# Automated Space/Time Scaling of Streaming Task Graphs on Field-Programmable Gate Arrays

by

Hossein Omidian Savarbaghi

B.Sc. in Computer Engineering, Isfahan University of Technology, 2007

M.Sc. in Computer Engineering, Science and Research Branch of IAU, 2011

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

**Doctor of Philosophy**

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia

(Vancouver)

October 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the dissertation entitled:

Automated Space/Time Scaling of Streaming Task Graphs on Field-Programmable Gate Arrays

submitted by Hossein Omidian Savarbaghi in partial fulfillment of the requirements for the degree of Doctor of Philosophy
in Electrical and Computer Engineering

**Examining Committee:**

Guy Lemieux, Electrical and Computer Engineering
Supervisor

Prof. Mieszko Lis, Electrical and Computer Engineering
Supervisory Committee Member

Prof. Andre Ivanov, Electrical and Computer Engineering
University Examiner

Prof. Alan Hu, Computer Science
University Examiner

Prof. Russell Tessier, Electrical and Computer Engineering
External Examiner

# Abstract

Parallel computing platforms provide good performance for streaming applications within a limited power budget. However, these platforms can be difficult to program. Moreover, when the size of the computing platform target changes, users must manually reallocate resources and parallelism. This thesis provides a framework to retarget applications described by a Streaming Task Graph (STG) for implementation on different platforms, where the framework can automatically scale the solution size to fit available resource or performance targets.

First, we explore automated space/time scaling for STGs targeting a pipelined coarse-grained architecture. We produce a tool that analyzes the degrees of parallelism in a general stream application and finds possible bottlenecks. We introduce possible optimization strategies for STGs, and two algorithmic approaches: a classical approach based upon Integer Linear Programming (ILP), and a novel heuristic approach which introduces a new optimization and produces better results (using 30% less area) with a shorter run-time.

Second, we explore automated space/time scaling for STGs targeting a fine-grained architecture (Field-Programmable Gate Array (FPGA)). We propose a framework on top of a commercial High-Level Synthesis (HLS) tool which adds the ability to automatically meet a defined area budget or target throughput. Within the framework, we use similar ILP and heuristic approaches. The heuristic automatically achieves over 95% of the target area budget on average while improving throughput over the ILP. It can also meet the same throughput targets as the ILP while saving 19% area.

Third, we investigate automated space/time scaling of STGs in a hybrid architecture consisting of a Soft Vector Processor (SVP) and select custom instructions.

To achieve higher performance, we investigate using dynamic Partial Reconfiguration (PR) by time-sharing the FPGA resources. The performance results achieve speedups far beyond what a plain SVP can accomplish: an 8-lane SVP achieves a speedup of 5.3 on the Canny-blur application, whereas the PR version is another 3.5 times faster, with a net speedup of 18.5 over the ARM Cortex A9 processor. By increasing the dynamic PR rate beyond what is available today, we also show that a further 5.7 times speedup can be achieved (105.9x speedup versus the Cortex A9).

# Lay Summary

Software applications are often computationally intensive. To maximize the use of resources on a range of hardware platforms with a differing amount of parallel resources and minimize the runtime of software applications as much as possible, engineers need to manually modify and optimize each application for each platform with different resource restrictions or desired performance targets. This process is time consuming and can be prone to error. We propose an approach to automatically explore different ways of implementing an application for a performance target or a resource restriction defined by users.

# Preface

The following publications have been adapted for inclusion in the dissertation:

- **Automated Space/Time Scaling of Streaming Task Graph** [66]

  Published in the 2016 International Workshop on Overlay Architectures for FPGA (OLAF 2016). Authored by Hossein Omidian and Guy Lemieux. Appears in chapter 3.

  For this paper, I performed all of the deign methodology and implementation of compiler and simulator. I also did the benchmarking, simulating and evaluation. I also formulated the ILP optimization and proposed and implemented the heuristic optimization approach presented in this paper. Guy Lemieux served in an advisory fashion.

- **Exploring Automated Space/Time Tradeoffs for OpenVX Compute Graphs** [67].

