by the original high-level applications. Consequently, the original high-level application can be compiled to the target hybrid CPU-FPGA computing system.

The SCGRA overlay is the backbone of the generated accelerators and has great influence on both performance and energy efficiency of the resulting acceleration system developed using QuickDough. In this work, a template of SCGRA consisting of an array of processing elements (PEs) is developed. It takes advantage of the inherent configurability of the FPGAs and provides just enough functionality to meet the requirements of the target application or a domain of applications. It

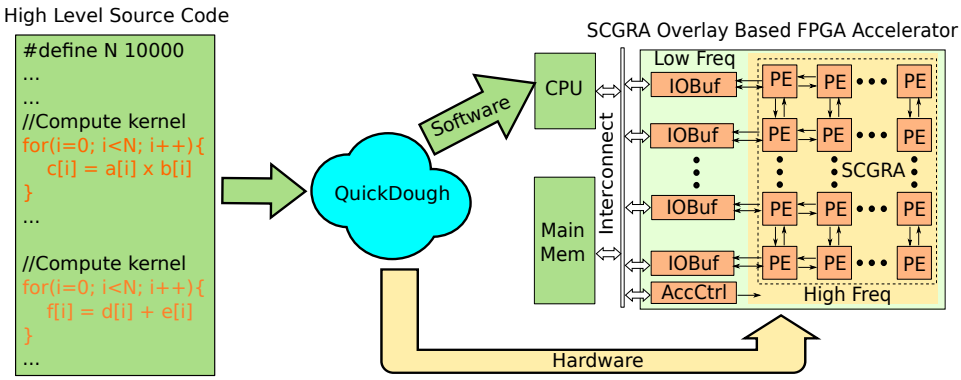



Figure 1.3: QuickDough Overview, it takes user-designated loop kernels as input and generates corresponding hardware acceleration system through a soft CGRA overlay targeting a hybrid CPU-FPGA computing system.

is clear that the SCGRA is light-weight, soft and different from the conventional CGRAs with composable functionality which have to support all the features on the silicon. The SCGRA template is highly pipelined to provide high implementation frequency and good performance of the resulting accelerators accordingly. At the same time, it is simple, flexible and scalable to ensure convenient customization for less resource consumption and higher energy efficiency.

One of the key advantages of using the SCGRA overlay for loop acceleration is the capability to customize the overlay specifically to an application or a domain of applications for higher performance and better energy efficiency. Otherwise, compiling the loop kernels to the SCGRA overlay based FPGA accelerators will not be as useful. However, the compilation process involves a labyrinth of architectural and compilation options and exploring such a complex and large design space is a slow and non-trivial process. To require a user to manually explore the design space is going to counteract the design productivity benefit of using overlay in the first place. To address this problem, QuickDough also supports intensive customization of the design parameters including both the compilation options and the SCGRA overlay architectural parameters all the way from high-level loops to corresponding FPGA accelerator bitstream. The customization process is relatively slow yet provides accelerators with optimized performance and energy efficiency. While it is not a frequent process, it may probably be performed at the final application optimization stage.

## 1.4 Project Contribution

The contribution of the dissertation work mainly includes the following aspects.

- By utilizing the SCGRA overlay as an intermediate fabric on top of physical FPGA, an FPGA loop accelerator design framework QuickDough that can rapidly produce loop accelerator is proposed. With pre-built accelerator library, a high-level loop kernel can be compiled to the corresponding FPGA accelerator bitstream in seconds dramatically increasing the number of debug-edit-compile cycle per day. When compared to a hard ARM processor, the performance of the resulting accelerators developed using QuickDough achieves up to 10X speedup.

- A highly pipelined, flexible and scalable SCGRA overlay template is developed. With this template, the generated loop accelerators with various configurations can work at high frequency and achieve higher performance accordingly. The scalability and flexibility of the overlay template makes it convenient to be adapted to a different application which also helps to automate the accelerator generation.

- By taking advantage of the regularity of the SCGRA overlay based loop accelerator, the design parameters of the overlay based accelerators from high-level loop unrolling, on-chip buffer sizing and overlay configuration are automatically customized efficiently achieving optimized performance and energy efficiency of the resulting accelerators. Compared with an exhaustive search, the proposed customization method can be two orders of magnitude faster while achieving similar performance.

## 1.5 Thesis Roadmap

In Chapter 2, the various approaches especially the overlay architectures that were proposed to address the FPGA design productivity challenges in the past decades

are summarized. Chapter 3 mainly illustrates the SCGRA overlay template showing the flexible reconfigurability for various applications. Low-level hardware design and optimization, especially pipelining, are detailed. In Chapter 4, the proposed FPGA loop accelerator design framework i.e. QuickDough is detailed. Particularly, it focuses on the processes of rapidly compiling the high-level nested loops to the corresponding SCGRA overlay based FPGA accelerators. Chapter 5 presents a two-step customization method, which efficiently explores the design parameters of the resulting FPGA loop acceleration system on a hybrid CPU-FPGA computing machine by taking advantage of the regularity of the SCGRA overlay based FPGA accelerator. Finally, Chapter 6 concludes the dissertation and provides insights into future research inspired by QuickDough.

# Chapter 2

# Literature Review

The use of FPGA as compute accelerators has been demonstrated to be successful in many domains of applications [8, 23, 90, 95, 100]. However, FPGA accelerator development is mostly carried out at relatively low abstraction level and limited to highly trained hardware specialists. As a result, the use of FPGA as accelerators in mainstream computing system is rather limited especially compared to that of Xeon Phi accelerators and GPU accelerators [103] which have much shorter history but get wide adoption in many computing. One of the major reasons that hinder the wide adoption of FPGAs as computing accelerators is the extremely low design productivity caused by both the lengthy FPGA compilation and the low-level design entry barrier. More and more researchers from both industry and academia believe that improving the FPGA design productivity by offering programmable FPGAs to software designers is the key to extend the reach of FPGAs in the future [30, 39, 46, 87, 102].

Over the past decades, significant progress has been made to enhance the design productivity of the FPGA development and many different approaches have been explored. This section will review these approaches and is organized as follows. The approaches developed for improving the productivity of the FPGA development will be classified and briefly introduced in next section. Then the overlay architectures that are most relevant to this work will be reviewed in detail. Finally, related work of overlay customization specifically to target applications are

presented.

## 2.1 Approaches to Improve FPGA Design Productivity

In order to improve FPGA design productivity, many approaches from various angles focusing on the different processes of FPGA development including raising the FPGA abstraction level of the design entry, reducing FPGA implementation time, automating the design space exploration, providing hardware/software support during run-time and compile-time and offering FPGA debugging facilities have been proposed over the years. They will be introduced in the rest part of this section.

### 2.1.1 Raising the Abstraction Level

Recent advances in HLS tools have significantly raised the abstraction level and lowered the design entry of the FPGA development [30]. Instead of using hardware description languages (HDLs) for hardware design at register transfer level (RTL) or behavioral level, high-level languages such as C/C++, OpenCL, Python and Matlab [17, 19, 55, 78, 80, 84, 88, 107], which software designers are likely to be familiar with or at least comfortable to pick up, are used for FPGA development. When the applications or algorithms are implemented with high-level languages, the resulting high-level language programs are then compiled to logic gates which are usually expressed in HDLs. Afterwards, the logic gates can be further mapped to the underlying FPGA fabrics automatically which is the same with the conventional HDL based FPGA development. Ideally, the HLS tools make the FPGA low-level design details transparent to the application developers and thus lower the design entry of FPGA development. In addition, HLS tools typically allow complex functionality to be expressed more easily compared with conventional hardware design using HDLs. This also helps the users to develop complex hardware system more efficiently.

Another way to raise the abstraction level is to build an advanced computing architecture overlay such as soft processor overlays, CGRA overlays and GPU overlays on top of the fine-grained FPGA fabrics [18, 24, 36, 57, 58, 62, 68, 94, 112]. Instead of programming on fine-grained FPGA fabrics, the designers may program on these overlays. These overlays with advanced computing architectures are typically easier for software designers to program with high-level languages. Therefore, the use of overlay also helps to lower the barrier-to-entry for software programmers and to make use of FPGAs with much higher design productivity.

### 2.1.2 Providing HW/SW Support

Some researchers also explored facilities to support mixed hardware-software co-design in a unified language and run-time environment [2, 6, 26, 37, 74, 96] on hybrid reconfigurable computing systems. For instance, ReconOS operating system [74] offered unified operating system services for functions executing in software and hardware and a standardized interface for integrating custom hardware accelerators. BORPH [96] viewed FPGA programs as UNIX processes that could communicate externally by means of UNIX pipes. CoRAM and LEAP [26, 37] investigated memory abstractions for FPGAs and provided a virtual memory environment to simplify the development. It is clear that the features provided by these systems can significantly lower the entry barrier for software designers and help to make use of FPGA for application acceleration on a hybrid CPU-FPGA computing system.

### 2.1.3 Reducing FPGA Implementation Time

Unlike compiling software programs, implementing a hardware design on an FPGA using standard hardware design tools can take dozens of minutes for smaller designs and upward of days with the largest designs. The disproportionally long run-time dramatically hinders productivity. To address the lengthy FPGA implementation problem, some researchers tried to make quality-run-time trade-offs [83, 89], devel-

oped novel implementation algorithms [76, 98, 104] and parallelized the implementation tools to reduce the hardware implementation time directly [33, 34, 42, 82]. Other researches took advantages of the dynamic partial reconfiguration capabilities of modern FPGAs to shorten the run-time by reusing the unchanged part of the design and reducing the design size that needs to be recompiled [11, 38, 48, 53]. Yet another group of researchers approached the problem from a higher level, innovating on how these implementation tools are being used from a design methodology's point of view. By using pre-built hard-macros, modular design flows were explored [60, 64–66]. While these approaches significantly reduce the hardware implementation time, they remain at least two orders of magnitude slower when compared to a typical software compilation process.

### 2.1.4 Offering FPGA Debugging Facilities

Debugging is an essential part of the FPGA design tool chain. However, the traditional FPGA design methodology relies heavily on cycle-accurate simulations for application development and debugging [5, 79, 109]. While such simulations are important to understand the low-level operations of FPGA, they are slow, tedious and provide only limited information about the run-time behavior of the design. Most importantly, they are rarely accessible to software designers without much circuit design experience. A few debugging tools on top of the latest HLS tools emerged recently [41, 49, 50, 85], and they are able to provide software-oriented debugging on top of the latest HLS based design tools. However, many challenges still exist in these tools such as lack of multiple-clock domain circuit debugging support, limited debugging coverage due to the on-chip block RAM constraint and insufficient hardware-software co-design etc. FPGA debugging facilities that target software designers are still in their infancy.

### 2.1.5 Automating Design Space Exploration (DSE)

Hardware design typically involves a labyrinth of architectural parameters and compilation parameters. Fine-tuning these design parameters is a slow, error-prone and non-trivial process. Requiring a user to manually explore the design space for an optimized design thus negatively affects design productivity as well. To address this problem, a number of automatic DSE algorithms have been proposed over the years. All these automatic DSE frameworks were developed on top of the HLS tools [20, 47, 61, 70, 91, 117].

It is difficult to predict the performance and resource consumption of the HLS tools, so they typically considered an HLS tool as a black box. A number of generic algorithms were adopted to perform the DSE and customization for better trade-off between performance and resource consumption. The authors in [47] adopted a genetic algorithm to the problem of identifying Pareto optimal solutions of time and area design space using HLS. They showed that a complete DSE would take quite a lot of design effort and a two-stage fitness function could effectively reduce the DSE time while obtaining a reasonable subset of the Pareto optimal solutions. The authors in [20] proposed a machine-learning based predictive model DSE for HLS tools. With a given error threshold, the predictive model avoids time-consuming synthesis and simulation of different HLS synthesis configurations. Compared to a generic simulated annealing algorithm, it is around 2 times faster while achieving similar results. The authors in [70] studied the application of learning based methods to HLS based DSE. Based on Random-Forest learning model, transductive experimental design and randomized selection, the proposed design methods can effectively find an approximate Pareto set of designs. Divide and conquer method was used in [91] and a calibration tree algorithm was used in [61]. While it is even more difficult to predict the FPGA implementation especially the timing information due to the complex and unpredictable placing and routing processes on a piece of irregular hardware design generated using HLS tools, the number of cycles are usually used as the performance metric and timing related metrics such as power and energy

consumption are rarely available for these HLS based DSE methods.

## 2.2 Overlay Architectures

Recently there has been an increased interest in applying the concept of overlay architectures and quite a few works on FPGA overlays have been developed [4, 7, 13, 15, 16, 18, 24, 31, 36, 43, 45, 59, 62, 68, 71, 75, 93, 94, 108, 112, 114]. By building an advanced computing architecture on top of FPGA devices, the users can program on top of these advanced computing architectures instead of the fine-grained low-level FPGAs. Thus it helps to enhance the designers' productivity from multiple aspects including raising the abstraction level of FPGA design entry, facilitating HW/SW co-design and FPGA debugging, reducing the implementation time for mapping applications to FPGAs and simplifying the design optimization. Therefore, it has become one of the most promising methods to make FPGA programming accessible to software designers. In this section, various types of FPGA overlays that have been explored in the past decade will be presented.

### 2.2.1 Virtual FPGA Overlay

One of the most easiest to understand categories of overlays are virtual FPGAs [15, 31, 43, 59, 75]. They are built either virtually or physically on top of the off-the-shelf FPGA devices. These overlays have different configuration granularity but typically have coarser configuration granularity than a typical FPGA device. By providing an additional layer, they can be used to improve application portability and compatibility over FPGAs of different parts or even different vendors. Furthermore, because of the coarser-grained configurable granularity, implementing designs on such overlay is relatively easier than on a fine-grained device. However, the additional layer imposes restrictions on the underlying fabrics' capability and usually results in moderate hardware overhead and timing degradation.

The authors in [75] developed a relatively fine-grained virtual FPGA as firm

cores expressed using structural VHDL. The virtual layer provides effective portability yet incurs relatively high performance and hardware overhead. Through the utilization of LUTRAMs as reprogrammable MUXs and LUTs, the authors in [15] could further reduce the resource consumption considerably while maintaining the routability and mapping efficiency. In [43], Grant et al. proposed a time-multiplexed virtual FPGA CAD framework MALIBU. The virtual FPGA adopted in MALIBU has both fine-grain and coarse-grain processing elements integrated into each logic cluster and can be used to reduce the compilation time significantly with moderate timing penalty. Coole and Stitt also proposed another island-style coarse-grained overlay called Intermediate Fabrics [31]. It uses coarse-grained operators such as adders instead of logic clusters and routes data through 8 to 32 bit buses achieving both portability and fast compilation. Koch et al. developed a fine-grained FPGA overlay in [59] to implement customized instructions on FPGAs. It allows the execution of a portable application consisting of a program binary and an overlay configuration in a completely heterogeneous environment.

### 2.2.2 Coarse-Grained Reconfigurable Array Overlay

Another category of overlay architecture employed is coarse-grained reconfigurable array (CGRAs) [18, 36, 58, 68, 94] which is essentially an array of connected processing elements with limited computing capability and reconfigurability. The use of CGRAs provides an efficient trade-off between flexibility of software and performance of hardware especially for compute intensive applications as demonstrated by numerous earlier CGRAs on application-specific integrated circuits (ASIC) [29, 99].

In one of the earlier works in this area [94], a customized CGRA overlay called QUKU was developed for DSP algorithms. It featured a two-level configuration including a high-speed configuration and a low-speed configuration. The high-speed configuration was used for operator reuse within an application and the low-speed reconfiguration was used for optimization between different applications. Due to

the limited flexibility, it targeted only a limited number of DSP algorithms. Kissler et al. developed WPPA (weakly programmable processor array), a VLIW architecture based parameterizable CGRA overlay [58]. It featured an interconnection wrapper unit for each processing element (PE) that could be used for dynamic topology customization. In [36], Ferreira et al. proposed a heterogeneous CGRA overlay with a global multi-stage interconnection on FPGA. This CGRA overlay is essentially an array of heterogeneous mathematical operators and there are no local register files near each operators except the pipeline registers. The simplicity of the overlay allows very fast application mapping. Compiling applications onto the overlay takes only milliseconds for smaller DFGs. The authors in [18] built a generic high speed mesh CGRA overlay. This work particularly focused on the overlay implementation. By using the elastic pipeline technique, it achieved high implementation frequency and thus high throughput. It adopted a data-driven execution flow and was suitable for smaller pipelined DFG execution. The authors in [52] proposed an island-style CGRA overlay that is constructed centering the primitive FPGA DSP blocks to achieve high-frequency implementation and high throughput result. Previous CGRA overlays have demonstrated the potential computing capability and promising compilation speed, but there are still little work on using the overlay for loop kernel acceleration on a hybrid CPU-FPGA computing system.

### 2.2.3 Processor-Like Overlays

A third category of overlay is processor-like design including soft general purpose processor [4, 24, 62, 108, 112], massively parallel processing arrays (MPPA) [7, 13, 45], many-core processor [67] and GPUs [57]. The main concerns of using the overlays in the third category are compatibility and usability of the overlay from a user's perspective. With the processor-like overlay, the users can essentially program FPGA by writing software programs. Consequently, there is almost no barrier for software designers to make use of FPGAs. To provide the required performance, these overlay architectures should remain configurable specifically to applications

with ample parallelism.

General purpose soft processor is one of the most widely used on FPGA and many such processors have been developed from both academia and industry [4, 24, 62, 108, 112]. Microblaze and Nios II are two commercial embedded soft processors from Xilinx and Altera [4, 108]. They adopt Reduced Instruction Set Computing (RISC) ISA and allow intensive customization including instruction set and the major architectural parameters such as cache architecture. Moreover, the micro-architectures of the processors are specially optimized for their own FPGA devices as well for higher performance. In academia, a number of soft processors have also been developed [24, 62]. Cheah et al. developed a lightweight soft processor that made best use of underlying Xilinx primitive DSP48E1 slices for a load-store processor. In the work [62], the authors also explored the potential of the underlying FPGA fabrics and developed a soft processor named Octavo. The processor was highly pipelined and could work at 550MHz on Stratix IV. Meanwhile, it is also highly parameterizable and customizable. In the work [112], the authors concentrated on application-specific customization of the soft processors and exhibited promising performance speedup as well as resource saving through the customization.

Beyond these processors, a number of VLIW soft processors [7, 13, 45], vector processors [93, 114] and specialized processors [16, 71] have also been explored on FPGAs in the past decade. Boppu and Hannig et al. a VLIW architecture based parameterizable MPPA overlay [13, 45]. Each processing element in the MPPA is basically a simplified VLIW core executing sequential instructions while the PEs can execute in parallel. Yiannacouras et al. explored the use of a fine-grained scalable vector processor for code acceleration in embedded systems [114]. Later in [93], Severance and Lemieux proposed a soft vector processor named VENICE to allow easy FPGA application development. It accepts simple C program as input and execute the code on the highly optimized vector processor on the FPGA for performance. The works in [16, 71] mainly focused on optimization of network

processor overlays on FPGAs.

In the work of MARC, Lebedev et al. explored the use of a many-core processor template as an intermediate compilation target [67]. In this work, they have demonstrated the improved usability while highlighting the need for customizing computational cores for the sake of performance and resource consumption. Finally, a GPU-like overlay was proposed in [57] and this work demonstrated the good performance while maintaining a compatible programming model for the users.

## 2.3 Overlay Customization

Not only does the overlay offer desirable software-programmable fabrics on top of physical FPGAs, it also takes advantage of the inherent programmability of the physical FPGA allowing the overlay to be specifically customized to an application or a group of applications for the sake of both performance and energy efficiency. While the application specific customization requires to navigate through a labyrinth of design parameters, it is almost impossible for a user to manually explore such a vast design space. Thus automatic overlay customization is critical to the adoption of the advanced computing architectures.

On top of the coarse-grained FPGA overlay [31], Coole and Stitt proposed to provide the overlay with limited flexibility instead of full configurability specifically to a group of design [32]. With this customization, the area overhead was reduced significantly. While most of the virtual FPGAs are typically developed for design portability and thus aim to provide a relatively general architecture to a broad range of applications, there are few works focusing on the application specific customization.

Soft processors are the most widely used overlay architecture and there have been a great number of frameworks and design methods from both industry and academia developed for application specific optimization [4, 22, 27, 35, 44, 51, 73, 108, 113]. The works in [4, 27, 44, 73, 108] typically relied on a parametrized cores allowing a limited aspects of architectural parameters to be tuned for the tar-

get applications. The design space is highly constrained, but the exploration can be relatively fast. Most importantly, the well-tuned parametrized cores typically provided efficient hardware implementation. While the works in [22, 35, 51, 113] provided a system view of the soft processor customization based on architectural description languages providing a complete solution including the creation of the customized compilers, instruction set simulators, cycle accurate simulators and so on. A survey of the soft processor customization can be found in [21, 40].