  Published in the 2017 International Conference on Field-Programmable Technology (FPT 2017). Authored by Hossein Omidian and Guy Lemieux. Appears in chapter 4.

  I performed all the design methodology, implementation and benchmarking as well as all the simulation and evaluation for this paper. Guy Lemieux served in an advisory fashion.

- **JANUS: A Compilation System for Balancing Parallelism and Performance in OpenVX** [65].

  Presented in the 2018 International Conference on Machine Vision and Information Technology (CMVIT 2018) and published in Journal of Physics:

Conference Series. Authored by Hossein Omidian and Guy Lemieux. Appears in chapter 3 and chapter 4.

I performed all the implementation and benchmarking for this paper. Guy Lemieux served in an advisory fashion.

- **An Accelerated OpenVX Overlay for Pure Software Programmers**

  Published as a shot paper (poster) in the 2018 International Conference on Field-Programmable Technology (FPT 2018). Authored by Hossein Omidian, Nick Ivanov and Guy Lemieux. Appears in chapter 5.

  I performed all the design methodology, implementation and benchmarking as well as all the simulation and evaluation for this paper. Nick Ivanov helped implementing runtime results for SVP and ARM. Guy Lemieux served in an advisory fashion.

# Table of Contents

# List of Tables

# List of Figures

# Glossary

**ALU** Arithmetic Logic Unit

**ASIC** Application-Specific Integrated Circuit

**BRAM** Block RAM

**CAD** Computer Aided Design

**CLB** Configurable Logic Block

**CPU** Central Processing Unit

**CV** Computer Vision

**DAG** Directed Acyclic Graph

**DIG** Different Implementation Generator

**DMA** Direct Memory Access

**DSP** Digital Signal Processing

**FIFO** First In, First Out

**FPGA** Field-Programmable Gate Array

**GLPK** GNU Linear Programming Kit

**HDL** Hardware Description Language

**HLL** High-Level Language

**HLS** High-Level Synthesis

**HPC** High Performance Computing

**IC** Integrated Circuit

**ICAP** Internal Configuration Access Port

**ILP** Integer Linear Programming

**JPEG** Joint Photographic Experts Group

**KPN** Kahn Processing Network

**LUT** Look-Up Table

**MIMD** Multiple Instruction, Multiple Data

**MPPA** Massively Parallel Processor Array

**MUX** Multiplexer

**MXP** Matrix Processor

**NOC** Network on Chip

**PE** Processing Element

**PR** Partial Reconfiguration

**PRR** Partial Reconfiguration Region

**RAM** Random Access Memory

**RTL** Register Transfer Level

**SDA** Stream Data Adjuster

**SDFG** Synchronous Data Flow Graphs

**SDK** Software Development Kit

**SIMD** Single Instruction, Multiple Data

**SRAM** Static RAM

**STG** Streaming Task Graph

**SVP** Soft Vector Processor

**VCI** Vector Custom Instruction

**VLIW** Very Long Instruction Word

# Acknowledgments

Baba, thanks for all the support. You taught me how to think differently. Maman, thanks for your unconditional love and helping me through the tough times. Thanks grandpa for inspiring me even after you left us, you always wanted me to improve myself.

Guy, thank you for everything you've done for me. You literally changed my life. I cannot thank you enough. Thanks Kia Bazargan for all the support during my Masters. You are the reason I started my PhD, you are my hero. Thank you Jayme Carvey for everything. You helped me getting my speech confidence back. Aaron, Thanks for helping me starting my research.

Thanks SOC faculties and members. It was a pleasure working with everyone in the SOC lab.

Thanks to NSERC for funding.

# Chapter 1

# Introduction

## 1.1 Motivation

Applications that are structured around the notion of a "stream" are increasingly important and widespread. One study shows that streaming applications are already consuming most of the cycles on consumer machines, and their use is continuing to grow [71]. Many of these are media and vision applications, and most of them are computationally intensive. There is no doubt that increasingly more complex streaming applications will continue to be introduced, so the demand for higher performance will continue.