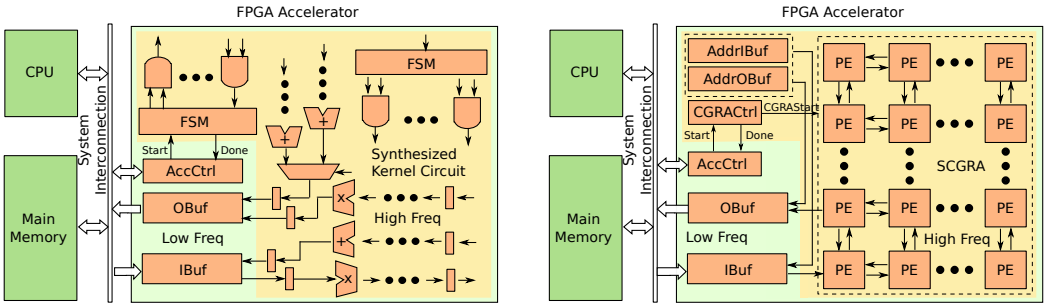
Customizing the CGRA specifically for an application or a domain of application provides promising performance improvement while saving the hardware resource at the same time as demonstrated in CGRA work targeting ASIC design [28, 81, 118]. While CGRA customization on ASIC is relatively limited due to the tape-out cost, CGRA overlays allow more intensive architectural customization providing just enough hardware to the target application or application domains because of the FPGA's inherent programmability. In [12], the authors formalized the loop acceleration on a regular processing array overlay on FPGA. They focused on the hardware resource constrain, IO bandwidth constrain and the loop parallelism partition while processing architectural design parameters were not included. In [68], Lin and So proposed a soft CGRA overlay for rapid compilation high-level loops with fully unrolling to the overlay. In particular, they demonstrated that customizing the overlay connection between PEs on a per-application basis improves the energy-efficiency in the expense of longer tool run time.

# Chapter 3

# Soft CGRA Overlay Based FPGA Accelerator

Instead of developing random applications on FPGA, this work targets a hybrid CPU-FPGA computing system while it offloads the compute intensive loop kernels to the FPGA for performance acceleration and leaves the rest of the control intensive part on the CPU. Figure 3.1(a) shows the design of a typical FPGA accelerator system. In such a system, on-chip memory is used to buffer data between the host CPU and the accelerator. A controller is also presented in hardware to control the operations of the accelerator as well as memory transfers. The controller attached to the system interconnection works at a separate clock domain with lower frequency. The entire design must be reimplemented every time a change is made to the accelerator design, going through the lengthy low-level hardware implementation tool flow.

On the other hand, Figure 3.1(b) shows the system generated on top of SCGRA overlay. While it features a similar overall design as a typical accelerator system, it utilizes a regular SCGRA overlay instead of irregular random logic to implement the computation and the overlay can be reused during the design iterations or within a domain of applications. The SCGRA consists of an array of simple processing elements (PEs) connected by a direct network executing synchronously. Each PE




(a) Conventional FPGA Accelerator (b) SCGRA Based FPGA Accelerator

Figure 3.1: Hybrid CPU-FPGA Computation System

computes and forwards data in lock steps, allowing deterministic multi-hop data communication that overlaps with computations. The action of each PE in each cycle is controlled by an instruction ROM that is populated with instructions generated by the compiler. Finally, a data memory is featured on each PE to serve as a temporary storage for run-time data that may be reused in the same PE or be forwarded in subsequent steps. Communication between the accelerator and the host processor is carried through a group of input/output buffers. Accesses to these I/O buffers from the SCGRA array take place in lock step with the rest of the system. The exact buffer location to be accessed is control by the AddrIBuf and AddrOBuf blocks. Both of them are ROM populated with address information generated from the SCGRA compiler which will be detailed in the next chapter. Finally, the kernel CGRA computing logic may repeat the same computing a number of times on a block of data stored in input buffer i.e. IBuf. For that, an additional CGRACtrl block is used to translate the controlling signals from AccCtrl block.

One key factor that determines the accelerator's performance rests on the design of the SCGRA overlay. In particular, the accelerator will not be as useful without significant performance speedup. For that, the overlay should be *reconfigurable* so that it can be rapidly customized specifically to an application or a domain of application for higher performance as well as energy efficiency. Meanwhile, the SCGRA overlay must be highly *pipelined* and make best use of the underlying FPGA fabrics to work at high frequency. In addition, the SCGRA overlay should also be *deterministic* for the sake of efficient lock-step computing.

## 3.1 Reconfiguration

There are two levels of reconfiguration that may be applied to the overlay to address different application needs. The first and the quickest form of reconfiguration keeps the physical implementation of the overlay intact. To modify the function of the implemented hardware, the SCGRA overlay can be configured by changing (i) the content of the instruction ROM of each PE, (ii) the content of the input/output buffer, (iii) the content of the I/O buffer address ROM AddrIBuf and AddrOBuf, and (iv) the accelerator control AccCtrl. Among these 4 aspects, (i) and (iii) are modified by replacing the ROM content of the bitstream in-place using tools such as `data2mem`. On the other hand (ii) and (iv) are controlled by software during run-time. As a result, all 4 types of customization can be performed rapidly within seconds. They allow the same overlay implementation to be targeted to different applications as well as to different loop iterations of the compute kernel easily.

A second level of customization can be applied to the overlay implementation itself. As a *soft* overlay, many aspects of the array can be customized according to the input application to achieve different tradeoffs in area, energy and performance. Customizations may involve, the size of the array, the type of supported operation in each PE, the size of data and instruction memory, and even the pipeline depth of the network and the PEs. When compared to the first level of customization, this level of customization involves reimplementation of the overlay and requires considerably longer run time. User may therefore opt for this level of customization only as needed. Note that in all cases, the overlay remains synchronous and deterministic to keep the overall flow of QuickDough intact.

## 3.2 Processing Element (PE)

The key design element of the CGRA overlay is its processing element (PE). On one hand, the design of the PE must be simple with low overhead to reduce area and energy consumption, and to improve performance. On the other hand, the design of

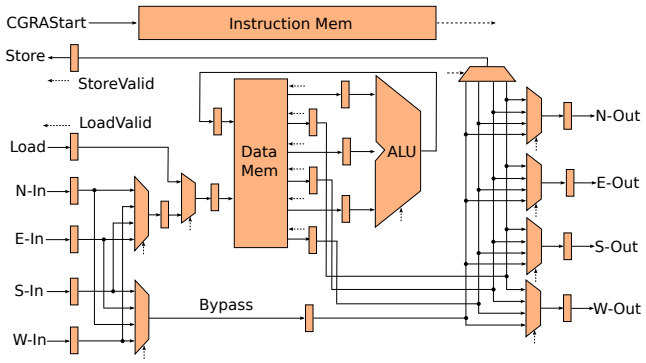



Figure 3.2: Fully pipelined PE structure. Each PE can be connected to at most 4 neighbours.

PE must also be flexible enough that it can support all the required operations in the target application.

Figure 3.2 shows the current implementation of a PE that features an optional load/store path. At the heart of the PE is an ALU, which is supported by a multi-port data memory and an instruction memory. Three of the data memory's read ports are connected to the ALU as inputs, while the remaining ports are sent to the output multiplexors for connection to neighboring PEs and the optional store path to OBuf external to the PE. At the same time, this data memory takes input from the ALU output, data arriving from neighboring PEs, as well as from the optional IBuf loading path. The action of the PE is controlled by the AddrCtrl unit that reads from the instruction memory. Finally, a global signal from the AccCtrl block controls the start/stop of all PEs in the array.

### 3.2.1 Instruction Memory and Data Memory

The instruction memory stores all the control words of the PE. As its content does not change at run-time, a ROM is used to implement this instruction memory. As shown in Figure 3.3, the address of the instruction memory is generated using a counter controlled by the `CGRAStart` from CGRACtrl block. Once the `CGRAStart` signal is valid, the instruction memory address will increase by one every cycle and the SCGRA execution will proceed accordingly. When the `CGRAStart` signal is invalid, the address will be reset to be 0 and the SCGRA execution will stop. In addition, pipeline registers are added to the output port of the instruction

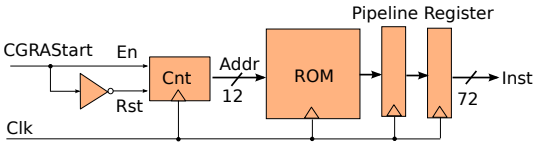



Figure 3.3: Instruction Memory

memory to ensure high implementation frequency.

Data memory stores intermediate data that can either be forwarded to the neighboring PEs or be sent to the ALU for calculation. To support non-blocking operations in the PE, at least 4 read and 2 write ports are needed. In each cycle, 3 reads are needed for the ALU and 1 read is needed for data forwarding. At a single cycle, one write port is needed to store input data from neighboring PEs and another one is needed to store the computing result of the ALU within the same cycle. The structure of the data memory is shown in Figure 3.4. In order to achieve high implementation frequency and meet various data memory capacity requirements, instead of using highly multiplexed flip flops, 3 primitive true dual port memory blocks that contain replicated data are employed to implement this data memory. The three BRAMs share 2 R/W selection lines for all the ports. It has two write ports and six read ports literally, but one write port and corresponding set of three reading ports are not allowed to act at the same cycle. For instance, when write port0 is selected, read port0, read port2 and read port4 are inactive. Compared to a standard 2-write-6-read multi-ported memory which allows 8 parallel read/write operations [1], it makes best use of the primitive block RAMs on FPGAs saving the resource consumption and achieving better timing.

### 3.2.2 ALU

At the heart of the proposed PE is the ALU that carries out the computations of the given application. As an overlay, the SCGRA overlay ALU must be simple, regular, and flexible such that it may easily be customized with different operations specifically for any given user application. In addition, it must also be fully pipelined in order to achieve high clock frequency and thus higher overall performance. Fig-

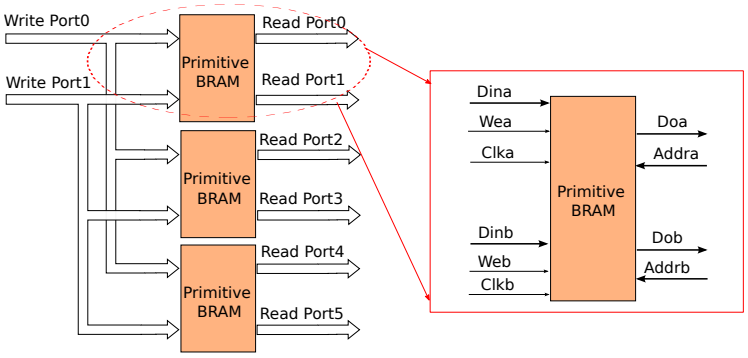



Figure 3.4: Multi-ported Data Memory, It makes best use of the primitive block RAM on FPGAs, though the read ports and write ports in the same group (e.g. Doa, Addra and Dina, Wea, Clka belong to the same group) are not allowed to act in parallel.

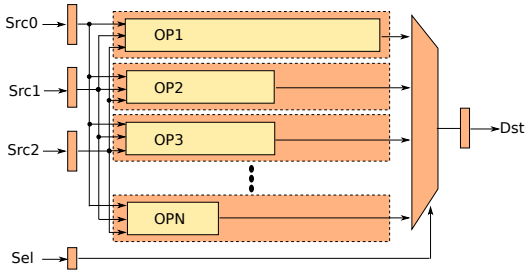



Figure 3.5: ALU of the SCGRA Overlay. It supports up to 16 fully pipelined 3-input operations.

ure 3.5 shows the current design of the ALU used in the SCGRA overlay.

The ALU supports up to 16 fully pipelined 3-input operations. Depending on the area-performance requirements, the ALU may be customized with operations specifically designed for the application. It may also be customized to support the common set of operations for multiple compute kernels. For example, Table 3.1 shows the set of operations we have developed for all the benchmarks in the experiment. Figure 3.5 shows the current set of operation. These operators in the ALU may execute concurrently in a pipelined fashion and must complete in a deterministic number of cycle. Given the deterministic nature of the operators, the SCGRA scheduler will ensure that there is never conflict at the output multiplexor.

### 3.2.3 Load/Store Interface

For the PEs that also serve as IO interface to the SCGRA, they have an additional load path and a store path as shown in 3.2. The data loading path and the SCGRA

Table 3.1: Operation Set Implemented in ALU. It covers all the four applications used in the experiments.

| Type | Opcode | Expression |
|---|---|---|
| MULADD | 0001 | Dst = (Src0 $\times$ Src1) + Src2 |
| MULSUB | 0010 | Dst = (Src0 $\times$ Src1) - Src2 |
| ADDADD | 0011 | Dst = (Src0 + Src1) + Src2 |
| ADDSUB | 0100 | Dst = (Src0 + Src1) - Src2 |
| SUBSUB | 0101 | Dst = (Src0 - Src1) - Src2 |
| PHI | 0110 | Dst = Src0 ? Src1 : Src2 |
| RSFAND | 0111 | Dst = (Src0 $\gg$ Src1) & Src2 |
| LSFADD | 1000 | Dst = (Src0 $\ll$ Src1) + Src2 |
| ABS | 1001 | Dst = abs(Src0) |
| GT | 1010 | Dst = (Src0 $>$ Src1) ? 1 : 0 |
| LET | 1011 | Dst = (Src0 $\leq$ Src1) ? 1 : 0 |
| ANDAND | 1100 | Dst = (Src0 & Src1) & Src2 |

neighboring input share a single data memory write port, and an additional pipeline stage is added to keep the balance of the pipeline. Similarly, the data storing path has an additional data multiplexers as well, but it doesn't influence the pipeline of the design.

To improve the performance of the resulting SCGRA overlay based accelerators, the load/store address calculation of the compute kernels is done in the QuickDough compiler. The addresses are generated at compilation time and then stored in the AddrIBuf and AddrOBuf. Consequently, there is only pure data in the load/store path while 1 bit `LoadValid` signal and 1 bit `StoreValid` signal are used to obtain the addresses from the address buffers i.e. AddrIBuf and AddrOBuf.

## 3.3 Accelerator Buffers

Input/output buffer is used to store data transmitted between the FPGA accelerator and main memory. Since the two ports of the input/output buffers are either specialized as input or output (For instance, the input buffer has one write only port connected to the system interconnection while they have the other read only port connected to the SCGRA computing logic.), simple dual port block RAMs can be used to implement the input/output buffers.

The input/output address buffers store the addresses of load/store operations.

As these addresses are decided at compilation time, the input/output buffers are implemented as ROM. While the addresses of load/store operations are stored in the same sequence as the load/store sequence. In order to return the required addresses upon to the `LoadValid`/ `StoreValid` signal, the ROM relies on a counter to generate the ROM reading addresses, which is similar to the instruction memory of each PE.

## 3.4 Accelerator Controller

AccCtrl starts the lock-step computing of the whole SCGRA array when it receives the `start` signal from the host processor. At the end of the computing, `done` signal will be sent to the host processor to collect the computing result. Since the regular SCGRA overlay typically runs at higher frequency than the interface connected to the system interconnection, they are allocated in two separate clock domains, and the `start` and `done` transmit across the two clock domains through three-level registers.

On top of the basic features, the accelerator allows the CGRA computing logic to be repeated on data transmitted through the input/output buffers, which helps to amortize the initial data transfer cost. To support this unique feature, CGRACtrl has two registers `ItNum` and `ItLen` to represent the number of the repeated CGRA computing and the length of each compute iteration. Based on the two registers, CGRACtrl produces the `CGRAStart` signal which controls the action of the instruction memory and the `Done` signal that indicates the end of the hardware execution.

## 3.5 Experiments

SCGRA overlay is the backbone of the accelerator and its implementation is critical to the resulting FPGA acceleration system. Thus a series of experiments are conducted to evaluate the SCGRA overlay implementation in terms of pipelining

Table 3.2: Pipeline Configurations

| Pipeline Options | `Input` → `Output` | `Input` → `Bypass` → `Output` | `Input` → `Write Back` |
|---|---|---|---|
| 100MHz | 2 cycles | 1 cycle | 4˜6 cycles |
| 150MHz | 2 cycles | 1 cycle | 5˜8 cycles |
| 200MHz | 4 cycles | 2 cycles | 7˜11 cycles |
| 250MHz | 7 cycles | 3 cycles | 11˜17 cycles |

Table 3.3: SCGRA Configuration

| SCGRA Topology | Instruction Memory | Data Memory | I/O Data Buffer | I/O Address Buffer |
|---|---|---|---|---|
| Torus | 1K × 72 bits | 256 × 32 bits | 2K × 32 bits | 4K × 18 bits |

and scalability which further exhibits the performance, power consumption and energy efficiency of the SCGRA overlay. This section is organized as follows. Basic experiment setup is presented in next subsection. Then the overlay pipelining and scalability are analyzed respectively.

### 3.5.1 Experiment Setup

Both $2 \times 2$ and $5 \times 5$ SCGRA overlays were developed in ISE 14.7 and Vivado 2013.4 targeting XC7Z020 FPGA which is the FPGA on Zedboard. The overlays support all the operations as mentioned in previous section.

In order to explore the SCGRA overlay pipelining, SCGRA overlays with different pipelining have been developed. As shown in Table 3.2, the SCGRA overlay implementations with different pipelining configurations can typically run at 100MHz, 150MHz, 200MHz and 250MHz respectively. Note that `Input` → `Output` in Table 3.2 represents the latency of a normal data transmitting from input port of a PE to an output port of the PE. `Input` → `Bypass` → `Output` stands for the bypass data path length of a PE. `Input` → `Write Back` means data transmitting latency from input port of a PE to the write port of data memory in the PE. As it includes the latency of data paths of different operations in ALU, it varies while it remains deterministic for each specific operation. The rest part of the basic configurations of the overlay are shown in Table 3.3.

In order to estimate the performance of the overlays with different pipelining,

Table 3.4: Detailed Configurations of the Benchmark

| Benchmark | MM | FIR | SE | KM |
|---|---|---|---|---|
| Configurations | Matrix Size | # of Input/ # of Taps+1 | # of Vertical Pixels/ # of Horizontal Pixels | # of Nodes/Centroids/ Dimension |
| C1 | 10 | 40/50 | 8/8 | 20/4/2 |
| C2 | 100 | 10000/50 | 128/128 | 5000/4/2 |
| C3 | 1000 | 100000/50 | 1024/1024 | 50000/4/2 |

Table 3.5: DFG Information (# of Input/ # of Output/ # of Operations)

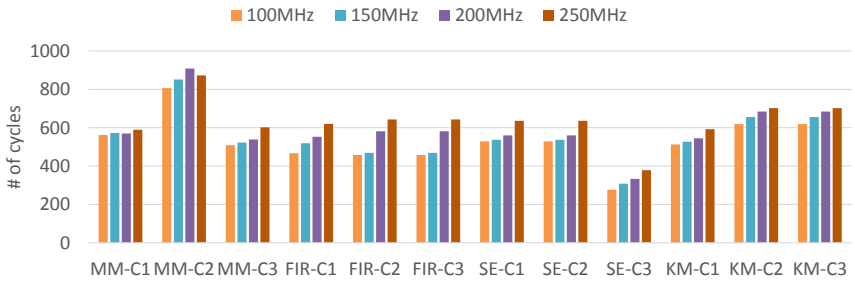
| Configurations | MM | FIR | SE | KM |
|---|---|---|---|---|
| C1 | 200/100/1000 | 70/20/860 | 31/8/1080 | 49/12/920 |
| C2 | 600/5/750 | 120/20/1000 | 31/8/1080 | 59/12/1144 |
| C3 | 400/1/301 | 120/20/1000 | 27/4/540 | 59/12/1144 |

a group of DFGs generated from matrix multiplication (MM), finite impulse filter (FIR), Sobel edge detector (SE), K-means (KM) with different configurations are mapped to the overlay as the benchmark. The configurations of the compute kernels are detailed in Table 4.1 and the extracted DFGs are presented in Table 3.5.

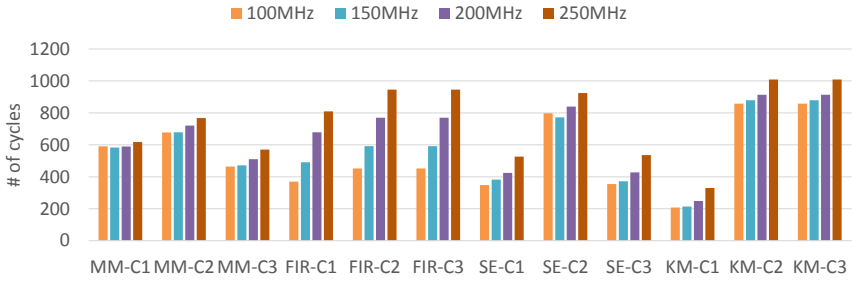
### 3.5.2 Pipelining

Pipelining influences both the latency of the SCGRA overlay data paths and the implementation frequency of the resulting accelerators, which further affects the performance, resource consumption and the energy efficiency of the FPGA acceleration system. To explore the SCGRA overlay pipelining, this subsection focuses on how different SCGRA overlay pipelining affects the performance, resource consumption and energy efficiency of the resulting hardware accelerators.