Increasing the clock frequency was the simple and traditional way to achieve high performance computing, however, since clock frequency scaling has essentially stopped due to power constraints, an issue known as Dennard scaling [25], computer designers and architects have focused on delivering increased levels of parallelism to improve both performance and performance-per-Watt [68]. Several different approaches have been introduced to address this issue. At one end is coarse-grained parallelism, often implemented through multi-core processors, usually through a high-level language. At the other end is fine-grained parallelism, often implemented through Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) devices by designing hardware-level solutions.

Unfortunately, both coarse-grained and fine-grained parallelism can be chal-

lenging to program. In software, it can be challenging to expose sufficient parallelism in common languages like C, and difficult to describe some types of computation in CUDA or OpenCL, for example. In hardware, Register Transfer Level (RTL) languages such as VHDL and Verilog achieve very good results, but it is not accessible to users without hardware design knowledge. RTL is also very tedious to program, as it requires describing everything on a cycle-by-cycle basis. This makes the above-mentioned approaches either inaccessible to general users or time consuming and hard to implement for more expert users.

In FPGAs, design is made easier through High-Level Synthesis (HLS) tools. These are tools that convert a High-Level Language (HLL) (e.g., Java or C) into RTL (e.g., an Hardware Description Language (HDL) such as Verilog or VHDL). However, they impose their own challenging constraints in writing the HLL and in what can be parallelized [23, 62]. Several industrial and academic HLS tools have been developed to provide an environment for users to describe their application in a HLL such as C/C++ to avoid the difficulty of HDL programming. However, current HLS tools require the user to explicitly manage resources at every stage in their algorithm in order to meet a specified area target or throughput target. This is done by manual control of things such as loop unrolling or pipelining by control pragmas to the code to change how it is synthesized. Moreover, in order to fit a design to different FPGA sizes or achieve a different throughput, users need to alter the design manually, which involves changing the pragmas at best, but in other cases may require more significant rewriting of the source code. Instead, users would benefit from an HLS tool that can automatically investigate different degrees of parallelism, explore space/time tradeoffs and find a suitable implementation for either a throughput target or an area budget.

As another example, coarse-grained streaming architectures such as Ambric's Massively Parallel Processor Array (MPPA) device [13] also require programmming in a HLL. In Ambric's case, Java is used to describe each thread and to explicitly allocate instances, where each thread has streaming input and output ports, and the Astruct custom language is used to connect these ports together [12]. The compiler creates a graph of communicating object-threads which must be placed and routed, much like an FPGA. This can be a great platform for low-power, high-performance computing [40]. However, the Astruct language design does not al-

low for the creation of dynamic connections; these must be known explicitly at compile-time. Although using object-oriented Java helps make this platform accessible to general users, to achieve high performance, users still need to manually do the resource management by partitioning the application into small atomic objects. This requires the user to learn the Ambric architecture and parallelism methods. Moreover, users need to go through the same process again if they want to increase performance, shrink the size, or target another MPPA with a different architecture and resources.

The approaches described above are specialized for getting high performance from specific hardware targets. In both cases, however, when the number of available resources changes, or the target changes, users need to change the source code of the implementation to satisfy the new constraints. For example, let us go through four different scenarios which need changing these targets. One scenario can be implementing an application on a different platform with different computing resources (from a small embedded device to a big data center). In this case, the user needs to fill up different chips (different area budgets) while maximizing the throughput. A second scenario is when the user has purchased a pre-made design, wants to change/customize one block within the whole design, and there is a limited amount of leftover area available. This needs an area budget. A third scenario can be with a big team of hardware and software developers, where the hardware developers assign a budget (throughput or area) to the software developers to implement their algorithm (e.g., part of a floorplanning process for a whole team). A fourth scenario is when a user wishes to put multiple applications onto a chip, and is trying to pack as many applications (or application instances) as possible; this needs setting throughput targets or area budgets for each instance, but the overall goal is to achieve maximum throughput for the entire chip. A final use-case is that a user may wish to develop several different products or solutions at different price (and performance) points; there is a need for a tools that allow them to more easily scale it to fit different chips or budgets.