By using the benchmark Table 4.1 and Table 3.5, Figure 3.6 shows the relation between the SCGRA overlay pipelining and the number of DFG execution on both a $2 \times 2$ SCGRA overlay and a $5 \times 5$ SCGRA overlay. It is clear that SCGRA overlays with less pipeline stages and lower implementation frequency typically achieves shorter execution cycles due to the shorter data path as detailed in Table 3.2. However, when taking the clock frequency in to consideration, SCGRA overlays with deeper pipelining and higher implementation frequency outperform eventually on the overall run-time as presented in Figure 3.7. There are exceptions for example FIR C1 mapped to the $5 \times 5$ SCGRA overlay exhibits slightly




(a) SCGRA2 $\times$ 2

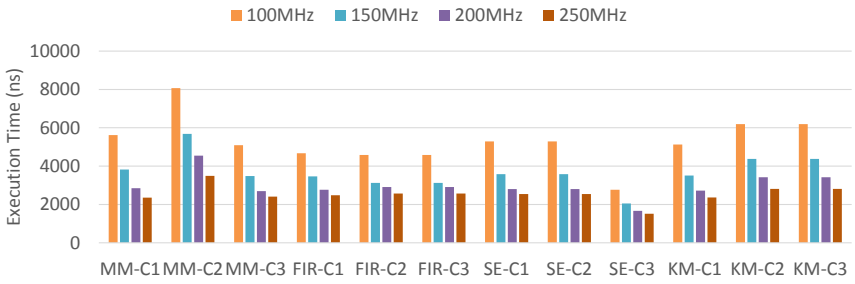



(b) SCGRA 5 $\times$ 5

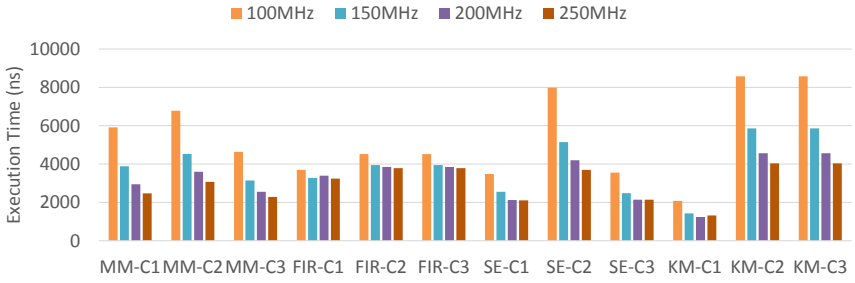
Figure 3.6: The Number of Cycles of DFG Execution on SCGRA Overlays with Different Pipelining

differently. It is mainly caused by the operation scheduling which is essentially an NP-complete problem. When mapping a relatively small DFGs to a larger SCGRA, it will be challenging for the scheduler to compromise between the load balance and the communication cost and scheduling performance may fluctuate. The detailed operation scheduling will be further discussed in next chapter. In this case, using a smaller SCGRA overlay instead of a larger one usually solves this problem. In summary, fully pipelined SCGRA overlay will be used in QuickDough aiming to provide high-performance FPGA accelerators.

The SCGRA overlay pipelining also had direct influence on the FPGA resource consumption. Figure 3.8 shows the hardware resource utilization of SCGRA overlay with different pipelining. It can be found that the overlays with deeper pipelining and higher implementation frequency consumes more Flip Flops (FFs) and slightly more LUTs while the RAM blocks and DSP slices remain the same. While the SCGRA overlays have rather low FFs and LUTs utilization, therefore deeper pipelining and higher implementation frequency typically will not bring significant resource constrain to the FPGA accelerator design.




(a) SCGRA $2 \times 2$




(b) SCGRA $5 \times 5$

Figure 3.7: Overall DFG Execution Time on SCGRA Overlays with Different Pipelining

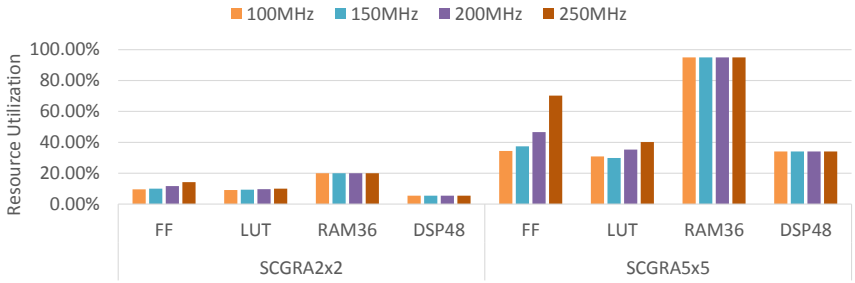



Figure 3.8: Hardware Resource Utilization of SCGRA Overlays with Different Pipelining

Finally, the relation between the SCGRA overlay pipelining and energy efficiency of the resulting accelerators are also analyzed. The SCGRA overlay power consumption which were obtained through XPower is given in Figure 3.9. As expected, SCGRA overlay with deeper pipelining and higher implementation frequency consumes larger power. With the estimated SCGRA overlay power consumption, energy efficiency represented by energy delay product (EDP) can be calculated. As shown in Figure 3.10, although there are exceptions especially the FIR mapped to $5 \times 5$ SCGRA overlay, SCGRA overlays with deeper pipelining and higher implementation frequency usually turns out to be more energy efficient. As mentioned in previous section, the exception mainly happens when the target applications don't have sufficient parallel operations for the scheduler to make best use

of the pipeline or processing elements. Typically a smaller SCGRA overlay should be used instead and it may achieve higher performance and higher energy efficiency as well.

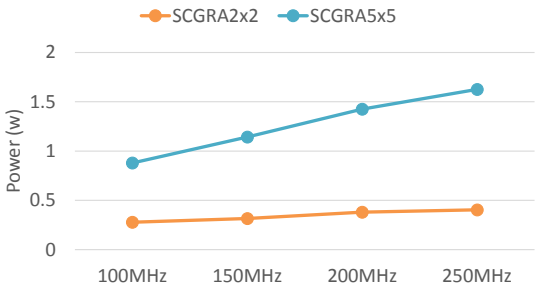
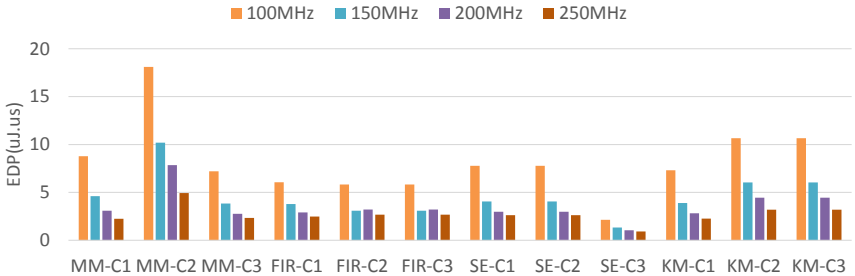
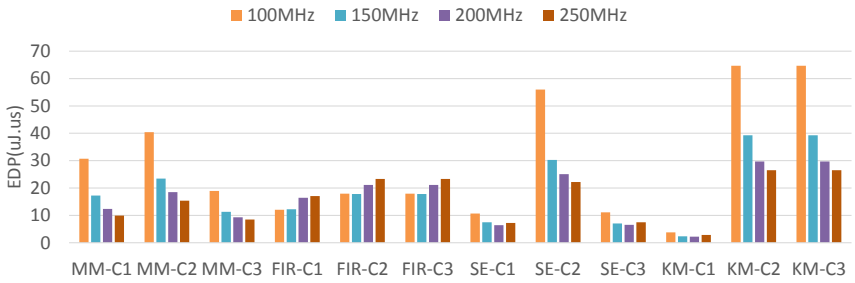



Figure 3.9: Power Consumption of SCGRA Overlays with Different Pipelining




(a) SCGRA2 $\times$ 2



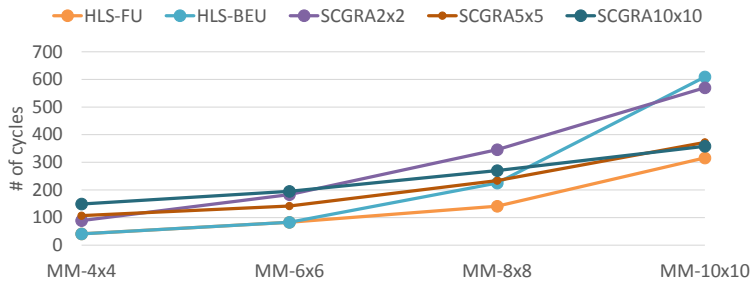

(b) SCGRA 5 $\times$ 5

Figure 3.10: Energy Delay Product of SCGRA Overlays with Different Pipelining

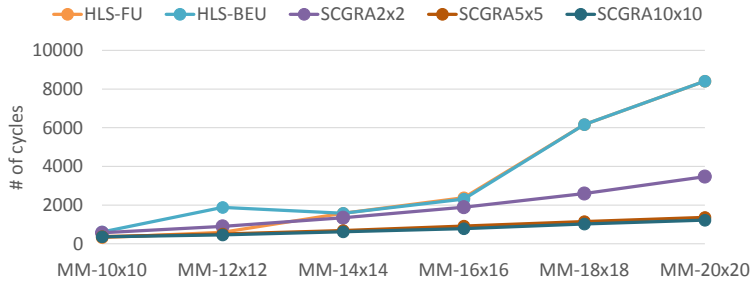
According to the experiments in this section, highly pipelined SCGRA overlays with high implementation frequency exhibit competitive performance and energy efficiency consuming slightly more FPGA resources. Consequently, highly pipelined SCGRA overlays are used in the rest of this work.

### 3.5.3 Scalability

In this subsection, we studied the scalability of the proposed overlay architecture in terms of problem size as well as processing array size in anticipation for future




(a)




(b)

Figure 3.11: # of cycles of MM Execution on Both HLS Based Design and SCGRA Overlay

FPGA devices.

To study the effect of problem size on the overlay performance, MM with matrix sizes ranging from $4 \times 4$ to $20 \times 20$ were used. In the case of SCGRA overlay, 3 different SCGRA sizes were studied: $2 \times 2$, $5 \times 5$, $10 \times 10$. They were targeted at the larger `zc706` FPGA with abundant hardware resource. In the case of direct HLS synthesis, Vivado HLS 2014.3 was used. 2 scenarios were considered. In the first scenario, the matrix multiplication was fully unrolled regardless of the resource consumption, which leads to the best achievable performance using HLS. The resulting design is labeled as HLS-FU. In the second scenario, a best-effort loop unrolling that results in maximum performance under hardware constraints was used. The design is labeled as HLS-BEU. In the case of the SCGRA overlay, the MM operations were fully unrolled. Figure 3.11 shows the number of cycles of MM execution on the resulting accelerators generated by both design methods. Communication cost was not taken into account as they remain comparable in both cases.

Results from the figure show that accelerators using direct HLS based design framework perform much better than the SCGRA overlay when the matrix size is

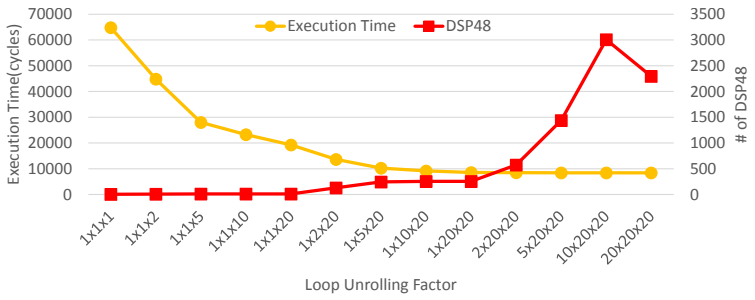
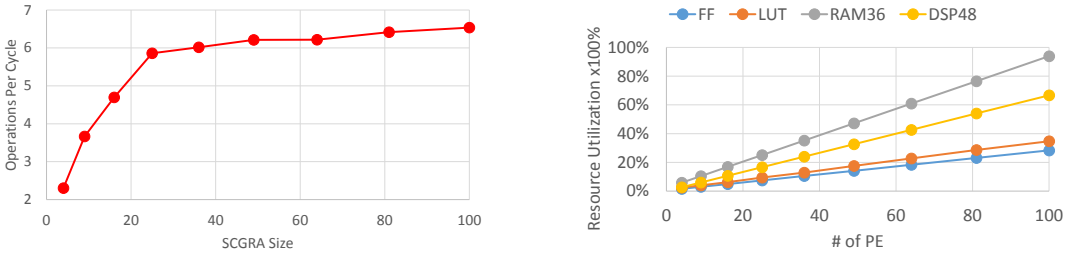



Figure 3.12: MM 20x20 Implemented Using Direct HLS with Various Loop Unrolling

small enough for the loops to be fully unrolled. However, as the matrix size grows, direct HLS can no longer afford the hardware overhead for intensive loop unrolling, limiting the performance of the resulting accelerators. As shown in Figure 3.12, the DSP resource consumption increases significantly with additional unrolling in return for higher performance as the matrix size increases. While it is possible for an expert to start manually time-sharing the resources, we did not explore such sophisticated implementation technique in attempt to maintain a comparable design effort.

On the other hand, the SCGRA overlay can naturally accommodate intensive loop unrolling through time-sharing of hardware resources. It is therefore capable of accelerating much larger portion of computation with relative ease. Its performance is mainly limited by the on-chip memory serving as instruction ROMs. In our experiments with MM that target the larger `zc706` device, MM-8x8 is the crossover point that SCGRA overlay begins to outperform. As a comparison, the crossover point on the originally targeted Zedboard was much smaller because of the limited hardware resource.

To further study the scalability of the proposed SCGRA overlay, the performance of processing array with increasing size was experimented using MM-20x20 as an example. Figure 3.13(a) shows that in general, the additional processing power provided by the larger arrays results in better performance as expected. The performance gain reduces as the array size increases to a point when there simply is not enough available compute operations to be scheduled. This point of reflection depends on the nature of the user application. In the case of MM, the $5 \times 5$ array




(a) Performance (b) Resource Consumption

Figure 3.13: Performance, and Resource Consumption of MM-20 implemented with increasing SCGRA overlay size.

Table 3.6: SCGRA Based FPGA Accelerator Configuration

| SCGRA Size | Inst. Rom | Data Mem | IBuf /OBuf | Addr Buf |
|---|---|---|---|---|
| 2x2, 3x3, ..., 10x10 | 1k, 4k, ..., 8k | 256x32 | 2kx32, 4kx32, ..., 8kx32 | 4kx16, 8kx16, ..., 16kx16 |

provided the near-optimal performance with reasonable amount of PEs.

As an overlay design, it is important that its design should allow flexible scaling of its hardware resource consumption to satisfy the various resource constraints from the users. As a simple, fully synchronous and highly regular reconfigurable array, the SCGRA overlay is very much scalable in that dimension as shown in Figure 3.13(b). From the figure, it can be seen that the hardware resource consumption associated with the SCGRA overlay increases linearly with the number of processing elements in the array.

Finally, as discussed in previous section, implementation frequency has significant influence on the performance, power consumption and energy efficiency of the

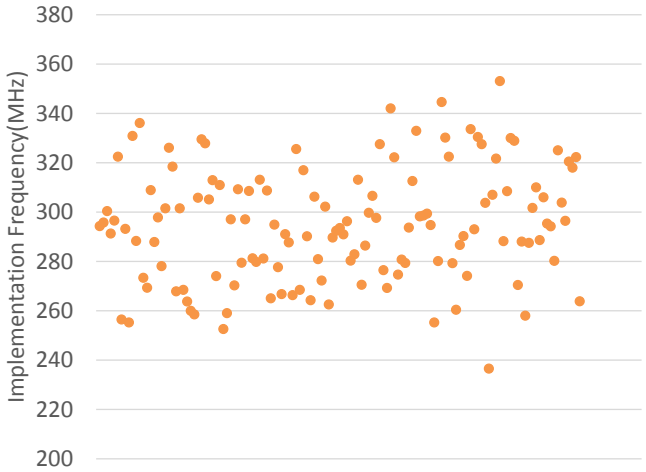



Figure 3.14: fmax of SCGRA Overlay with Various Configurations

resulting accelerators. Therefore, it is important to maintain high implementation frequency when the configurations of the overlay changes. To explore the scalability of the SCGRA overlay, a group of SCGRA overlays with various configurations as shown in Table 3.6 were implemented on Zedboard. Figure 3.14 shows the fmax of these SCGRA overlays. It can be found that the implementation frequency remains around 300MHz and stays relatively stable for various configurations.

## 3.6 Summary

In this work, a template of SCGRA overlay based FPGA accelerator is developed. It is highly pipelined and exhibits higher performance and energy efficiency compared to the one with less pipeline stages. Meanwhile it is simple and easy to be extended for various applications. In addition, the regular SCGRA overlay is scalable in terms of problem size, resource consumption, implementation frequency. Particularly, the implementation frequency of the SCGRA overlay is competitive and relatively stable with various configurations.

# Chapter 4

# QuickDough Design Framework

QuickDough is a development framework for FPGA-accelerated applications. It generates FPGA accelerators for compute intensive loop kernels rapidly through the use of a pre-built soft coarse-grained reconfigurable array (SCGRA) overlay. QuickDough also performs application-specific customization and generates optimized SCGRA overlay as well as communication infrastructure between the CPU host and the accelerator automatically, integrating both software and hardware generation in a unified framework.

The overall design goal of QuickDough is to enhance the designer's productivity by greatly reducing the hardware generation time and by providing automatic optimization of the data I/O between the host software and the accelerator. Instead of spending hours on conventional hardware implementation tools, QuickDough is capable of producing the targeted hardware-software system in the order of seconds. By doing so, it provides a rapid development experience that is compatible with that expected by most software programmers.

To achieve this compilation speed, while maintaining a reasonable accelerator performance, QuickDough avoids the creation of custom hardware directly for each application. Instead, the compute kernel loop bodies are scheduled to execute on a CGRA overlay, which is selected from a pre-implemented CGRA based accelerator library. By sidestepping the time-consuming low-level hardware implementation tool flow, the time to implementing an accelerator in QuickDough is reduced to

essentially just the time spent on overlay selection and scheduling compute operations on the resulting overlay. In addition, taking advantage of the overlay's softness and regularity, QuickDough allows users to perform trade-off between compilation time and performance by selecting and customizing the overlay on a per application basis. The result is a unified design framework that seamlessly produces the entire hardware-software infrastructure with a design experience similar to developing conventional software.

This chapter mainly focuses on the the basic compilation flow from high-level loop kernels to the SCGRA overlay based FPGA accelerator bitstream and it will be organized as follows. The overall QuickDough design framework will be presented in the next section. The processes in the rapid QuickDough compilation flow will be detailed in Section 4.2. Finally, the accelerator library used in the rapid compilation flow as well as its pre-building process will be explained in Section 4.3.

## 4.1 QuickDough Overview

Figure 4.1 summarizes the hardware-software compilation flow of QuickDough. Users begin by specifying the regions for accelerations in the form of compute intensive loops. Once a loop is identified, it is further compiled to an SCGRA overlay based FPGA accelerator through SCGRA customization and SCGRA compilation while the rest part of the program is compiled to the processor through a conventional software compilation.

QuickDough partitions the complex SCGRA overlay based accelerator development flow into two paths. Along the rapid and common acceleration generation path, QuickDough first translates the compute-intensive loop kernels into data flow graphs (DFGs) and then statically schedules the DFGs on to the SCGRA overlay. Afterwards, it integrates the scheduling result with a partially implemented overlay bitstream which is selected from a pre-built overlay based accelerator bitstream library. By employing different selection algorithms on the accelerator library, QuickDough allows users to perform trade-off between performance and compila-

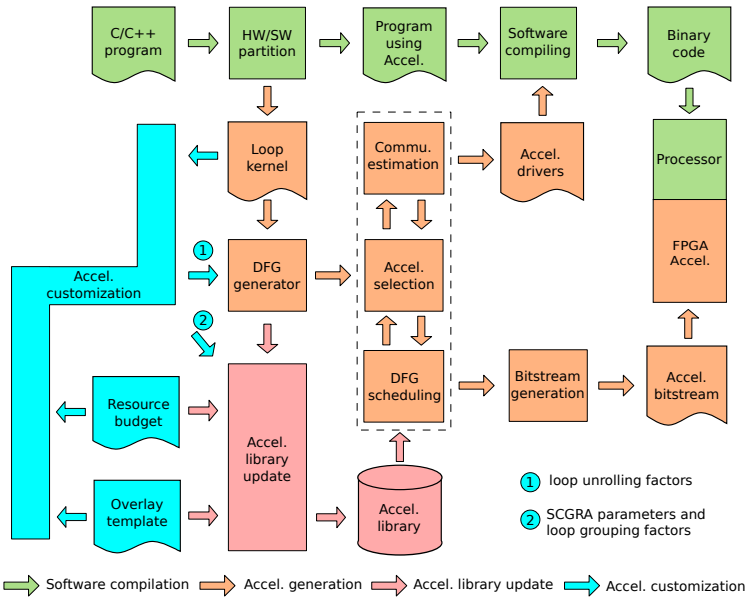



Figure 4.1: QuickDough: FPGA Loop Accelerator Design Framework Using SCGRA Overlay.

tion time. At the end of the selection process, optimized communication interfaces will be generated accordingly.