This process of retargetting and scaling an application in all the scenarios mentioned above can be time consuming and expensive, and in many cases programmers need extensive skill as well as knowledge of the target platform. There is a need to have general, scalable and flexible approach which allows users to describe

streaming applications in a retargetable way.

Streaming applications can be described by a Streaming Task Graph (STG) which are collections of tasks with dependencies between their inputs and outputs. This means they can leverage pipelined architectures. This work focuses on STGs with Directed Acyclic Graph (DAG) topologies. These type of graphs can easily be manipulated to adopt different methods such as pipelining, node replicating and node combining in order to find different degrees of parallelism. This makes an STG suitable model for exploring space/time implementation tradeoffs on pipelined architectures. The scope of automatically exploring space/time tradeoffs for STGs has motivated a considerable amount of research to improve the usability of parallel resources in different pipelined architectures.

The goal of this dissertation is to broaden the overall usability of parallel resources by providing an environment which allows users to automatically explore space/time tradeoffs and find suitable implementations regarding a throughput target or an area budget on a wide range of different pipelined architectures. This can be added to a HLL or to the HLS process to make them more flexible, scalable and more area efficient. Moreover this makes a broad range of pipelined architectures with different granularity and resources accessible to people with limited specialized knowledge. They can easily define an application as a STG in an HLL instead of dealing with using low level RTL or parallel programming languages such as OpenCL. At the same time, we wish to ensure that, by using this approach, the HLS approach is able to automatically find a broad range of different solutions which can each compete with manually optimized implementations. Moreover, the HLS tools need to automatically explore the different solutions and find a suitable implementation based on different defined restrictions or desired targets.

## 1.2 Approach

As mentioned above, the main goal of this dissertation is to broaden the overall usability of parallel resources in different pipelined architectures by making scaling an implementation easier for general users. This can be done by providing an environment which allows users to define applications as STGs, explore the space/time design space, and find a suitable solution for a defined restriction or a desired tar-

get. In other words, we wish to address the automated space/time scaling problem for STGs on different pipelined architectures. We start with addressing this problem targeting a coarse-grained architecture (an MPPA), then target a fine-grained architecture (an FPGA), and finally use the experience learned to target a hybrid architecture with both coarse and fine components. Below, we discuss our approach in greater detail.

### 1.2.1 Experimental Architecture Models

Our experiments into automated space/time scaling for STGs examine three different parallel architecture models.

First, we examine pipelined coarse-grained parallelism modelled after the Ambric MPPA. Like Ambric, we develop a Java-based compiler tool chain targeting the MPPA. The compiler attempts to find the maximum degrees of parallelism in the stream application, then performs throughput analysis, throughput propagation and looks for possible bottlenecks. We study different STG manipulations to find different implementations in the space/time tradeoff space. We show that a classical Integer Linear Programming (ILP) optimization strategy, based upon prior work, finds a locally optimal design point subject to the given area or throughput target. In addition, we introduce a heuristic approach that runs faster and achieves better results than the ILP approach because it has fewer restrictions. We discuss our solutions to this coarse-grained model in detail in chapter 3.

Second, we examine pipelined fine-grained parallelism available on a Xilinx FPGA. We investigate automating the ability to make space/time tradeoffs in the commercial HLS tool, Xilinx Vivado HLS [43, 96], targeting the FPGA fabric. Since we do not have access to the Xilinx Vivado HLS source code, we propose a framework on top of it which evaluates many ways pragmas can be used to parallelize a STG. To represent STGs, we restrict ourselves to the OpenVX standard which enables the creation of Computer Vision (CV) applications as compute graphs. Our OpenVX HLS system uses heavily parameterized CV kernels as well as multiple optimization approaches to automatically expand and explore the space/time design space. Ultimately, our system finds a suitable fine-grained implementation for a given area or throughput target. We compare both the classical ILP and novel

heuristic optimization approaches, and compare these to manually written implementations. We discuss this model in detail in chapter 4.

Third, we examine a hybrid approach that uses both coarse-grained and fine-grained parallelism. For coarse-grained parallelism, we target a Soft Vector Processor (SVP) architecture implemented in the Xilinx FPGA fabric that streams data through a wide Single Instruction, Multiple Data (SIMD) datapath. For fine-grained parallelism, we target custom instructions implemented in the Xilinx FPGA fabric built using the HLS tool developed earlier. Each Vector Custom Instruction (VCI) offers horizontal parallelism up to the same SIMD width as the SVP, as well as vertical parallelism created with pipelining. Furthermore, additional coarse-grain pipeline parallelism can be created by cascading the output of one VCI to another, producing a chain of VCI operations. The entire VCI chain must respect the two-input, one-output restriction of a single VCI, and must be of the same SIMD width. To assemble such a system, we rely upon a feature of FPGAs known as Partial Reconfiguration (PR), where a Partial Reconfiguration Region (PRR) can be reconfigured dynamically at run time. The VCI chain is connected to the SVP using a multiplexer network added to the PRR logic. Using this system, we examine the ability to implement OpenVX compute graphs with space/time tradeoffs on an FPGA device with limited resources. This means our approach decides which part of the graph runs on the SVP and which part runs as a VCI or VCI chain. Similar to previous approaches, we use a heuristic approach to leverage runtime optimization techniques which cannot be done as easily in the classical ILP approach. We discuss this model in detail in chapter 5.

### 1.2.2 Experimental Methodology

For each of the three experimental architectural models, we use a slightly different methodology for building our tools and collecting results.

In the first model, we develop a Java-based compiler platform similar to the Ambric system. This compiler performs optimization and code generation for our target device which is similar to the Ambric MPPA. However, we did not follow precise instruction encodings, and we did not attempt to run the code on a real Ambric device. Instead, we developed a cycle-accurate simulator which measures execu-

tion time. Benchmarks are written in Java and run on this simulator. Although real Ambric devices are no longer available, similar coarse-grained overlays are being developed on FPGAs such as the GRVI Phalanx [34] as well as an Ambric clone. [1]

In the second model, we use the actual Xilinx tools and devices to produce a real bitstream. In particular, the Vivado HLS tool is used to synthesize C for the FPGA. Benchmarks are written in C using the OpenVX API. We develop a front-end tool which synthesizes an FPGA implementation of an entire OpenVX compute graph. This tool is given an overall area or throughput target; it analyzes the full graph, optionally transforms it, and determines the best space/time tradeoff to meet the target. Each node is an OpenVX compute kernel which is written in C and fully annotated with pragmas for maximum parallelism with Vivado HLS. Our tool determines which pragmas are needed for each kernel instance in the graph to meet the overall target. The final output is a C program which is compiled by the Vivado tools into an FPGA bitstream, where the area usage, clock speed, and throughput metrics can be verified. These graphs use AXI-stream inputs and outputs, allowing them to be easily connected to a full system.

In the third model, we continue to use Xilinx tools and devices together with the VectorBlox Matrix Processor (MXP) SVP, but we produce final results using a performance equation. For each OpenVX compute kernel, we use three implementations: scalar ARM Cortex-A9 code produced using regular C with gcc, vectorized code for the ARM and VectorBlox MXP, and a VCI. The scalar code is only used to produce a performance baseline. The VCI implementations are produced by compiling the OpenVX kernels (from the previous model) for varying SIMD widths, up to the width of the SVP, and compiled into the smallest area required. To save time in this work, we did not implement the multiplexer network required to connect VCI chains into the MXP, nor did we implement the PR control logic. Instead, we model the time required to perform PR based upon performance specifications given by Xilinx for their Internal Configuration Access Port (ICAP) on-chip configuration controller. We are particularly interested making the ICAP controller faster, so we model this speed as a variable, and use performance equations to model the speed of the overall system. With this feasibility study, we have confidence that a

[1] Michael Butts, private communication.

working system could be implemented to verify the estimated gains.

## 1.3 Contributions

The contributions of this dissertation are summarized in the following paragraphs.