QuickDough also includes a relatively slow yet less frequent accelerator library update path which pre-build the SCGRA overlay based accelerator library. To expedite the library generation process, a small representative set of accelerator configurations are chosen as the library and generated automatically using an SCGRA overlay template based system. Although it does take some time to build the accelerator library, the library can be reused during design iterations of an application or shared by a domain of applications and thus the library building time can be amortized. In addition, the SCGRA overlay template is highly pipelined to allow high-frequency and high-performance overlay based accelerator implementation. The SCGRA overlay is also regular and scalable, which makes the resulting accelerators conveient to be adapted to different applications.

In addition, QuickDough supports automatic application-specific customization all the way from high level loop unrolling, on-chip buffer sizing and overlay structure configuration to achieve optimized performance and energy efficiency of the resulting accelerator. The automatic customization process helps the users to explore the large and complex design space and it also makes QuickDough accessible

to software designers. The customization process introduces moderate time consumption, but it is optional and the customized accelerator configuration can also be added to the accelerator library for reuse in future. Detailed customization process will be presented in next chapter.

The slow path of QuickDough is to update the accelerator library upon users' request and users may simply provide the hardware resource budget. Then target operations to be supported will be decided automatically by analyzing the DFGs produced by the DFG generator. With the resource budget and the supported operation set, a set of representative accelerator HDL models will be generated by utilizing the overlay based accelerator template. Finally, the accelerator HDL models are implemented on the target FPGA platform and further added to the accelerator library.

## 4.2 Rapid Accelerator Generation

The rapid acceleration generation path of QuickDough consists of a series of interrelated steps as illustrated in Figure 4.1. In the first step, the compute intensive loop kernel is statically transformed to the corresponding data flow graph (DFG) with specified loop unrolling factor which can either be user input or be obtained from the customization process. Then the accelerator selection process selects an accelerator from a pre-built accelerator library based on the scheduling performance and the communication cost. The scheduling performance is obtained from the SCGRA scheduling process which schedules the generated DFG to the SCGRA overlay included in the selected accelerator while the communication cost is obtained through a CPU-FPGA communication estimation model. After the accelerator selection process, the accelerator drivers can be generated accordingly based on the on-chip buffer size of the selected accelerator. Meanwhile, the selected empty pre-built accelerator bitstream and the corresponding scheduling result are integrated to create the final FPGA accelerator configuration bitstream. This bitstream, in combination with the binary code created in the software compilation process, forms the final

application that will be executed on the target CPU-FPGA system.

### 4.2.1 DFG Generation and Loop Execution

The main input to the QuickDough framework for acceleration are compute intensive kernels extracted from the user applications. The first step of the compilation process is therefore to extract data flow graphs (DFGs) from these kernels that are often expressed as inner loop bodies. In order to express the loop bodies as DFGs, data dependency in the loop body must be known at compilation time. Similar to LLVM intermediate representation [72], branches in the loop bodies can be removed by using PHI instructions which can be further mapped to the PHI operation as described in previous chapter. While this strategy helps to extend the reach of the SCGRA overlay based loop acceleration, it remains challenging to convert large and complex branches. In this work, a C++ library is built to help automate the DFG generation with specified operation set and loop unrolling.

The loop kernels are partially unrolled, transformed to DFGs and scheduled to the SCGRA overlays of the accelerators. A straightforward way to perform the whole loop computation on the overlay is to repeat the same DFG computation until the end of the loop. Nevertheless, this may require data transfer between host processor and I/O buffer for each DFG computation. As a result, the communication cost increases dramatically especially when the amount of each data transfer is small. Worse still, input data of the consecutive DFGs may be reused and the straightforward data transfer strategy may greatly increase the total amount of data transfer through out the loop computation.

To alleviate this problem, we have proposed to batch data transfers for multiple executions of the same DFG into groups as shown in Figure 4.2. Specifically, after the loop is unrolled $U$ times, $G$ of them are grouped together for each data transfer. This group strategy helps to amortize the initial communication cost between host processor and the accelerator. In addition, it allows input data to be reused for different DFG computation in the same group and the group size is mainly lim-

ited by the I/O buffer depth. Meanwhile, the accelerator communicates with host processor for each group execution, and thus the accelerator driver that handles the communication depends on the I/O buffer depth as well.

While grouping data transfers helps amortize the communication cost between CPU and the accelerator, it also imposes additional requirement for on-chip memory to serve as buffer for the extra data transferred. As a result, the unrolling factor $U$ and grouping factor $G$ has to be co-optimized to balance performance and on-chip resource utilization. For instance, increasing $U$ leads to a larger DFG to be executed in the QuickDough overlay, which may be benefited from a larger processing array. However, the increased processing array limits the amount of on-chip buffer available for data and address buffer. As a result, the amount of DFG grouping $G$ is limited and may lead to higher performance penalty due to communication. In addition, the proposed accelerator employs address buffers to store all the on chip buffer accessing addresses of the whole group execution. While the addresses of different DFG execution in a group can't be reused, the capacity of the address buffers is the product of the number of input/output addresses of a DFG and the number of DFGs in a group, which is usually larger than the capacity of the input/output buffers especially when there are a large amount of input/output data reuse between different DFGs in the same group.

### 4.2.2 DFG Scheduling

The operations from the user DFG are scheduled to execute on the reconfigurable array. Since the processing elements in the QuickDough overlay execute in lock steps with deterministic latencies, a classical list scheduling algorithm [92] was adopted. The challenge in this scheduler is that data communication among the processing elements must be carried out via multi-hop routing in the array. As a result, while it is desirable to schedule data producers and consumers in nearby processing elements to minimize communication latencies, it is also necessary to utilize as much parallel processing power as possible for sake of load balancing. Building on top

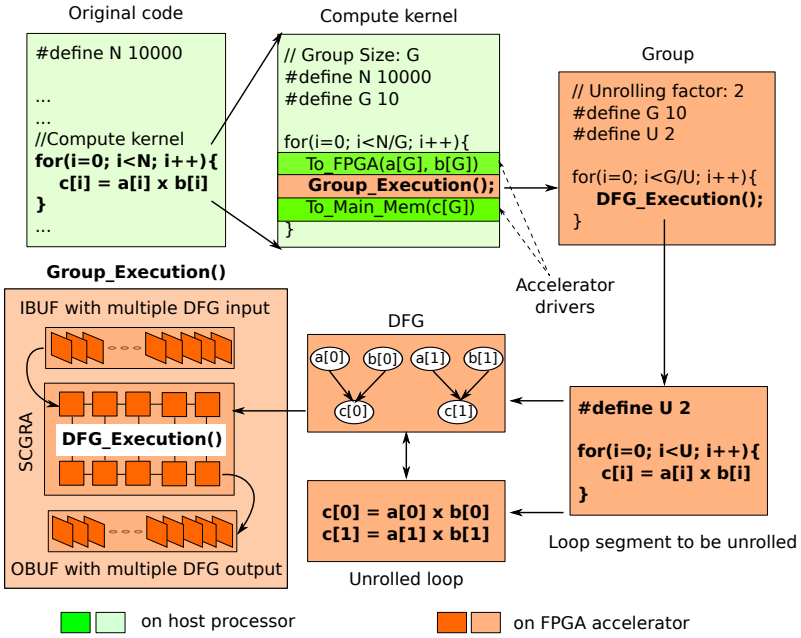



Figure 4.2: Loop execution on an SCGRA overlay based FPGA accelerator

of our previous work presented in [68], a scheduling metric considering both load balancing and communication cost was adopted in our current implementation.

**Algorithm 1** The QuickDough scheduling algorithm.

```
procedure ListScheduling
    Initialize the operation ready list L
    while L is not empty do
        select a PE p with lowest utilization
        select an operation l that completes computing earliest
        OPScheduling(p, l)
        Update L
    end while
end procedure

procedure OPScheduling(p,l)
    for all predecessor operations s of l do
        Find nearest PE q that has a copy of operation s
        Find shortest routing path from PE q to PE p
        Move operation s from PE q to PE p along the path
    end for
    Do operation l on PE p
end procedure
```

Algorithm 1 briefly illustrates the scheduling algorithm implemented in QuickDough. Initially, an operation ready list is created to represent all operations that are ready to be scheduled. The next step is to select a PE from the SCGRA and an operation from the ready list using a combined communication and load balance

metric. Basically the PE with lowest utilization is choosen as the target processing element for the next operation. Then estimate the execution time of all the ready operations on the selected PE. The operation that completes computing earliest will be selected and scheduled. When both the PE and the operation to be scheduled are determined, the `OPScheduling` procedure starts. It determines an optimized routing path, moves the source operands to the selected PE along the path, and schedules the selected operation to execute accordingly. After this step, the ready list is updated as the latest scheduling may produce more ready operations. This `OPScheduling` procedure is repeated until the ready list is empty. Finally, given the operation schedule, the corresponding control words for each PE and the IO buffer accessing sequence will be produced. These control words will subsequently be used for bitstream generation in the following compilation step.

### 4.2.3 Accelerator Selection

Accelerator selection process selects an accelerator from the accelerator library based on the performance of the resulting accelerator which mainly depends on the computation latency and communication latency. The computation latency of the loop kernel can be calculated using (4.1). $DFG\_Lat$ stands for the number of cycles needed to complete the SCGRA scheduling and mostly depends on the SCGRA overlay size while $Freq$ stands for the pre-built accelerator implementation frequency. The communication latency can be calculated using (4.2) where $Trans()$ represents the data transfer latency function of the target platform and $GpIn$ and $GpOut$ represent the amount of data transfer of a group which is limited by the capacity of the I/O buffers.

$$CompLat = DFG\_per\_Loop \times DFG\_Lat / Freq \tag{4.1}$$

$$CommLat = Gp\_per\_Loop \times (Trans(GpIn) + Trans(GpOut)) \tag{4.2}$$

In summary, the performance of the accelerator can be estimated with analytical

models when the scheduling performance is obtained through the DFG scheduling while the scheduling performance is mostly determined by the SCGRA overlay size. The analytical estimation is fast while the scheduling process is relatively slow. Therefore, the accelerator selection process essentially centers the SCGRA overlay size selection and then explores all the accelerator configurations (such as on chip buffer size, instruction memory depth and data memory size) with the same SCGRA overlay size.

To compromise the loop accelerator generation time and performance, three different levels of accelerator selection optimization levels are provided in this framework namely O0, O1 and O2 centering the SCGRA overlay size selection. O0 doesn't provide any optimization, and it selects an accelerator with the smallest SCGRA overlay. O1 estimates three typical accelerators with the smallest SCGRA overlay, a medium one and the largest SCGRA overlay. Then the one that provides the best performance will be adopted. O2 explores all the accelerators in the library and searches for the best accelerator configuration. With the increase of the optimization level, the accelerator selection process spends more effort in searching the accelerator library for better performance and thus results in longer compilation time.

### 4.2.4 Bitstream Integration

The final step of the compilation is to generate the instructions for each PE and the address sequences for the I/O buffers according to the scheduler's result, which will subsequently be incorporated into the configuration bitstream of the overlay produced from previous steps. By design, our overlay does not have any mechanism to load instruction streams from external memory. Instead, we take advantage of the reconfigurability of SRAM based FPGAs and store the cycle-by-cycle configuration words using on-chip ROMs. The content of the ROMs are embedded in the bitstream and the `data2mem` tool from Xilinx [106] is used to update the ROM content of the pre-built bitstream directly. To complete the bitstream integration,

BMM file that describes the organization and placements of the ROMs in the overlay is extracted from XDL file corresponding to the overlay implementation [10]. This bitstream integration process costs only a few seconds of the compilation time.

## 4.3 Accelerator Library Update

The accelerator library consists of a number of pre-built SCGRA overlay based accelerators with different configurations. It is the basis for the proposed rapid FPGA loop accelerator generation framework. In this section, we will illustrate how the accelerator library is updated given the hardware resource budget and target loop kernels.

An accelerator library update is essentially to pre-implement a group of SCGRA overlay based FPGA accelerators upon the users' request, which may either target a specified application or a domain of applications. As QuickDough aims to enhance the designers' productivity and make FPGA accelerator design accessible to high-level application designers, the library update which involves low-level circuit design and optimization is thus automated so that it will not become a new barrier to the application developer.

Figure 4.3 presents the proposed automatic accelerator library update flow. It roughly consists of four steps i.e. DFG generation, common operation analysis, minimum accelerator configuration set analysis, and accelerator HDL model generation and implementation. Since DFG generation has been discussed in previous section, we will mainly detail the remaining three steps in this section.

### 4.3.1 Common Operation Analysis

In this framework, the operations used to construct the DFG is up to the DFG generation process and the common operation analysis step mainly decides the minimum operation set that is needed to support the target applications. It is possible to co-optimize the DFG generation and common operation set analysis, but it is beyond

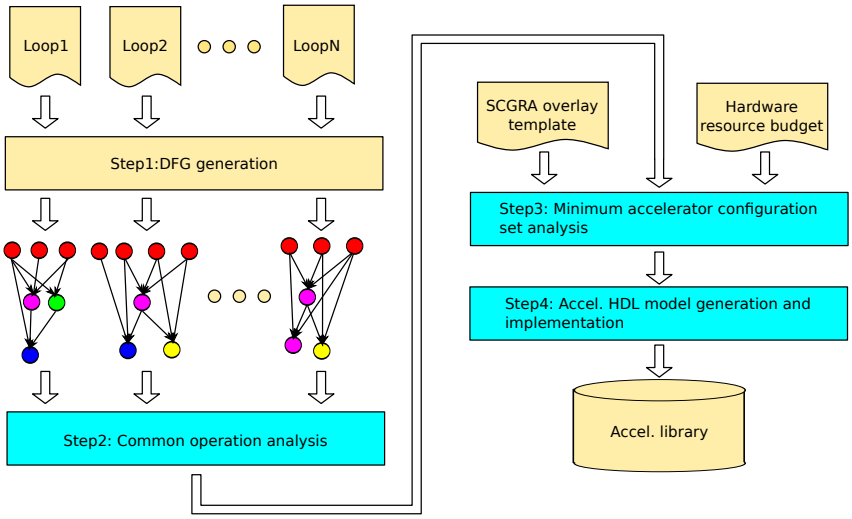



Figure 4.3: Automatic SCGRA overlay based FPGA accelerator library update

the discussion of this work. Currently, we just perform a union of the operation types included in the DFGs. It is trivial, but the minimum operation set can be decided automatically and rapidly.

### 4.3.2 Minimum Accelerator Configuration Set Analysis

Although the library can be implemented off-line, it does take a long time to complete. Therefore, we try to find out the minimum set of accelerator configurations that need to be pre-implemented as the library and maintain the application coverage of the library at the same time.

The proposed SCGRA overlay based FPGA accelerator utilizes block RAM to implement the instruction memory, data memory, on-chip buffer as well as the address buffer. As a result, the block RAM becomes the hardware resource bottleneck and the library basically depends on how the block RAM budget is allocated to different components of the accelerators. Therefore, the minimum library can be obtained using equation (4.3). *Row* and *Col* stand for the SCGRA overlay size and they are integers. *IM*, *DM*, *AIOB* and *DIOB* stand for the instruction memory capacity, data memory capacity, address IO buffer capacity and IO buffer capacity. In addition, they can only increase with the granularity of a primitive block RAM. *B* stands for the user specified block RAM budget.

$$Row \times Col \times (IM + DM) + AIOB + IOB \leq B \tag{4.3}$$

Moreover, empirical settings such as limiting data memory in each PE to a single primitive block RAM (i.e. $DM = 1$), constraining the difference between SCGRA row size and column size (i.e. $Col \leq Row \leq (Col + Gap)$, $Gap$ is an integer) and setting $AIOB = IOB$ are employed to further reduce the number of accelerators pre-built in the library.

### 4.3.3 Accelerator HDL Model Generation and Implementation

The QuickDough overlay is a highly regular processing array that can be generated easily according to the template as described in previous chapter. The overlay may be customized in many aspects, including the type of operation supported by each processing element (PE), the processing array size, its topology, as well as the number and capacity of the data buffers. With the proposed SCGRA overlay template and the accelerator configurations to be pre-built in the library, corresponding HDL models of the SCGRA overlay based FPGA accelerators are generated with a python script. Then the library can be implemented using the conventional hardware implementation tools. The lengthy implementations can be done in parallel. Moreover, the regular tiling structure even allows the implementations to be accelerated using macro based implementation techniques as presented in [115], which can be up to 20X faster than a standard HDL implementation with negligible timing and overhead penalty. After the implementation, implementation frequency is added to the corresponding accelerator configuration, which completes the whole library update process.

## 4.4 Experiments

With an objective to improve designers' productivity in developing FPGA accelerators, the key goal of QuickDough is to reduce FPGA loop accelerator development

time for a hybrid CPU-FPGA system. By using four typical loop kernels as the benchmark, we have evaluated the FPGA accelerator generation time with Quick-Dough. Meanwhile, to warrant the merit of such framework, the performance of the generated acceleration system should remain competitive. For that purpose, the performance is then compared against to that of software executed on an ARM processor. Finally, the pre-built accelerator library that affects both the design productivity and overhead of the resulting accelerators is also discussed.

### 4.4.1 Benchmark

Four applications were used as benchmark in this work, namely, a matrix-matrix multiplication (MM), a finite impulse response (FIR) filter, a K-mean clustering algorithm (KM) and a Sobel edge detector (SE). The basic parameters and configurations of the benchmark are illustrated in Table 4.1.

Table 4.1: Detailed Configurations of the Benchmark

| MM | FIR | SE | KM |
|---|---|---|---|
| Matrix Size | # of Input/<br># of Taps+1 | # of Vertical Pixels/<br># of Horizontal Pixels | # of Nodes/Centroids/<br>Dimension |
| 100 | 10000/50 | 128/128 | 5000/4/2 |

### 4.4.2 Experiment Setup

The Xilinx implementation tools were run on a computer with Intel Core i5-3230M CPU and 8 GB of RAM. The resulting HW-SW Co-design was targeted at Zedboard with both a hard ARM processor and an XC7Z020 FPGA. Software runtime was obtained from the ARM processor with -O3 compiling option. The accelerators were implemented on the FPGA of Zedboard. ISE 14.7 was used to implement the overlay based FPGA accelerators.

The acceleration system handles the input data loading, accelerator computation and output data storing sequentially. The performance of the accelerators is calculated using (4.1) and (4.2) in Section 4.2.3. The data transfer latency used in (4.2) is estimated based on Zedboard DMA between main memory and FPGA on-chip

Table 4.2: DMA transfer latency on Zedboard through AXI high performance port

| transfer size (word, 32bit) | ≥512 | 256 | 128 | 64 | 32 | 16 | ≤8 |
|---|---|---|---|---|---|---|---|
| Latency per word (ns) | 10.08 | 11.28 | 13.32 | 15.18 | 21.45 | 36.24 | 63 |

Table 4.3: QuickDough unrolling setup

| | MM | FIR | SE | KM |
|---|---|---|---|---|
| Unrolling | $1 \times 5 \times 100$ | $50 \times 50$ | $16 \times 16 \times 3 \times 3$ | $125 \times 4 \times 2$ |
| DFG size | 750 | 2500 | 9720 | 5768 |
| Full Loop | $100 \times 100 \times 100$ | $10000 \times 50$ | $128 \times 128 \times 3 \times 3$ | $5000 \times 4 \times 2$ |

buffer through AXI high-performance port. The transfer latency is detailed in Table 4.2. When the transfer size is not included in the table, a simple linear model is used to estimate its latency. Fmax and the number of cycles were extracted from the ISE14.7 and SCGRA scheduler respectively.

Loop unrolling is a critical design input parameter for FPGA loop accelerators developed using QuickDough. It determines the parallelism that are exposed to the overlay architecture and influences the accelerator selection. While tuning the major accelerator design parameters together helps to achieve an optimized loop accelerator design, it requires more design efforts as to be detailed in Chapter 5. Table 4.3 shows the loop unrolling factors that are used for the loop accelerator generation.

### 4.4.3 Accelerator Library Update

To ensure a rapid FPGA accelerator generation, we have implemented a group of SCGRAs based accelerators as the pre-built library by using the SCGRA overlay template. The library is developed to support all the four loop kernels, and it includes 12 3-source-1-destination operations as presented in previous chapter.