- **Chapter 3** is a demonstration of the benefits of automated space/time scaling for STGs mapped onto MPPA overlays rather than manually implementing, scaling and optimizing. We introduce an HLS tool that automatically allows exploring area/throughput tradeoffs for STGs. We improve upon the classical ILP approach by introducing a heuristic approach.

  Our approach differs from existing approaches because:

  1. It automatically investigates partitioning and finding different implementations.
  2. It combines module selection and replication methods with node combining and splitting in order to automatically find a better area/throughput tradeoff.
  3. It presents a novel heuristic approach which is more flexible and can find design points not feasible to find with a classical ILP approach.

- **Chapter 4** is a demonstration of our solution for automating the ability to make space/time tradeoffs in a commercial HLS tool. This leads to a C-to-RTL tool which can automatically explore the space/time problem space for implementing OpenVX applications defined as STGs on FPGAs. Based on our knowledge, this cannot be done with existing C-to-RTL tools. The experiment shows the proposed approach is able to achieve the same performance as manually written implementations and also it finds several more solutions for a variety of different throughput targets which leads to a 30% area reduction. The tool is able to achieve over 95% of the target area budget on average while improving the throughput. The Inter-node Optimizer step of our heuristic is able to hit the same throughput targets while reducing the area cost by 19% on average compared to the ILP approach. In terms of efficient use of parallel resources on the chip, the experiment shows the

tool manages to satisfy different throughput targets while using parallel resources efficiently. For example, it achieves up to 5.5 GigaPixel/sec for the Sobel application on a small Xilinx 7Z020 device.

- **Chapter 5** is a demonstration of run-time acceleration using dynamic partial reconfiguration. More specifically, a SVP software system with coarse-grained parallelism is further accelerated using rapid reconfiguration of VCI chains. The experiment shows speedups far beyond what a plain SVP can accomplish. For example, an 8-lane SVP achieves a net speedup of 18 versus the scalar ARM processor for running the $Canny-blur$ application. This was achieved by using automated space/time scaling, node clustering and dynamic PR. However, if FPGA vendors can provide a much faster PR rate, a net speedup of 106 is possible.

  Our approach differs from existing approaches because:

  1. It is the first work to explore PR and time-sharing of VCI for speeding up SVP.
  2. It further improves performance by adding scratchpad bypass with VCI chaining.
  3. It uses pre-synthesized node fusion of common VCI chains to save area.

## 1.4 Dissertation Organization

Chapter 2 presents the background of this dissertation, including the general approach to find different degrees of parallelism in an application and programming models to describe the application. Moreover, it presents the background of different reconfigurable computing platforms we used in this dissertation. Chapter 3 details our implementation of automated space/time scaling of STG on a coarse-grained architecture (MPPA). Chapter 4 details our approach of exploring automated space/time tradeoffs for CV application described as an OpenVX compute graph on a fine-grained architecture (FPGA). Chapter 5 details our implementation of a compilation system which uses run-time reconfiguration to accelerate applications on a hybrid SVP/FPGA architecture.

# Chapter 2

# Background

This chapter provides the necessary background information for this dissertation. First, it presents an overview of finding parallelism in a general program to show different transformations that find different degrees of parallelism in a general program. Next, it describes different programming models and their pros and cons. To give the necessary background to understand different reconfigurable computing platforms used in this dissertation, FPGA, MPPA and SVP are introduced. This leads to discussing OpenVX applications on reconfigurable computing platforms. Finally, it discusses the advantages and limitations of partial reconfiguration.

## 2.1 Finding Parallelism in a General Program

Regardless of what programming model we will use for describing a computational problem, the first step to get better throughput and use parallel resources efficiently is analyzing the program and finding the potential parallelism in it. Getting high performance on a platform with parallel resources requires not only finding parallelism in the program but also minimizing the synchronization overhead because the synchronization process may stall the system; a processing element may have to wait for another processing element to reach the corresponding synchronization point and make data ready. High synchronization frequency generally comes with high levels of data communication between processing elements, which might reduce the performance of the system. As a result, a program with fine-grain syn-

chronization can run on a multi-core system even slower than one processor [56]. It is therefore important to find parallelism that requires minimal synchronization. In other words the final goal is identifying the coarsest granularity of parallelism in a program by finding the largest set of independent computations that can be run by different processing elements in a synchronization-free manner. As mentioned above, it is important to find parallelism before implementing a design. A parallelism finder algorithm analyzes and transforms the program to find all the degrees of parallelism in it.