In addition, empirical settings are adopted to reduce the number of accelerators to be built in library. Input and output buffer depth are set to be the same. The depth of the address buffers are set to be twice with that of the I/O buffer depth. The data memory in each PE consumes only one primitive block RAM. Instruction memory depth and I/O buffer depth are set to be $2^n$K $(n = 0, 1, 2, ...)$. The SCGRA overlay

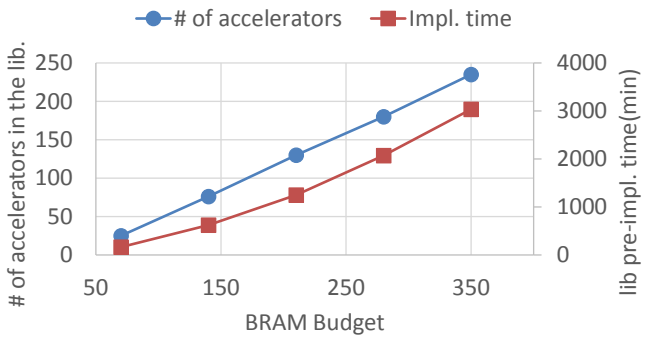



Figure 4.4: Accelerator library size and implementation time given different BRAM budgets.

adopts a torus topology, and the row size is set to be equal to the column size or larger by one for the sake of performance. Eventually, different accelerator configurations mainly differ on the on-chip I/O buffer depth, SCGRA size and instruction memory depth when the data width is determined.

To explore the library update process efficiency, we have evaluated the number of accelerators included in the accelerator library and the time consumed to implement the library when different block RAM budgets ranging from 70, 140, 210, 280 to 350 RABM36 (Note that there are 140 RAMB36 on Zedboard FPGA) are provided. As presented in Figure 4.4, when the BRAM budget increases, the number of accelerators in the library and the library implementation time increase almost linearly. With a single computer, the accelerator library implementation time ranges from 164 minutes to 3035 minutes. However, with a cluster, the time cost of the highly parallel library implementation can decrease by an order of magnitude easily.

### 4.4.4 Accelerator Generation Time

In this section, the loop accelerator generation time of QuickDough is evaluated. It is used as an indicator on the designer's productivity as it greatly limits the number of compile-debug-edit cycles achievable per day.

In order to evaluate the loop accelerator generation time, we took the FPGA resource on Zedboard as the resource budget and pre-built the accelerator library. Then FPGA loop accelerators were generated for each application in the bench-

Table 4.4: Accelerators generated using QuickDough

| Opt. option | Resulting Config. | MM | FIR | SE | KM |
|---|---|---|---|---|---|
| O0 | SCGRA size | $2 \times 2$ | $2 \times 2$ | $2 \times 2$ | $2 \times 2$ |
| | Inst. Mem depth | 4K | 4K | 4K | 4K |
| | I/O buffer depth | 4K | 4K | 4K | 4K |
| | Grouping factor | 50x5x100 | 2500x50 | 128x64x3x3 | 1250x4x2 |
| O1 | SCGRA size | $3 \times 3$ | $3 \times 3$ | $3 \times 3$ | $5 \times 5$ |
| | Inst. Mem depth | 2K | 2K | 4K | 1K |
| | I/O buffer depth | 2K | 2K | 1K | 2K |
| | Grouping factor | 25x5x100 | 1250x50 | 64x32x3x3 | 1000x4x2 |
| O2 | SCGRA size | $3 \times 3$ | $4 \times 4$ | $4 \times 4$ | $5 \times 5$ |
| | Inst. Mem depth | 2K | 1K | 2K | 1K |
| | I/O buffer depth | 2K | 8K | 1K | 2K |
| | Grouping factor | 25x5x100 | 5000x50 | 64x32x3x3 | 1000x4x2 |

mark using the three different accelerator selection options i.e. O0, O1 and O2. Table 4.4 shows the configurations of the resulting FPGA accelerators as well as corresponding grouping factors.

With the pre-built library, every implementation iteration in QuickDough involves 3 steps:

- DFG generation: The compute kernel is translated to corresponding DFG.
- Accelerator selection and DFG scheduling: Select an accelerator configuration and schedule the DFG to it through an operation scheduling.
- Bitstream generation: The scheduling result is embedded into a pre-built accelerator bitstream to produce the final FPGA bitstream of the compute kernel.

Figure 4.5 shows loop accelerator generation time of QuickDough. Both DFG generation and bitstream integration are very fast compared to the DFG scheduling step. The DFG scheduling is relatively slower, but it usually completes in a few seconds. Since the DFG scheduling process must be repeated when QuickDough explores different SCGRA size in the accelerator library, the time consumption increases accordingly. Typically, the accelerator generation time is relatively longer for more intensive accelerator selection. With O0 selection where only a single DFG scheduling process targeting 2x2 SCGRA is needed, the accelerator can be

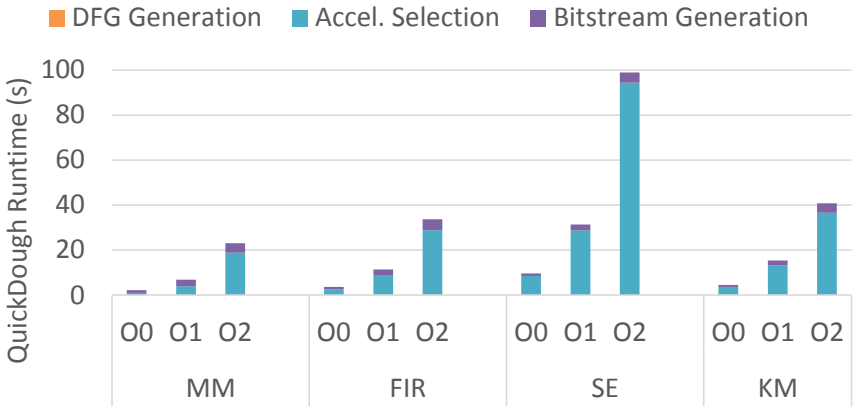



Figure 4.5: Time Consumption of Loop Accelerator Generation Using QuickDough.

produced in less than 10 seconds. With O1 selection where three typical size of SCGRA (i.e. 2x2, 3x3, 5x5) will be evaluated, the accelerator generation process completes in around half a minute. With the O2 selection where an exhaustive search is performed, there are 76 accelerator configurations but only 7 different types of SCGRA size are included in the library. 7 DFG scheduling processes are needed and the accelerator is generated in one or two minutes.

Clearly, the designer must spend the time to physically pre-implement the overlay architecture on the target FPGA, spending considerable time on the implementation tools. However, it can be reused by the whole benchmark. Moreover, the designer may iterate via the above rapid steps during design and debugging phases using an initial overlay implementation. Once the functionality is frozen, the designer may then opt to further optimize performance through more intensive overlay customization and update the library. We argue that the ability to separate functionality and optimization concern, and the possibility of performing rapid debug-edit-implement iterations in QuickDough are crucial factors that contribute to a high-productivity design experience.

### 4.4.5 Performance

While improving designers' productivity is the primary goal of QuickDough, the FPGA accelerators that it generates must remain competitive in terms of performance when compared to corresponding software executed on general purposed

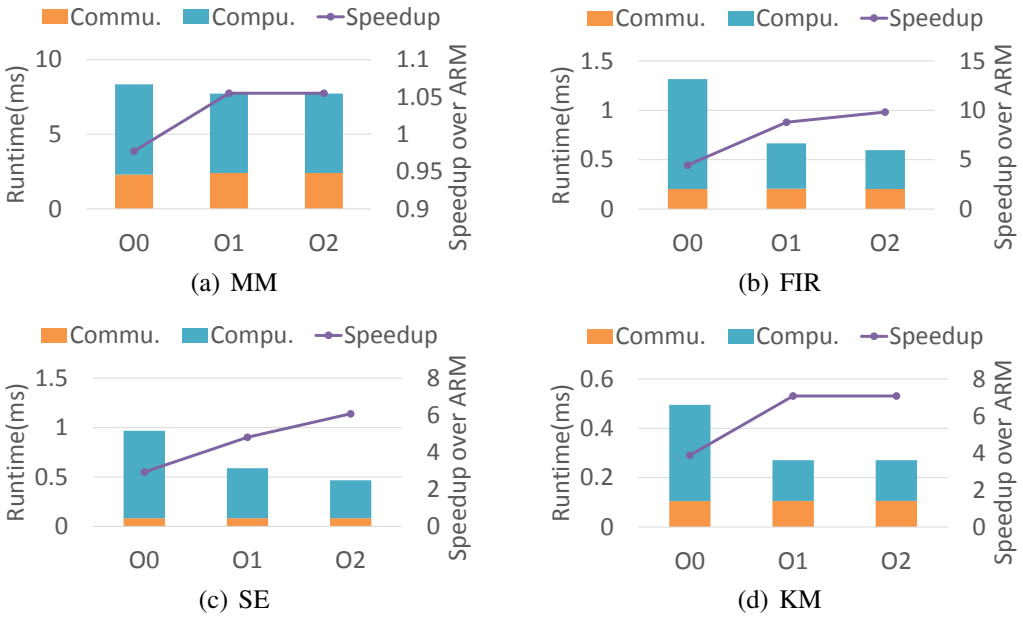



Figure 4.6: Benchmark performance speedup over software executed on ARM processor and execution time decomposition of loop accelerators generated using QuickDough.

processors. Therefore, execution time of the loop kernels executed on ARM processor of Zedboard and FPGA accelerators generated using QuickDough are compared.

Figure 4.6 shows the accelerator performance speedup over software execution on the ARM processor and execution time decomposition of the 4 benchmark programs. The reported loop execution time on the accelerators includes time spent on I/O data communication between FPGA and the ARM processor as well as FPGA computation.

The results in Figure 4.6 show that the accelerators generated using QuickDough are capable to provide up to 9X performance speedup over software executed on ARM processor. For FIR, SE as well as KM which have abundant parallelism and moderate I/O requirements, the maximum speedup goes up to 9X, 6X and 6X respectively. Even with simple acceleration selection and smallest SCGRA size, clear performance speedup can be observed. MM optimized by simple loop unrolling is eventually reduced to a matrix-vector multiplication, so the compute kernel has low compute-to-IO rate and the single port connection between compute logic and input/output buffers becomes the bottleneck hindering the performance of the ac-

celerator. (Note that the communication time in Figure 4.6 simply shows the data movement time from/to main memory to/from accelerator IO buffers, it is NOT a sign of compute-to-IO rate. Usually, we may use operations per IO as the metric of compute-to-IO while IO indicates the load/store from/in input/output buffers.)

Accelerators with larger SCGRA overlay size typically achieve better performance than the ones with smaller overlay size. However, larger SCGRA overlay will not always guarantee better performance. As shown in the experiments, the largest $5 \times 5$ SCGRA overlay based accelerator is only optimal for KM. There are a few reasons for such an accelerator selection. First of all, accelerators with larger overlay size consume more block RAM for instruction memory leaving less block RAM for I/O buffer. As a result, the I/O buffer may limit the transfer size between main memory and FPGA on-chip I/O buffer and reduce the chance of data reuse between DFGs included in a single group. This increased number of transfer between main memory and FPGA significantly limits the overall performance accordingly. Secondly, accelerator with larger SCGRA overlay may confront scheduling problem as larger SCGRA overlay requires larger average cost between PEs and the compute performance may degrade as well. Finally, larger SCGRA overlay based FPGA accelerators may result in lower implementation frequency and degrade the overall performance as well. Optimal accelerators have the best trade-off on buffering, scheduling and implementation frequency. According to Figure 4.6, 3x3 or 4x4 SCGRA based FPGA accelerator achieve relatively better performance.

### 4.4.6 Implementation Frequency and Hardware Overhead

One advantage of employing a simple and regular overlay architecture allows highly pipelined implementations with much higher frequencies as shown in Figure 4.7. The increased running frequency in turns results in higher overall performance of the system. Though both larger SCGRA overlay size and deeper instruction memory may degrade the implementation frequency, the FMax of the implementations is typically around 250MHz on Zedboard FPGA which is much higher than random

logic synthesis on Zedboard.

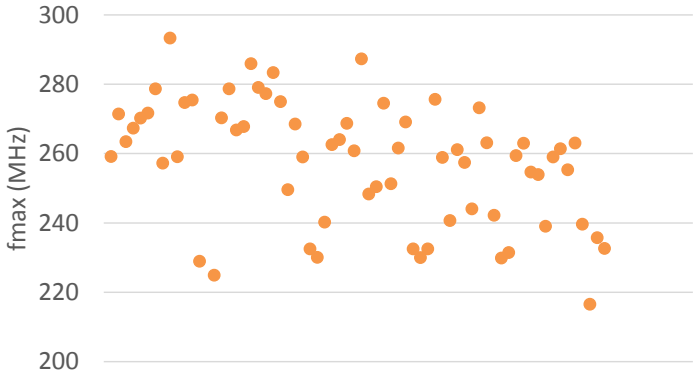



Figure 4.7: fmax of The Accelerators Generated Using QuickDough

As presented in Figure 4.8, block RAM is the resource bottleneck for the SCGRA overlay based FPGA accelerators. It may result in lower implementation frequency as the high utilization may cause tight placing and routing. LUT, FF and DSP48 overhead mainly depends on the SCGRA overlay size and only a portion of them are utilized.

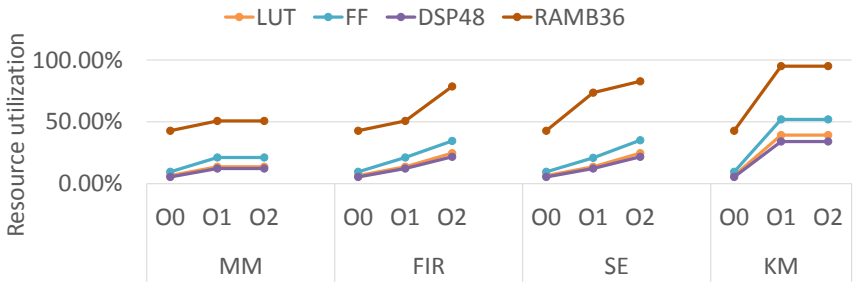



Figure 4.8: FPGA Accelerator Recource Utilization

## 4.5 Summary

In this work, the QuickDough compilation framework for high productivity development of FPGA-based accelerators is presented. QuickDough makes use of a soft coarse-grained reconfigurable array as an overlay architecture to greatly improve the designer's compilation experience. Taking advantage of a pre-built SCGRA overlay based accelerator libray, the lengthy low-level FPGA accelerator implementation is reduced to a rapid operation scheduling problem and the compilation

time of QuickDough is reduced to seconds, which contributes directly into higher application designers' productivity. Despite the use of an additional layer of overlay architecture on the target FPGA, the overall application performance remains competitive in many cases.

# Chapter 5

# Loop Accelerator Customization

Despite the great advantages on design productivity, the additional layer on top of the physical FPGA inevitably imposes performance and resource consumption penalty. An overlay must ensure that the overall FPGA acceleration performance remains competitive. Otherwise, mapping the loop kernels to the overlay based FPGA accelerators will not be as useful. While a random SCGRA overlay configuration can rarely meet the performance requirements and the resource consumption budgets, therefore the capability to customize the overlay specifically to an application or a domain of application is essential to the overlay based FPGA accelerator design. However, navigating through a labyrinth of architectural and compilation parameters to fine-tune an accelerator's performance is a slow and non-trivial process. To require a user to manually explore such vast design space is going to counteract the productivity benefit of the utilizing overlay in the first place.

In order to address this problem, QuickDough can also automatically customize the overlay architectural parameters and exploit the optimized loop unrolling and hardware-software communication as well as buffer sizing specifically to a high-level compute intensive loop kernel with given high-level resource constraints. Most importantly, the customization makes all the hardware design and optimization details transparent to the users, which makes the whole system accessible to high-level designers. The user may choose to perform the customization only when the loop is changed dramatically and previous customization doesn't fit the updated loop

kernel. In particular, by taking advantage of the regularity of the SCGRA overlay, a multitude of design metrics such as performance, hardware consumption and energy efficiency can be accurately estimated using analytical models once the overlay scheduling result is available. Since the overlay scheduling depends on much fewer design parameters, the overall customization framework can be dramatically simplified. With both the efficient application-specific customization and rapid compilation, the proposed design framework ensures both high design productivity and high performance of FPGA loop acceleration.

Application-specific customization provides unique opportunity to reduce the resource consumption and improve performance and energy efficiency of the resulting accelerators. However, taking the system as a black box and exhaustively searching all the possible configurations can be inefficient and slow. In this work, by taking advantage of the regularity of the SCGRA overlay based FPGA accelerator, we can reduce the complex customization problem to a much simpler sub design space exploration (DSE) together with a simplified search problem. With the customization, optimized application-specific nested loop accelerator can be produced efficiently.

## 5.1 Customization Problem Formulation

In this section, the customization problem of the nested loop acceleration on an SCGRA overlay based FPGA accelerator is formalized. Various design metrics including performance which is expressed as loop run time, energy consumption, energy efficiency and hardware resource consumption can either be used as the optimization goals or design constraints. However, the accelerator with smallest SCGRA overlay configuration and input/output data buffer apparently consumes the least FPGA resources while the resulting performance usually will not be acceptable, which is not quite useful in reality. Typically resource consumption will be used as design constraints instead of design goals. Energy consumption is not an appropriate design goal as well while the reason is not as obvious. As the power

consumption model used in this work doesn't take the circuit activity into consideration, the power consumption is proportional to the SCGRA overlay size while the accelerator performance improves slower with the increase of the SCGRA overlay size. As a result, accelerator with a single PE will consume the least energy with little performance benefit. Basically both the energy consumption and the resource consumption are used as optional design constraints. In this formulation, performance is set to be the design goal while the rest metrics are used as design constraints.

Table 5.1: Design Parameters of Nested Loop Acceleration

| Design Parameters | | Denotation |
|---|---|---|
| Nested Loop Compilation | Loop Unrolling Factor | $\boldsymbol{u} = (u_0, u_1, ...)$ |
| | Grouping Factor | $\boldsymbol{g} = (g_0, g_1, ...)$ |
| Overlay Configuration | SCGRA Topology | 2D Torus, fixed |
| | SCGRA Size | $r \times c$ |
| | Data Width | $W_0$ |
| | Data Mem | $D_0 \times W_0$ |
| | Input Buffer | $D_1 \times W_0$ |
| | Output Buffer | $D_2 \times W_0$ |
| | Instruction Mem | $D_3 \times W_1$ |
| | Input Address Buffer | $D_4 \times W_2$ |
| | Output Address Buffer | $D_5 \times W_2$ |
| | Operation Set | fixed |
| | Implementation Frequency | $f$, fixed |
| | Pipeline Depth | fixed |

Suppose $\boldsymbol{\Psi}$ represents the overall nested loop acceleration design space. $\boldsymbol{C} \in \boldsymbol{\Psi}$ represents a possible configuration in the design space and it includes a number of design parameters as listed in Table 5.1. Assume that the loop to be accelerated has $n$ nested levels and loop count can be denoted as $l = (l_1, l_2, ..., l_n)$. $R = (R_1, R_2, R_3, R_4)$ stands for the FPGA resource (i.e. BRAM, DSP, LUT and FF) that are available on a target FPGA and $ResConsumption(\boldsymbol{C}, i)$ denotes the four different types of FPGA resource consumption. $Power(\boldsymbol{C})$ represents the power consumption. $Energy(\boldsymbol{C})$ and $EDP(\boldsymbol{C})$ are the estimated enegy consumption and energy delay product respectively while $Energy\_Budget$ and $EDP\_Budget$ are the corresponding budgets from the users. $In(\boldsymbol{g})$ and $Out(\boldsymbol{g})$ stand for the amount of input and output of a group. Similarly, $In(\boldsymbol{u})$ and $Out(\boldsymbol{u})$ stand for the amount of input and output of a DFG. $DFGCompuTime(\boldsymbol{C})$ represents the number of cycles needed

to complete the DFG computation. $\alpha_i$ and $\beta_i$ are constant coefficients depending on target platform where $i = (1,2,...)$. With these denotations, the customization problem targeting minimum run time can be formulated as follows:

Minimize

$$RunTime(\boldsymbol{C}) = CompuTime(\boldsymbol{C}) + CommuTime(\boldsymbol{C}) \tag{5.1}$$

subject to

$$\begin{aligned} &ResConsumption(\boldsymbol{C}, i) \leq R_i, i = 1,2,3,4 \\ &Energy(\boldsymbol{C}) \leq Energy\_Budget \\ &EDP(\boldsymbol{C}) \leq EDP\_Budget \end{aligned} \tag{5.2}$$

$$\begin{aligned} &In(g) \leq D_1 \\ &Out(g) \leq D_2 \end{aligned} \tag{5.3}$$

$$\begin{aligned} &\prod_{i=1}^{n} \frac{g_i}{u_i} \times In(u) \leq D_4 \\ &\prod_{i=1}^{n} \frac{g_i}{u_i} \times Out(u) \leq D_5 \end{aligned} \tag{5.4}$$

$$DFGCompuTime(\boldsymbol{C}) \leq D_3 \tag{5.5}$$

The estimated hardware consumption, energy consumption and energy delay product should be within the budgets of the users, and the high-level constraints shown in (5.2) should be fulfilled. On top of the basic high-level design constraints, there are a number of accelerator architectural constraints. As the accelerator transfers data in the granularity of a group, the input/output buffers should be able to accommodate all the input/output data of a group and constraints in (5.3) should be satisfied. While the address buffers contain all the load/store addresses of DFGs

in the same group, the total amount of the address buffers should be equal to or larger than the total amount of input/output of the DFGs in a group. As there may be data reuse between neghboring DFGs, the address buffer depth should be usually larger than the total amount of input/output of a group as shown in (5.4). The instruction memory stores the control words of every cycle of PEs. Therefore, the instruction memory depth should be equal to or larger than the number of cycles of DFG execution as shown in (5.5).

$RunTime(\boldsymbol{C})$ represents the number of cycles needed to compute the loop on the CPU-FPGA system. It consists of both the time consumed for computing on FPGA and communication between FPGA and host CPU. Assume there is no pre-fetching or overlap between computing and communication, and then $RunTime(\boldsymbol{C})$ can be calculated using (5.1).

Since the unrolled part of the loop will be translated to DFG and then scheduled to the SCGRA overlay. Thus the DFG computation time is essentially a function of $\mathbf{u}$, $r$ and $c$, and it can also be denoted by $DFGCompuTime(\mathbf{u}, r, c)$. The nested loop is computed by repeating the same DFG execution, and the nested loop computation can be calculated using (5.6).

$$CompuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{u_i} \times DFGCompuTime(\mathbf{u}, r, c) \tag{5.6}$$

DMA is typically used for the bulk data transmission. Communication cost per data can be modeled with a piecewise linear function and thus DMA latency can be calculated using $DMA(x)$ where $x$ represents the amount of DMA transmission. Assume the input data and output data are transferred sequentially, and then the communication time of the whole nested loop can be calculated by (5.7).

$$CommuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{g_i} \times (DMA(In(\mathbf{g})) + DMA(Out(\mathbf{g}))) \tag{5.7}$$

On top of the performance model, we futher developed energy model and energy efficiency model. These models can be found in the appendix.

## 5.2 Customization Method

To solve the customization problem as formalized in previous section, an automatic FPGA loop accelerator customization method based on the QuickDough compilation framework is presented as shown in Figure 5.1.

The customization method can be roughly divided into two steps. In the first step, a sub DSE targeting loop execution time is performed and the feasible design space can be obtained. Since loop execution time is mostly determined by the operation scheduling which simply depends on the loop unrolling factor and SCGRA size, the sub DSE is much simpler compared to the overall system DSE which includes more than 10 design parameters. In the second step, each configuration in the feasible design space will be evaluated. Instead of using simulation based methods, analytical models as proposed in previous section are employed to estimate the accelerator metrics including performance, energy consumption, energy efficiency and hardware resource consumption. These analytical models are accurate because of the regularity of the SCGRA overlay. Even though the feasible design space is still large, it is very fast to evaluate all the configurations in it when compared to the time cost of the SCGRA scheduling process. After the evaluation process, customization for best performance becomes trivial and the optimized design parameters can be obtained immediately.

### 5.2.1 Sub Design Space Exploration

The first step of the customization method is to perform the sub DSE for shorter loop computation time. As shown in Figure 5.1, this sub DSE focuses on the operation scheduler. The operation scheduling process is fast but not trivial. Although the sub DSE is already much simpler than the overall system DSE, it still requires to repeat the operation scheduling many times and straightforward sub DSE will take a large amount of time. Fortunately, it is clear that the operation scheduling only depends on the SCGRA overlay size and the loop unrolling factor. In addition, the two design parameters basically have monotonic influence on the DFG execution time

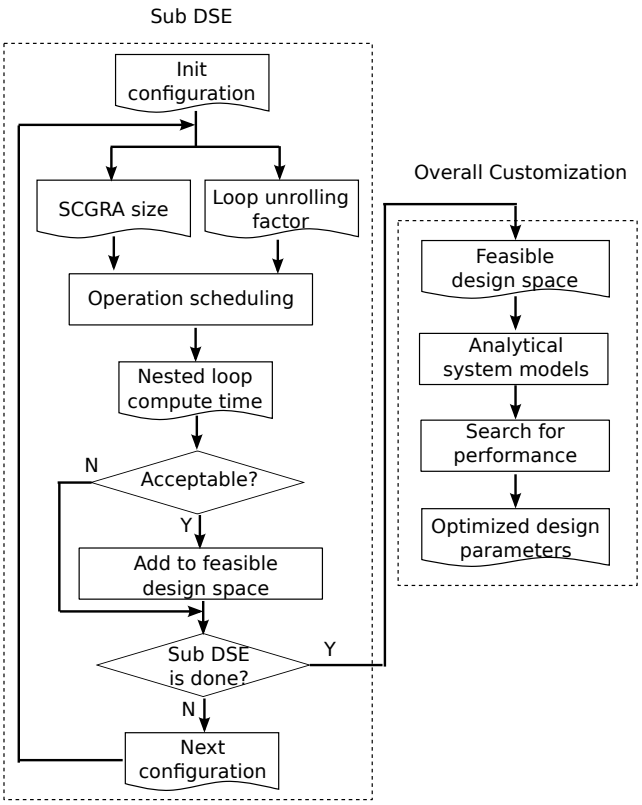
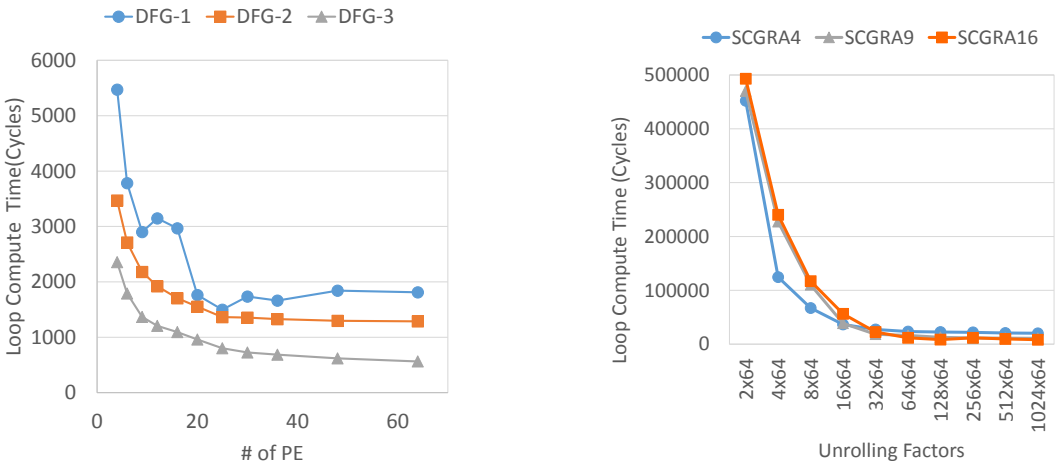



Figure 5.1: Two-Step Customization Method

according to our experiments as shown in Figure 5.2. Typically both the larger loop unrolling factor and SCGRA size help to improve the performance of the DFGs mapped to the SCGRA overlay, though there are minor exceptions caused by the complex operation scheduling. Meanwhile, it can be found that the improvement slows down when the two design parameters get larger. When the improvement reaches a threshold, the increase of the design parameters has negligible influence on the DFG execution time.

With this observation, the feasible configurations must satisfy (5.8) and (5.9), where $\Phi$ denotes the feasible design space and $\varepsilon$ indicates the percentage of the performance benefit obtained by the increase of loop unrolling or SCGRA size. Whenever a configuration fails the sub DSE condition, all the configurations which are larger on one design parameter and remain the same on the rest design parameters can be safely pruned. Note that $\varepsilon$ must be small enough to prune the configurations that are not worthwhile.




(a) SCGRA Size (b) Loop Unrolling

Figure 5.2: The design parameters typically have monotonic influence on the loop computation time and the computation time benefit degrades with the increase of the design parameter. (a) SCGRA Size, the SCGRA topology used are torus with $2 \times 2$, $3 \times 2$, $3 \times 3$, ... while DFG-1, DFG-2 and DFG-3 are DFGs extracted from matrix-matrix multiplication, fir and Kmean respectively. (b) Unrolling Factor, the loop used is a 63-tap Fir with 1024 input.

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r', c', ...) \in \Phi, \\ &(r+1 == r' \text{ and } c == c') \text{ or } (r == r' \text{ and } c+1 == c') : \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \varepsilon \end{aligned} \tag{5.8}$$

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r, c, ...) \in \Phi, \\ &\boldsymbol{u} \text{ and } \boldsymbol{u'} \text{ are consecutive unrolling factors :} \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \varepsilon \end{aligned} \tag{5.9}$$

In order to achieve the desired customization, the feasible design space defined by (5.8) and (5.9) must be able to cover the configuration that produces the optimal customization. A brief proof is presented as follows.

$\forall \boldsymbol{C'} \notin \Phi$, there must be a configuration $\boldsymbol{C}$ that fails (5.8) or (5.9). Suppose $\boldsymbol{C'} = (..., \boldsymbol{u}, r+1, c, ...)$ and $\boldsymbol{C} = (..., \boldsymbol{u}, r, c, )$. Thus it can be concluded that

$$CompuTime(\boldsymbol{C}) \geq CompuTime(\boldsymbol{C'}) \geq (1-\varepsilon) \times CompuTime(\boldsymbol{C}) \tag{5.10}$$

Since $\varepsilon$ is small, $CompuTime(\boldsymbol{C'})$ is almost equal to $CompuTime(\boldsymbol{C})$. Meanwhile, the I/O buffer depth of both configurations are the same, so the $CommuTime(\boldsymbol{C'})$ is equal to $CommuTime(\boldsymbol{C})$. Therefore it can be concluded that $RunTime(\boldsymbol{C'}) \approx RunTime(\boldsymbol{C})$ according to (5.1). In addition, as the unrolling factors of configuration $\boldsymbol{C}$ and $\boldsymbol{C'}$ are the same, $DFGCompuTime(\boldsymbol{C'}) \approx DFGCompuTime(\boldsymbol{C})$, which means that the instruction memory depth of configuration $\boldsymbol{C'}$ will be the same with that of configuration $\boldsymbol{C}$. While the increase of row size of the SCGRA overlay will result in significant BRAM consumption and power consumption. Thus $Power(\boldsymbol{C'}) \geq Power(\boldsymbol{C})$. As mentioned in previous paragraph, $RunTime(\boldsymbol{C'})$ is larger or equal to $RunTime(\boldsymbol{C})$. According to (A.3), it is clear that $Energy(\boldsymbol{C'}) \geq Energy(\boldsymbol{C})$. Taking all these design metrics into consideration, any configuration that is pruned during the sub DSE will not be an optimized configuration. Similarly, we can also draw the same conclusion when different occasions in (5.8) and (5.9) appear.

The sub design space exploration essentially aims to remove the configurations that will not be able to produce optimal design goals. As analyzed in previous paragraphs, instead of targeting the design goal directly, it simply focuses on the loop computing time which are mostly determined by the loop unrolling and SCGRA size. While the two design parameters have clear monotonic influence on the loop computing time, a simple branch and bound algorithm as detailed in Algorithm 2 is used to explore the sub design space. It starts with a configuration with minimum SCGRA overlay size and unrolling factor. As a torus topology is used, the exploration analyzes the SCGRA row size, SCGRA column size and loop unrolling factor respectively. The increase of each parameter is evaluated using a revenue function $Revenue(\boldsymbol{C}, \boldsymbol{C'})$. When the revenue is larger than the pre-defined threshold, the configuration will be regarded as a feasible configuration and thus added to the feasible

design space $\Phi$.

**Algorithm 2** Sub Design Space Exploration.

**procedure**
Initialize $r = 2, c = 2, \boldsymbol{u} = (1,1,...)$, feasible design space $\Phi = \emptyset$, $\boldsymbol{C} = (...,r,c,\boldsymbol{u},...)$, maximum SCGRA overlay $r_{Max} \times c_{Max}$.
**while** $r < r_{Max}$ **do**
**while** $c < c_{Max}$ **do**
**while** $\boldsymbol{u}$ is not fully unrolled **do**
Generate DFG with $\boldsymbol{u}$
DFG Scheduling with configuration $\boldsymbol{C}$
Estimate performance $CompuTime(\boldsymbol{C})$
Get neighbor $\boldsymbol{C'} \in \Phi$ with smaller loop unrolling
**if** $\boldsymbol{C'}$ exists and $Revenue(\boldsymbol{C},\boldsymbol{C'}) \leq \varepsilon$ **then**
Break
**else**
Add $\boldsymbol{C}$ to $\Phi$
**end if**
update $\boldsymbol{u}$ with larger neighbor unrolling factor
**end while**
Get neighbor $\boldsymbol{C''} \in \Phi$ with smaller column size
**if** $\boldsymbol{C''}$ exists and $Revenue(\boldsymbol{C},\boldsymbol{C''}) \leq \varepsilon$ **then**
Break
**end if**
$c = c + 1$
**end while**
Get neighbor $\boldsymbol{C'''} \in \Phi$ with smaller row size
**if** $\boldsymbol{C'''}$ exists and $Revenue(\boldsymbol{C},\boldsymbol{C'''}) \leq \varepsilon$ **then**
Break
**end if**
$r = r + 1$
**end while**
**end procedure**

**procedure** $Revenue(\boldsymbol{C},\boldsymbol{C'})$
return $\frac{CompuTime(\boldsymbol{C'}) - CompuTime(\boldsymbol{C})}{CompuTime(\boldsymbol{C'})}$
**end procedure**

### 5.2.2 Overall Customization

The second step is to perform the overall FPGA loop accelerator customization and to produce the optimized FPGA loop accelerator. As the sub DSE in the first step have already determined a set of loop unrolling and SCGRA overlay size that may provide the optimized FPGA loop accelerator, this step focuses on the rest of

the parameters of the FPGA loop accelerator including loop grouping, input/output buffer capacity, input/output address buffer capacity as well as instruction memory capacity.

Given feasible loop unrolling and SCGRA overlay size, the number of cycles that is needed to complete the DFG extracted from the unrolled loop on the specified SCGRA overlay is immediately available. Then the optimized instruction memory depth which is larger or equal to the cycle count of the DFG execution can be obtained. Meanwhile, loop grouping which repeats the unrolled loop multiple times can be further explored. Given a possible loop grouping, the input/output data buffer capacity which is equal to or larger than the amount of input/output of the group can be thus decided.

While the address buffer capacity is relatively difficult to decide, it is equal to or larger than the amount of the loop iteration input/output multiplied by the number of loop iterations (i.e. the unrolled loop body) included in each group. The depth of the address buffer is typically larger than that of the corresponding data buffer depending on the input/output data reuse between neighboring loop iterations. When there is no data reuse, the address buffer depth should be equal to that of the corresponding data buffer. When all the input data are reused and each loop iteration accesses the input data differently, then the input address buffer depth is $n$ times larger where $n$ is the number of loop iterations in the group. Note that the memories of the SCGRA overlay based FPGA accelerator are constructed using primitive block RAMs of the FPGA devices, so these memory capacity should be able to be divided by the primitive block RAM capacity.

When all the potential configurations are decided, design goals including performance, energy consumption, energy efficiency can be estimated using the models proposed in previous section. Finally, the configuration that produce the optimal design goals is thus considered to be the customized configuration. In addition, when the customized configuration is decided, both the FPGA loop accelerator and the corresponding loop accelerator interfaces will be generated providing a com-

plete hardware accelerator solution to the user. As the overall customization step searches through the feasible design space mostly based on the analytical models, the run time is negligible to that spent in the first step.

In addition, as all the design metrics of the feasible design metrics can be rapidly obtained, it is also possible to present a series of Pareto-optimal design spaces such as Energy-Performance and Resource Consumption-Performance allowing the users to make the final decision.

## 5.3 Experiments

The experiments mainly include two parts. In the first part, the implementations of the SCGRA overlay based FPGA accelerators with different configurations are analyzed to demonstrate the regularity of the SCGRA overlay based FPGA accelerators, which are the basis of the proposed models. In the second part, both the efficiency and quality of the proposed customization framework is evaluated. In the experiments, the time consumption of the loop accelerator customization using the proposed two step customization is measured and compared to that of an exhaustive search. Then the performance speedup over a hard ARM processor is also presented. Meanwhile, the performance of the resulting accelerators generated using both the proposed customization and an exhaustive search are also compared.

### 5.3.1 Experiment Setup

The customization run-time was obtained using a computer with Intel(R) Core(TM) i5-3230M CPU and 8GB RAM. Zedboard which has an ARM processor and an FPGA was used as the computation system. PlanAhead 14.7 was used for the SCGRA overlay based design. As the customization relies on analytical models instead of physical implementations, all the overlay implementations on Zedboard is assumed to work at 250MHz. To perform the customization, $\varepsilon$ is set to be 0.05 and all the resource on Zedboard is set to be the resource budget. Software run-time is

obtained from ARM processor of Zedboard.

In this section, four applications including Matrix Multiplication (MM), FIR, Kmean(KM) and Sobel Edge Detector (SE) are used as the benchmark. The configurations of the benchmark are the same with that detailed in the experiments in Chapter 4.

### 5.3.2 Customization Time

Although customization is not a frequent process of QuickDough, it is also important to the productivity of the designers. Figure 5.3 shows the customization time of both the proposed two step (TS) customization and an exhaustive search based customization (ES). TS typically completes the customization in 10 minutes to 20 minutes and it is around 100X faster than the ES on average. In particular, ES is extremely slow on MM which has three levels of loop with relatively large loop count and thus larger design space. Though TS does need a dozen of minutes to complete the customization, it skips most of the unfeasible configurations and the run time is less sensitive to the size of the design space.

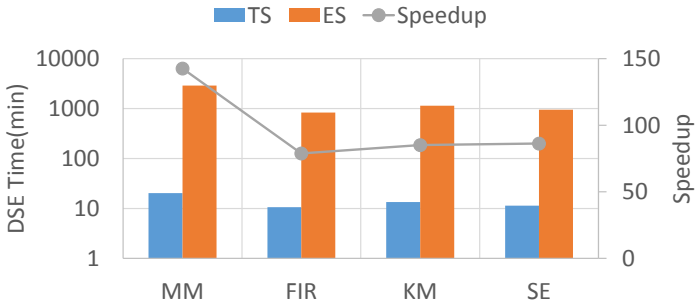



Figure 5.3: Customization Time Using Both TS and ES

In order to evaluate the sensitivity of the threshold $\varepsilon$ to the customization, both of the customization time and the resulting accelerator run-time is analyzed with a set of different $\varepsilon$ setup. Figure 5.4 shows the influence of the $\varepsilon$ on the customization of FIR. When the $\varepsilon$ is large, the customization time is much faster while the resulting accelerator performance is not optimized. On the other hand, smaller $\varepsilon$ typically helps to produce accelerators with shorter runtime of accelerators, but customization time increases dramatically. In particular, when $\varepsilon$ is small enough and optimized

accelerator has already been found, even smaller $\varepsilon$ will not lead to accelerators with better performance. Experiments on the rest of the applications show similar results and they are omitted to save the space. The optimal $\varepsilon$ is up to the target application, while 0.05 works fine for all the applications in this benchmark.

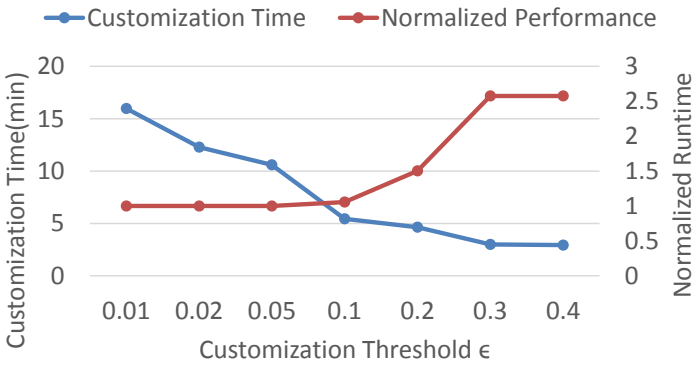



Figure 5.4: $\varepsilon$ Influence on FIR Customization Time and Resulting Accelerator Run-time

### 5.3.3 Customized Accelerator Performance

The proposed design framework will not be as useful if the performance of the generated system fails to provide competitive performance speedup over a general purpose processor which software designers can easily work with. Therefore, the computer kernel run-time on an ARM processor is presented as a basic comparison. In addition, in order to demonstrate both the necessity of the customization and the quality of proposed two-step customization method, the performance of the accelerators with a random configuration and customized configurations obtained using ES are compared with the performance of the accelerators customized using the proposed TS method. The detailed configurations of the accelerators are listed in Table 5.2. Basically, the random configurations may come from an user who doesn't quite understand the underlying overlay architectures. The configurations generated using ES may cover all the possible configurations. The configurations generated using TS only explores the representative configurations expressed by the proposed models.

The performance comparison is shown in Figure 5.5. It is clear that random configuration is usually far from the optimized configuration. Therefore, customization

Table 5.2: Accelerator configurations (Note that the configurations include loop unrolling factor, grouping factor, SCGRA array size, instruction memory depth and IO buffer depth)

| | | |
|---|---|---|
| MM | Base | $(1 \times 2 \times 100, 4 \times 2 \times 100, 5 \times 5, 1k, 2k)$ |
| | TS | $(1 \times 5 \times 100, 50 \times 5 \times 100, 4 \times 4, 1k, 8k)$ |
| | ES | $(1 \times 5 \times 100, 25 \times 5 \times 100, 5 \times 4, 1k, 8k)$ |
| FIR | Base | $(10 \times 50, 100 \times 50, 3 \times 3, 1k, 2k)$ |
| | TS | $(50 \times 50, 2000 \times 50, 4 \times 4, 1k, 4k)$ |
| | ES | $(100 \times 50, 5000 \times 50, 5 \times 4, 1k, 8k)$ |
| SE | Base | $(4 \times 4 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 3 \times 2, 1k, 8k)$ |
| | TS | $(16 \times 16 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 4 \times 4, 1k, 4k)$ |
| | ES | $(16 \times 16 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 5 \times 4, 1.5k, 4k)$ |
| KM | Base | $(25 \times 4 \times 2, 2500 \times 4 \times 2, 4 \times 3, 1k, 8k)$ |
| | TS | $(125 \times 4 \times 2, 625 \times 4 \times 2, 5 \times 5, 1k, 2k)$ |
| | ES | $(125 \times 4 \times 2, 625 \times 4 \times 2, 5 \times 5, 1k, 2k)$ |

is needed to provide meaningful FPGA loop accelerators. The customized accelerators obtained using TS achieves up to 10X speedup over the ARM processor on the benchmark. Particularly, the customized accelerator For FIR, SE and KM, the speedup is promising. MM has relatively low compute-IO rate and the single input and output between the on-chip buffer and the SCGRA overlay limits the performance of the accelerator. This problem can hopefully be alleviated by appropriate on-chip buffer partition, which will be supported in the proposed framework in future. Meanwhile, it can also be found that the performance of the accelerators customized using TS is quite close to the performance of that customized using ES.

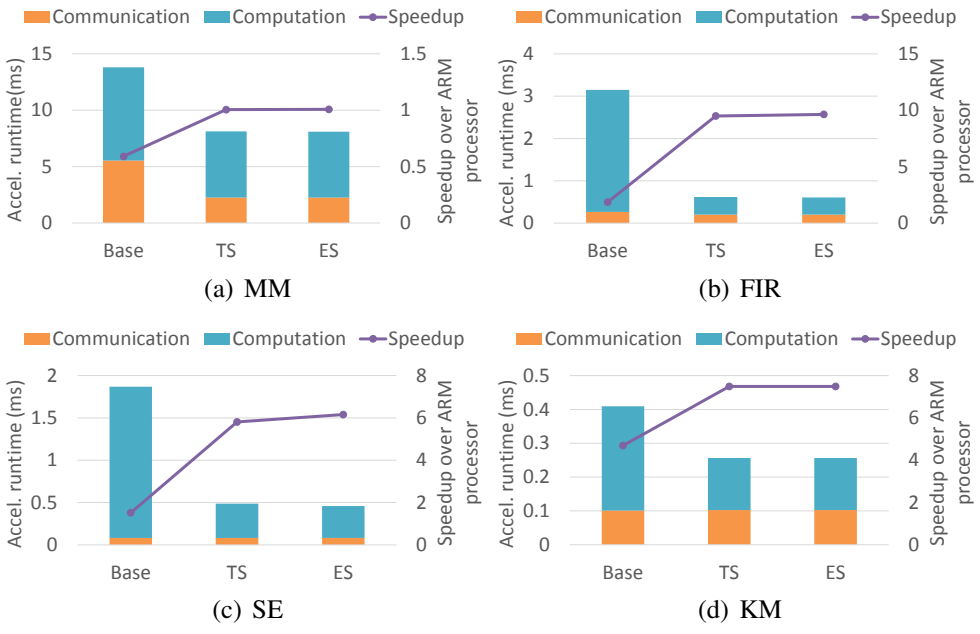



Figure 5.5: Customized FPGA Loop Accelerator Performance Comparison

Finally, the FPGA resource consumption of the resulting accelerators are also compared. As block RAM is the resource bottleneck of the SCGRA overlay based FPGA accelerator design as mentioned in experiments in Chapter 3, only block RAM resource consumption is presented in Figure 5.6 to save the space. As shown in the figure, accelerators with random configurations may consume larger resource with worse performance. Compared to the accelerators produced using ES, the accelerators generated using TS typically consume less block RAM resource while achieving similar performance.

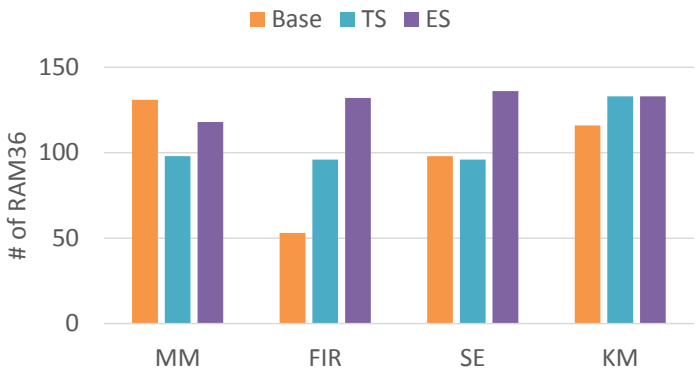



Figure 5.6: Block RAM Consumption Comparison

## 5.4 Summary

In this work, an automatic nested loop accelerator customization framework that is based on a soft coarse-grained reconfigurable array overlay is presented. By taking advantage of the regularity of the SCGRA overlay, a two-step customization method is proposed for intensive system customization specific to the given user application. According to the experiments, it can be performed efficiently, resulting in up to 5 times performance improvement over solutions without customization at the cost of 10 to 20 minutes additional tools run time (The run time here means the customization time. If the configuration of the customized accelerator doesn't exist in the library, additional implementation time is required). Overall, the framework is able to generate accelerators that achieve up to 10 times speed up over software running on the host processor, resulting in a high design productivity experience for

software programmers.

# Chapter 6

# Conclusion and Future Work

In this work, we have presented the QuickDough compilation framework for high productivity development of FPGA-based loop accelerators. QuickDough makes use of SCGRA as an overlay architecture to greatly improve the designer's compilation experience. By making best use of the underlying FPGA fabrics, the QuickDough overlay is highly pipelined and is able to work at high clock frequency. Meanwhile it is highly scalable, simple and easy to extend for application-specific customization for the sake of performance and energy efficiency. With the overlay architecture, QuickDough pre-builds a group of representative SCGRA overlay based accelerators as a library and allows the pre-built library to be reused during the compile-debug-edit design iterations. Consequently, the lengthy low-level FPGA loop accelerator implementation process is reduced to a rapid acceleration selection process and a fast operation scheduling process. The compilation time of QuickDough is reduced to seconds, which contributes directly into higher application designers' productivity. At the same time, QuickDough also generates the communication interfaces which allow the users to make use of the resulting FPGA loop accelerators during the high-level application development and develop a complete HW/SW co-design on a hybrid CPU-FPGA computing system.

Despite the use of an additional layer of overlay architecture on the target FPGA, the overall application performance remains competitive in many cases. Implementation with higher clock frequency resulting from the highly pipelined SCGRA

overlay, in combination with an in-house scheduler that can effectively schedule operations to hide the pipeline latency contributes to the competitive performance of the resulting accelerators generated using QuickDough.

Finally, QuickDough also includes an automatic application-specific customization process which tunes the design parameters of the FPGA loop accelerators such as the loop unrolling, loop grouping, SCGRA overlay configurations and so on. By taking advantage of the regularity of the overlay, intensive system customization specific to the given user application can be performed efficiently, resulting in up to 5 times performance improvement over solutions without customization at the cost of 10 to 20 minutes additional tools run time. Overall, the framework is able to generate accelerators that achieve up to 10 times speed up over software running on the host processor, resulting in a high design productivity experience for software programmers.

## 6.1 Future Work

Although this work has demonstrated that the use of an SCGRA overlay is promising to enhance the designers' design productivity and accessibility to software designers compared to conventional FPGA accelerator development, there are still a number of aspects that may further be studied.

### 6.1.1 High Level Compilation

QuickDough focuses on producing FPGA loop accelerator targeting a hybrid CPU-FPGA computing system. To serve as a complete automatic HW/SW co-synthesis framework for software designers, many high-level design options such as the identification of compute kernels that are most appropriate to offload to FPGAs are still highly required. In addition, when multiple loop kernels from a single application are mapped to the same FPGA device with limited hardware resources, optimized strategies allowing the loop kernels to share the limited FPGA resource while

achieving optimized performance of the overall application need to be explored as well. It can be challenging especially when some of the loop kernels are dependent while some of the loop kernels may be executed in parallel.

On top of the high-level compilation options, compiling the loop kernels to the specified SCGRA overlay is also difficult. In order to compile a high-level loop kernel to an SCGRA overlay, DFG is typically adopted as an intermediate representation (IR). Currently, the generation of DFGs from the loop kernels involves several manual conversion steps to match the user design with the target overlay. It is anticipated that an automated process that is able to analyze the user application and generate the corresponding DFG suitable for the overlay will be developed. Many existing works typically generate the DFGs through well-known compilation framework such as LLVM. However, the compilation framework adopts an IR that is initially developed for processors. It is convenient to map the IR to instructions defined by ISAs of the processors, while it is rather challenging to transform the IRs to DFGs for SCGRA overlay scheduling. An IR that are friendly to CGRA architectures may help to solve this challenge.

Unlike processors which typically have ISAs to separate the micro architecture implementations and high-level application compilation, the SCGRA overlay typically have the implementation details exposed to the operation scheduler. As a result, it is difficult to reuse the same scheduler for SCGRA overlays with different design options such as the pipeline depth, which essentially limits the design space exploration of the SCGRA overlay based FPGA loop accelerator design. A flexible SCGRA overlay abstraction that separates the overlay implementation details and high level compilation may solve this problem.

QuickDough adopts an SCGRA as the overlay architecture and it relies on lock-step computing to exploit the potential parallel operations of the loop kernels, so the data dependency in the loop kernels typically should be known at compilation time. Although some of the high-level code such as simple branches in the loop kernels can be transformed to code with deterministic data dependency through if-

conversion, many high-level code such as `while` loop with data dependent loop count can't be transformed to DFGs and executed on the SCGRA overlay, which greatly limits the adoption of the SCGRA overlay for loop acceleration. One possible solution is to add control support to PEs of the SCGRA overlay so that each PE can support branch and loop. While adding control support to the SCGRA overlay is not a barrier, the basis of lock-step computing that helps to explore the data level parallelism on the SCGRA overlay is no longer existed. To that end, instead of scheduling operations in the same loop iteration to the PEs, multiple loop iterations can be mapped to the same PE of the SCGRA overlay. Basically each PE in the overlay executes sequentially while the overlay executes in parallel globally to explore the loop-level parallelism. The locally sequential execution and globally parallel execution model can be adopted to allow the SCGRA overlays to handle loop kernels with more complex control logic in the loop body. However, when the loop iterations are dependent, both the communication between the PEs and the loop scheduling will be challenging.

### 6.1.2 Heterogeneous SCGRA Overlay

With a homogeneous SCGRA overlay template, this work has already shown the promising performance and energy efficiency improvement through application-specific SCGRA overlay customization. Heterogeneous SCGRA overlay allows even more intensive customization specifically to an user application is thus beneficial to the overlay performance and energy efficiency.

Some of the complex operations such as division and square root may not appear as frequently as simple operations in an loop kernel, but they may still be performance bottleneck or have great influence on the overall loop execution time. At the same time, it is not worthwhile to put these complex operations on host processor which may cause considerable communication between the accelerator and host processor. In addition, implementing the complex operations on all the PEs of a homogeneous SCGRA overlay results in a large amount of resource as well as energy

consumption. In this case, implementing these complex operations on part of the PEs of an heterogeneous SCGRA overlay will outperform on energy consumption while achieving similar performance, while the increased design space makes the DSE even more challenging.

On top of the complex operations, frequent operations with close data dependency may be accumulated as a customized complex operation as well. It brings two-folded benefits to the overall FPGA acceleration system. On the one hand, the customized operation implemented with separate data path typically has shorter latency compared to the group of operations executed on different PEs scheduled by the SCGRA overlay scheduler. In addition, a group of operations with close communication scheduled as a unit on a single PE also reduces the communication which is essentially beneficial to the overall performance of the loop kernel. On the other hand, although the additional data paths of the customized operations consume more FPGA resources including LUT, DSPs and FFs, it reduces overlay execution time and thus the requirements of instruction memory which further alleviates the resource bottleneck of the SCGRA overlay based FPGA accelerators. However, the design space is extremely large and it is rather challenging to determine the optimized customized operations of an application. Moreover, when the customized operations are decided, it is also difficult to automate the customized operation data path generation.

# Appendix A

# Accelerator Models

Hardware resource on FPGA mainly includes DSP, LUT, FF and BRAM (block RAM). LUT, FF and DSP consumption can be roughly estimated with a linear function of SCGRA size and can be calculated using (A.1). BRAM consumption $ResConsumption(\boldsymbol{C},1)$ which is slightly different from LUT, FF and DSP consumption can be calculated precisely based on the memory block configurations.

$$ResConsumption(\boldsymbol{C},i) = \alpha_i \times r \times c + \beta_i, (i = 2,3,4) \tag{A.1}$$

Power consumption is consisted of block RAM power, clock power, signal power, DSP power and so on according to the XPower report. The block BRAM power consumption shows very good linearity with the amount of block RAM consumption while the rest part of the power is linear to the SCGRA size. Thus the power consumption can be calculated by (A.2).

$$Power(\boldsymbol{C}) = ResConsumption(\boldsymbol{C},1) \times \alpha_5 + \alpha_6 \times r \times c + \beta_5 \tag{A.2}$$

With the performance and power models, the energy consumption which is the production of power and loop execution time can be obtained using (A.3). Finally, energy efficiency represented by energy delay product which is the production of energy consumption and loop execution time can be calculated using (A.4).

$$Energy(\boldsymbol{C}) = Power(\boldsymbol{C}) \times RunTime(\boldsymbol{C})/f \tag{A.3}$$

$$EDP(\boldsymbol{C}) = Energy(\boldsymbol{C}) \times RunTime(\boldsymbol{C})/f \tag{A.4}$$

# Appendix B

# Accelerator Implementation Analysis

In order to analyze the FPGA resource consumption and power consumption of the SCGRA overlay based FPGA accelerators, three groups of accelerators including SCGRA1, SCGRA2 and SCGRA3 with various configurations as detailed in Table B.1 are implemented on Zedboard. Based on the implementation results, the FPGA resource consumption and power consumption are analyzed respectively in the section.

Table B.1: SCGRA Based FPGA Accelerator Configuration

| Group | Size | Inst. Rom | Data Mem | IBuf /OBuf | Addr Buf |
|---|---|---|---|---|---|
| SCGRA1 | 2x2, 3x2, 3x3, 4x3, 4x4, 5x4 | 1kx72 | 256x32 | 2kx32 | 4kx16 |
| SCGRA2 | 2x2, 3x2, 3x3, 4x3, 4x4 | 2kx72 | 256x32 | 2kx32 | 4kx16 |
| SCGRA3 | 2x2, 3x2, 3x3 | 4kx72 | 256x32 | 2kx32 | 4kx16 |

Figure B.1 shows the relation between the four types of FPGA resource consumption and SCGRA overlay size. It can be found that the consumption of FF, LUT and DSP does not change much with the memory configurations and they present very good linearity to the SCGRA overlay size. Therefore, they can be estimated with linear models. Block RAM consumption depends on both the overlay size and the memory configurations, and single variable linear model will not work

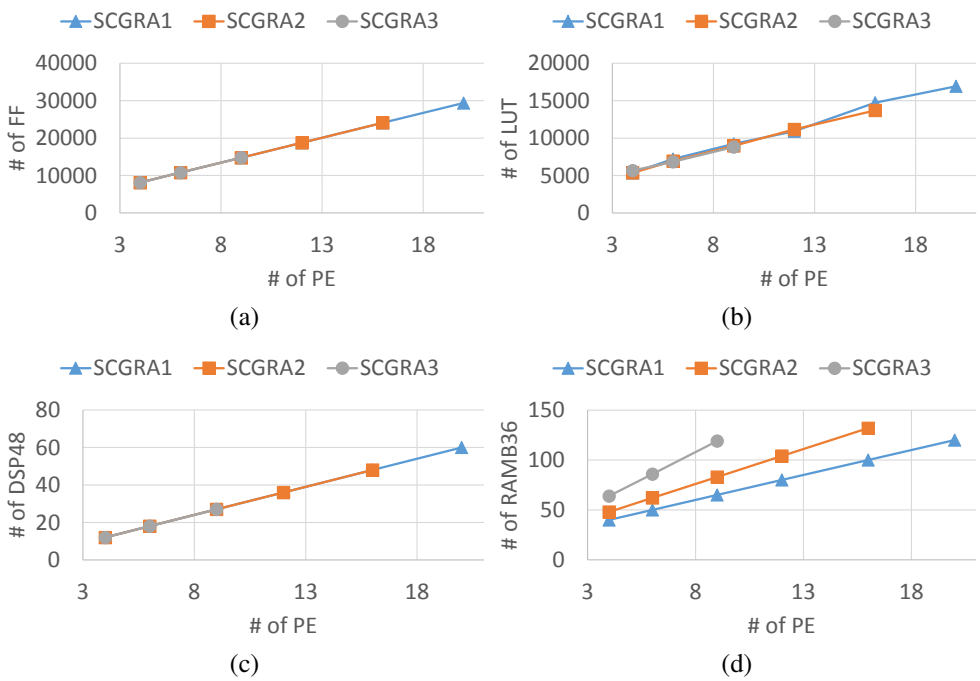



Figure B.1: Relation between The Accelerators' FPGA Resource Consumption and The SCGRA Overlay Size, (a) FF Consumption, (b) LUT Consumption, (c)DSP Consumption, (d)BRAM Consumption

for the estimation. Fortunately, all the memory components in the SCGRA overlay are implemented using primitive block RAMs and they can be calculated precisely with the memory configurations.

According to the power decomposition in Xpower, the power consumption of an FPGA design includes signal power, clock power, BRAM power and so on. To simplify the power model of the SCGRA overlay based FPGA accelerator, the power consumption is divided into BRAM power and base system power which essentially includes the power consumption of the rest part of the system. As shown in Figure B.2, the base system power exhibits good linearity over the SCGRA overlay size while the BRAM power is near linear to the BRAM consumption. As mentioned in previous paragraph, the BRAM consumption can be immediately calculated given the SCGRA overlay configuration. Therefore, both of the two parts of the power consumption can be estimated.

On top of the FPGA resource consumption and power consumption, the implementation frequency of the accelerators is also relatively predictable as shown in

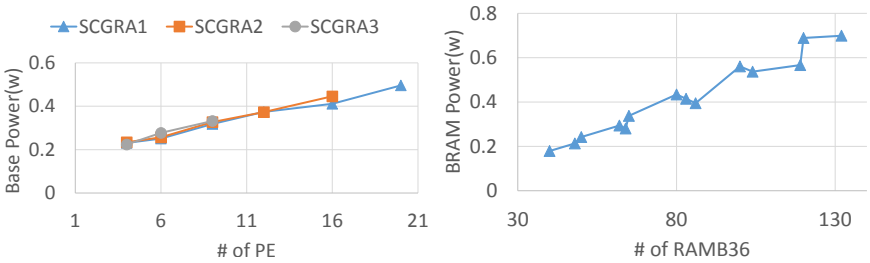



Figure B.2: Power Consumption of the SCGRA Overlay Based FPGA Accelerators, (a) Base System Power Including DSP Power, Clock power, Signal Power, etc., (b) BRAM Power

experiments in Chapter 3. With all the highly predictable implementation features, it is convenient to further estimate the energy consumption and energy efficiency and present insight on the FPGA loop accelerator in early design stage. Although the experiments are still limited to Zedboard, it is believed that similar highly predictable implementation results can be observed on a different platform as the predictability comes from the regularity of the underlying SCGRA overlay. This is one of the advantages adopting SCGRA overlay for FPGA loop accelerator design compared to the design flow using conventional HDL or HLS which rarely guarantees the implementation details due to the extremely complex placing and routing process of the FPGA implementation.

The data transfer between the accelerator and main memory is also very important to the FPGA loop accelerator customization. In this work, the communication latency is modeled based on DMA latency obtained from Zedboard. The FPGA accelerator implemented on programmable logic and the main memory may either communicate through a general purpose (GP) AXI port or a high performance AXI port, and the communication latency per word i.e. 32 bit data is presented in Figure B.3. Despite the two different DMA engines, the basic trend of the communication cost per word is very similar. When the data transfer size is small, DMA communication cost per word is much larger due to the initial DMA transfer cost. When the data transfer size goes up, the initial DMA cost is amortized and communication cost per word is almost a constant. To estimate the data communication

cost, a piece-wise linear function is used to model the DMA transfer.

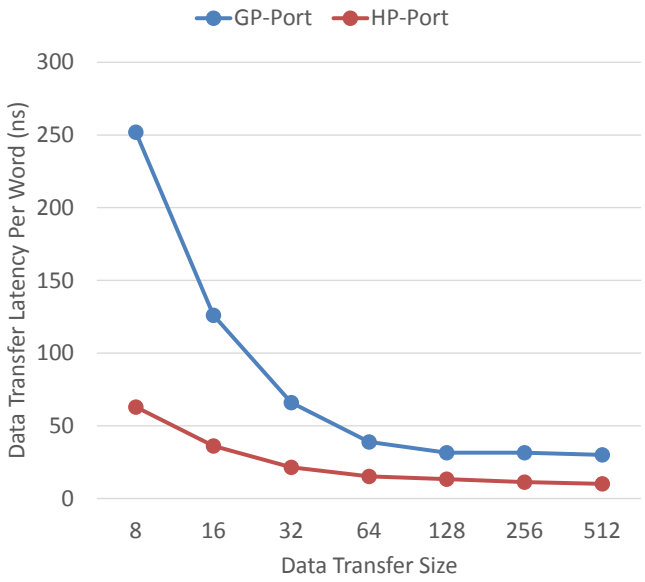



Figure B.3: Zedboard DMA Transfer Latency Per Word

According to the experiment, it can be found that the data transfer between FPGA accelerator and the main memory should be large enough to amortize the initial DMA cost, otherwise the communication cost can be 6X to 8X times larger which dramatically decreases the overall performance speedup achieved by the FPGA accelerator. This is also one of the major reasons that additional loop grouping strategy is adopted in QuickDough.

# Bibliography


[1] Ameer Abdelhadi and Guy GF Lemieux. Modular multi-ported sram-based memories. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 35–44. ACM, 2014.

[2] Jason Agron. Domain-specific language for hw/sw co-design for fpgas. In *Domain-Specific Languages*, pages 262–284. Springer, 2009.

[3] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and Weng-Fai Wong. Guppy: A GPU-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60, Dec 2012. doi: 10.1109/FPT.2012.6412112.

[4] Altera. Nios embedded processor. `http://www.altera.com/products/ip/processors/nios/nio-index.html`, 2014. [Online; accessed 25-June-2014].

[5] Altera. Quartus II Handbook Volume 3: Verification. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts_qii5v3.pdf`, 2015. [Online; accessed 31-October-2015].

[6] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving programming model abstractions for reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):34–44, 2008.

[7] Fakhar Anjam, Muhammad Nadeem, and Stephan Wong. A vliw softcore processor with dynamically adjustable issue-slots. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 393–398. IEEE, 2010.

[8] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 126–131. IEEE, 2009.

[9] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, pages 151–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-542-4. doi: 10.1145/774789.774820. URL `http://doi.acm.org/10.1145/774789.774820`.

[10] C. Beckhoff, D. Koch, and J. Torresen. The Xilinx design language (XDL): Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8. IEEE, 2011.

[11] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 37–44. IEEE, 2012.

[12] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 101–111, 2007. doi:

10.1145/1229428.1229446. URL http://doi.acm.org/10.1145/1229428.1229446.

[13] Srinivas Boppu, Frank Hannig, and Jürgen Teich. Compact code generation for tightly-coupled processor arrays. *Journal of Signal Processing Systems*, 77(1-2):5–29, 2014.

[14] Dimitris Bouris, Antonis Nikitakis, and Ioannis Papaefstathiou. Fast and efficient fpga-based feature detection employing the surf algorithm. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 3–10. IEEE, 2010.

[15] A. Brant and G.G.F. Lemieux. ZUMA: An open FPGA overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96, April 2012. doi: 10.1109/FCCM.2012.25.

[16] Piotr Buciak and Jakub Botwicz. Lightweight multi-threaded network processor core in fpga. In *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS'07. IEEE*, pages 1–5. IEEE, 2007.

[17] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. doi: 10.1145/1950413.1950423. URL http://doi.acm.org/10.1145/1950413.1950423.

[18] D. Capalija and T.S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic*

*and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013. doi: 10.1109/FPL.2013.6645515.

[19] João MP Cardoso and Pedro C Diniz. *Compilation techniques for reconfigurable architectures*, volume 81. Springer Science & Business Media, 2011.

[20] B. Carrion Schafer and K. Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *Computers Digital Techniques, IET*, 6(3):153–159, May 2012. ISSN 1751-8601. doi: 10.1049/iet-cdt.2011.0115.

[21] Anupam Chattopadhyay. Ingredients of adaptability: A survey of reconfigurable processors. *VLSI Design*, 2013:10, 2013.

[22] Anupam Chattopadhyay, B Geukes, David Kammler, Ernst Martin Witte, Oliver Schliebusch, Harold Ishebabi, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Automatic adl-based operand isolation for embedded processors. In *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, volume 1, pages 1–6. IEEE, 2006.

[23] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107. IEEE, 2008.

[24] Hui Yan Cheah, S.A. Fahmy, and D.L. Maskell. iDEA: A DSP block based FPGA soft processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 151–158, Dec 2012. doi: 10.1109/FPT.2012.6412128.

[25] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Xpilot: A platform-based behavioral synthesis system. *SRC TechCon*, 5, 2005.

[26] Eric S Chung, James C Hoe, and Ken Mai. Coram: an in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 97–106. ACM, 2011.

[27] COBHAM. Leon sparc. `http://www.gaisler.com/`, 2015. [Online; accessed 9-Dec-2015].

[28] K. Compton and S. Hauck. Totem: Custom reconfigurable array generation. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 111–119, March 2001.

[29] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[30] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30 (4):473–491, 2011.

[31] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22, Oct 2010.

[32] James Coole and Greg Stitt. Adjustable-cost overlays for runtime compilation. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 21–24, May 2015. doi: 10.1109/FCCM.2015.49.

[33] Altera Corporation. Quartus II 14.0 handbook. `https://www.altera.com/en_US/pdfs/literature/hb/qts/quartusii_handbook.pdf`, 2015. [Online; accessed 18-March-2015].

[34] Xilinx Corporation. Command line tools user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf, 2015. [Online; accessed 18-March-2015].

[35] Robert Dimond, Oskar Mencer, and Wayne Luk. Custard-a customisable threaded fpga soft processor and tools. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 1–6. IEEE, 2005.

[36] R. Ferreira, J.G. Vendramini, L. Mucida, M.M. Pereira, and L. Carro. An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 195–204. ACM, 2011.

[37] Kermin Fleming, Hsin-Jung Yang, Michael Adler, and Joel Emer. The leap fpga operating system. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[38] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson. PATIS: Using partial configuration to improve static FPGA design productivity. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, april 2010. doi: 10.1109/IPDPSW.2010.5470755.

[39] Frank Hannig and Dirk Koch and Daniel Ziener. First International Workshop on FPGAs for Software Programmers. https://www12.informatik.uni-erlangen.de/ws/fsp2014/, 2014. [Online; accessed 31-October-2015].

[40] Carlo Galuzzi and Koen Bertels. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(2):18, 2011.

[41] Jeffrey Goeders and Steven JE Wilton. Effective fpga debug for high-level synthesis generated circuits. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[42] Jeffrey B Goeders, Guy GF Lemieux, and Steven JE Wilton. Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 41–48. IEEE, 2011.

[43] David Grant, Chris Wang, and Guy G.F. Lemieux. A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 123–132, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. doi: 10.1145/1950413.1950441. URL `http://doi.acm.org/10.1145/1950413.1950441`.

[44] Matthias Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI journal*, 38(2):131–183, 2004.

[45] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler co-design approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133, 2014.

[46] Hayden So and John Wawrzynek. First Workshop on Overlay Architectures for FPGAs. `http://olaf.eecs.berkeley.edu/`, 2015. [Online; accessed 31-October-2015].

[47] M. Holzer, B. Knerr, and M. Rupp. Design space exploration with evolutionary multi-objective optimisation. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 126–133, July 2007. doi: 10.1109/SIES.2007.4297326.

[48] Edson L Horta, John W Lockwood, David E Taylor, and David Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfiguration. In *Proceedings of the 39th annual Design Automation Conference*, pages 343–348. ACM, 2002.

[49] Eddie Hung and Steven JE Wilton. Accelerating fpga debug: Increasing visibility using a runtime reconfigurable observation and triggering network. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 19(2):14, 2014.

[50] Eddie Hung and Steven JE Wilton. Incremental trace-buffer insertion for fpga debug. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 22(4):850–863, 2014.

[51] Makiko Itoh, Shigeaki Higaki, Yoshinori Takeuchi, Akira Kitajima, Masaharu Imai, Jun Sato, and Akichika Shiomi. Peas-iii: an asip design environment. In *iccd*, page 430. IEEE, 2000.

[52] Abhishek Kumar Jain, Suhaib A. Fahmy, and Douglas L. Maskell. Efficient overlay architecture based on DSP blocks. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 25–28, May 2015. doi: 10.1109/FCCM.2015.15.

[53] Cindy Kao. Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005.

[54] Nachiket Kapre, Bibin Chandrashekaran, Harnhua Ng, and Kirvy Teo. Driving timing convergence of FPGA designs through machine learning and cloud computing. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*, pages 119–126, 2015. doi: 10.1109/FCCM.2015.36. URL `http://dx.doi.org/10.1109/FCCM.2015.36`.

[55] Khronos. Opencl. https://www.khronos.org/opencl, 2015. [Online; accessed 9-May-2015].

[56] SangDon Kim, SeongMo Lee, SangMuk Lee, JiHoon Jang, Jae-Gi Son, YoungHwan Kim, and SeungEun Lee. Compression accelerator for hadoop appliance. In RobertC.-H. Hsu and Shangguang Wang, editors, *Internet of Vehicles - Technologies and Services*, volume 8662, pages 416–423. Springer International Publishing, 2014.

[57] Jeffrey Kingyens and J. Gregory Steffan. The potential for a GPU-Like overlay architecture for FPGAs. *Int. J. Reconfig. Comp.*, 2011, 2011.

[58] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A dynamically reconfigurable weakly programmable processor array architecture template. In *ReCoSoC*, pages 31–37, 2006.

[59] D. Koch, C. Beckhoff, and G.G.F. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013. doi: 10.1109/FPL.2013.6645517.

[60] Sebastian Korf, Dario Cozzi, Markus Koester, Jens Hagemeyer, Mario Porrmann, U Ruckert, and Marco D Santambrogio. Automatic HDL-based generation of homogeneous hard macros for FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 125–132. IEEE, 2011.

[61] Maciej Kurek, Tobias Becker, Thomas C.P. Chau, and Wayne Luk. Automating optimization of reconfigurable designs. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 210–213, May 2014. doi: 10.1109/FCCM.2014.65.

[62] Charles Eric LaForest and John Gregory Steffan. OCTAVO: An FPGA-centric processor family. In *Proceedings of the ACM/SIGDA International*

*Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 219–228, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1155-7. doi: 10.1145/2145694.2145731. URL http://doi.acm.org/10.1145/2145694.2145731.

[63] C. Lavin, M. Padilla, S. Ghosh, B. Nelson, B. Hutchings, and M. Wirthlin. Using hard macros to reduce FPGA compilation time. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 438–441. IEEE, 2010.

[64] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. HMFlow: Accelerating FPGA compilation with hard macros for rapid prototyping. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 117 –124, may 2011. doi: 10.1109/FCCM.2011.17.

[65] Christopher Lavin, Brent Nelson, and Brad Hutchings. Improving clock-rate of hard-macro designs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 246–253. IEEE, 2013.

[66] Cristina Lavin, Brent Nelson, and Brad Hutchings. Impact of hard macro size on fpga clock rate and place/route time. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6. IEEE, 2013.

[67] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. MARC: A many-core approach to reconfigurable computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7–12, Dec 2010. doi: 10.1109/ReConFig.2010.49.

[68] Colin Yu Lin and Hayden Kwok-Hay So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture syn-

thesis. *SIGARCH Comput. Archit. News*, 40(5):58–63, March 2012. ISSN 0163-5964. doi: 10.1145/2460216.2460227. URL http://doi.acm.org/10.1145/2460216.2460227.

[69] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. Automatic nested loop acceleration on fpgas using soft CGRA overlay. *CoRR*, abs/1509.00042, 2015. URL http://arxiv.org/abs/1509.00042.

[70] Hung-Yi Liu and L.P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–7, May 2013.

[71] Zhen Liu, Kai Zheng, and Bin Liu. Fpga implementation of hierarchical memory architecture for network processors. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 295–298. IEEE, 2004.

[72] LLVM. The LLVM compiler framework. http://llvm.org, 2013. [Online; accessed 19-September-2013].

[73] Andrea Lodi, Mario Toma, and Fabio Campi. A pipelined configurable gate array for embedded processors. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 21–30. ACM, 2003.

[74] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, October 2009. ISSN 1539-9087. doi: 10.1145/1596532.1596540. URL http://doi.acm.org/10.1145/1596532.1596540.

[75] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 271–271, New York, NY, USA, 2005.

ACM. ISBN 1-59593-029-9. doi: 10.1145/1046192.1046247. URL http://doi.acm.org/10.1145/1046192.1046247.

[76] Pongstorn Maidee, Cristinel Ababei, and Kia Bazargan. Fast timing-driven partitioning-based placement for island style fpgas. In *Proceedings of the 40th annual design automation conference*, pages 598–603. ACM, 2003.

[77] T. Majumder, P.P. Pande, and A. Kalyanaraman. Hardware accelerators in computational biology: Application, potential, and challenges. *Design Test, IEEE*, 31(1):8–18, Feb 2014. ISSN 2168-2356. doi: 10.1109/MDAT.2013.2290118.

[78] MathWorks. Matlab Solutions: FPGA Design and SoC Design. http://www.mathworks.com/solutions/fpga-design/, 2015. [Online; accessed 31-October-2015].

[79] Mentor Graphics. Certus Silicon Debug. https://www.mentor.com/products/fv/certus-silicon-debug, 2015. [Online; accessed 31-October-2015].

[80] Mentor Graphics Corporation. Handel-C. https://www.mentor.com/products/fpga/handel-c/, 2015. [Online; accessed 31-October-2015].

[81] Narasinga Rao Miniskar, Soma Kohli, Haewoo Park, and Donghoon Yoo. Retargetable automatic generation of compound instructions for CGRA based reconfigurable processor applications. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*, pages 1–9. IEEE, 2014.

[82] Yehdhih Ould Mohammed Moctar and Philip Brisk. Parallel fpga routing based on the operator formulation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[83] Chandra Mulpuri and Scott Hauck. Runtime and quality tradeoffs in fpga placement and routing. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 29–36. ACM, 2001.

[84] MyHDL. MyHDL. `http://www.myhdl.org/`, 2015. [Online; accessed 31-October-2015].

[85] Zdravko Panjkov, Andreas Wasserbauer, Timm Ostermann, and Richard Hagelauer. Hybrid fpga debug approach. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.

[86] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014. doi: 10.1109/ISCA.2014.6853195.

[87] Salil Raje. Extending the power of fpgas to software developers. `http://fpl2015.org/pdf/keynotes/1.pdf`, 2015. [Online; accessed 1-November-2015].

[88] ROCCC. ROCCC2.0. `http://www.jacquardcomputing.com/roccc/`, 2014. [Online; accessed 19-January-2014].

[89] Yaska Sankar and Jonathan Rose. Trading quality for compile time: Ultra-fast placement for fpgas. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 157–166. ACM, 1999.

[90] Souradip Sarkar, Turbo Majumder, Ananth Kalyanaraman, and Partha Pratim Pande. Hardware accelerators for biocomputing: A survey. In *IEEE International Symposium on Circuits and Systems*, pages 3789–3792, 2010. doi: 10.1109/ISCAS.2010.5537736.

[91] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. Divide and conquer high-level synthesis design space exploration. *ACM Trans. Des. Autom. Electron. Syst.*, 17(3):29:1–29:19, July 2012. ISSN 1084-4309. doi: 10.1145/2209291.2209302. URL `http://doi.acm.org/10.1145/2209291.2209302`.

[92] JMJ Schutten. List scheduling revisited. *Operations Research Letters*, 18(4): 167–170, 1996.

[93] A. Severance and G. Lemieux. VENICE: A compact vector processor for FPGA applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 261–268, Dec 2012. doi: 10.1109/FPT.2012.6412146.

[94] S. Shukla, N.W. Bergmann, and J. Becker. QUKU: a two-level reconfigurable architecture. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, March 2006. doi: 10.1109/ISVLSI.2006.76.

[95] Iouliia Skliarova and Antonio De Brito Ferrari. Reconfigurable Hardware SAT Solvers: A Survey of Systems. *IEEE Transactions on Computers*, 53: 1449–1461, 2004. doi: 10.1109/TC.2004.102.

[96] Hayden Kwok-Hay So and Robert Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *Transactions on Embedded Computing Systems*, 7(2):1–28, 2008. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1331331.1331338.

[97] S Sukhsawas and Khaled Benkrid. A high-level implementation of a high performance pipeline fft on virtex-e fpgas. In *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pages 229–232. IEEE, 2004.

[98] Russell Tessier. Fast placement approaches for fpgas. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(2):284–305, 2002.

[99] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1-2):7–27, 2001.

[100] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fpgas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72. ACM, 2009.

[101] Xiang Tian, Khaled Benkrid, and Xiaochen Gu. High performance monte-carlo based option pricing on fpgas. *Engineering Letters*, 16(3):434–442, 2008.

[102] Tobias Becker and Frank Hannig and Dirk Koch and Daniel Ziener and Friedrich-Alexander. Second International Workshop on FPGAs for Software Programmers. `https://www12.informatik.uni-erlangen.de/ws/fsp2015/`, 2015. [Online; accessed 31-October-2015].

[103] TOP500. Top500 list june 2015. `http://top500.org/lists/2015/06`, June 2015. [Online; accessed 18-October-2015].

[104] Michael G Wrighton and André M DeHon. Hardware-assisted simulated annealing with application for fast fpga placement. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 33–42. ACM, 2003.

[105] Guiming Wu, Yong Dou, Yuanwu Lei, Jie Zhou, Miao Wang, and Jingfei Jiang. A fine-grained pipelined implementation of the linpack benchmark on fpgas. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 183–190. IEEE, 2009.

[106] Xilinx. data2mem. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf`, 2012. [Online; accessed 19-September-2012].

[107] Xilinx. Vivado HLS. `http://www.xilinx.com/products/design-tools/vivado/`, 2014. [Online; accessed 18-October-2014].

[108] Xilinx. MicroBlaze soft processor core. `http://www.xilinx.com/tools/microblaze.htm`, 2014. [Online; accessed 25-June-2014].

[109] Xilinx. ChipScope Pro and the Serial I/O Toolkit. `http://www.xilinx.com/tools/cspro.htm`, 2015. [Online; accessed 31-October-2015].

[110] Xilinx. Virtex ultrascale+ fpgas. `http://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html`, 2015. [Online; accessed 1-November-2015].

[111] Que Yanghua, Chinnakkannu Adaikkala Raj, Harnhua Ng, Kirvy Teo, and Nachiket Kapre. Case for design-specific machine learning in timing closure of FPGA designs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, pages 169–172, 2016. doi: 10.1145/2847263.2847336. URL `http://doi.acm.org/10.1145/2847263.2847336`.

[112] P. Yiannacouras, J.G. Steffan, and J. Rose. Exploration and customization of FPGA-Based soft processors. *Computer-Aided Design of Integrated Circuits*

*and Systems, IEEE Transactions on*, 26(2):266–277, Feb 2007. ISSN 0278-0070. doi: 10.1109/TCAD.2006.887921.

[113] Peter Yiannacouras, Jonathan Rose, and J Gregory Steffan. The microarchitecture of fpga-based soft processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 202–212. ACM, 2005.

[114] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain performance scaling of soft vector processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 97–106, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-626-7. doi: 10.1145/1629395.1629411. URL `http://doi.acm.org/10.1145/1629395.1629411`.

[115] M.X. Yue, D. Koch, and G.G.F. Lemieux. Rapid overlay builder for xilinx fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 17–20, May 2015. doi: 10.1109/FCCM.2015.48.

[116] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.

[117] Guanwen Zhong, V. Venkataramani, Yun Liang, T. Mitra, and S. Niar. Design space exploration of multiple loops on fpgas using high level synthesis. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 456–463, Oct 2014. doi: 10.1109/ICCD.2014.6974719.

[118] Li Zhou, Dongpei Liu, Jianfeng Zhang, and Hengzhu Liu. Application-specific coarse-grained reconfigurable array: architecture and design methodology. *International Journal of Electronics*, (ahead-of-print):1–14, 2014.