There have been several studies on program transformation to achieve parallelism and data locality for a program. These transformations are generally limited to loops that use affine functions for representing bounds and array accesses [6].

Below we describe an algorithm that finds the maximum degree of parallelism in a general program with nested loops and affine index expressions for array accesses proposed by Lim et al. [56]. Index expressions are affine if it involves multiplying the loop index variables by constants and adding constants. All the instructions in a program are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loop index values to a partition number. Partition numbers are used for two different purposes, space partitioning and time partitioning. Operations belonging to the same space partition are mapped to the same processing element. On the other hand operations belonging to time partition $i$ should execute before those in partition $i+1$. We try to find a combination of affine space and time partition mappings that maximizes the degree of parallelism with successively greater degree of synchronization. Several transformations are described in [5, 7, 8, 14, 44, 74, 92]. We can achieve all of the loop-level parallelism with a combination of these transformations such as:

- Fusion
- Fission
- Re-indexing
- Scaling
- Reversal

- Permutation
- Skewing

First we should explain the different forms of parallelism and present our problem statement. A program has $k$ degrees (number of dimensions) of parallelism if $O(n^k)$ units of computations can be executed in parallel, where $n$ is the number of iterations in a loop. Also we say that different degrees of parallelism in a loop nest exist at the same nesting level if and only if they require the same amount of synchronization. The algorithm described in this section locates all the degrees of parallelism in a program. In other words it finds the maximum degree of parallelism at each level of granularity, starting from coarsest to finest. It also finds opportunities for pipelining. This algorithm assumes there is an infinite number of virtual processors, which means for each independent thread of computation there is a processor. To generate code for a specific number of processing elements in a target architecture we can simply combine multiple of these parallel threads and assign them to the same processing element.

We can describe the overall problem of finding the maximum degree of parallelism into subproblems such as: how to maximize the degree of parallelism that requires 0, $O(1)$, and $O(n)$ amount of synchronization, where $n$ is the number of iterations in a loop. By solving each of these problems in turn, the algorithm finds successively more degrees of parallelism at a higher cost of synchronization. The above-mentioned algorithm repeats these steps to find parallelism requiring $O(n^2), O(n^3), ...$ synchronization until it finds sufficient parallelism to occupy all of the available hardware. Below we describe these subproblems with more detail.

The subproblem of maximizing synchronization-free parallelism studies the problem of parallelizing an application without allowing any communication or synchronization between processors at all. In other words it is formulated as partitioning the dynamic operations, into the largest number of independent partitions. More specifically, it finds an affine partition mapping for each instruction that maximizes the degree of parallelism. By using a set of space-partition constraints in the affine partition mapping process, we ensure that the processing elements executing operations in different partitions need no synchronization with each other.

The next subproblem is to find parallelism with $O(1)$ synchronization which

means the number of synchronizations must be independent of the number of iterations in a loop. This algorithm divides instructions into a sequence of stages (strongly connected components) and locates synchronization-free parallelism within each stage, then it inserts barrier synchronization before and after each parallelized stage.

Finally, to find parallelism with $O(n)$ synchronization, the algorithm tries to find an affine time partition mapping for each instruction. By using a set of time-partition constraints in the affine mapping process, we ensure that data dependences can be satisfied by executing the partitions sequentially. The goal is to find affine mapping that yields the maximum parallelism among operations within each of the time partitions.

The time partitions and space partitions are similar in many ways and are amenable to the same kind of techniques. The affine form of the Farkas lemma [29] has been used to transform the constraints into linear inequalities. The problem of finding a partition mapping that gives the maximum degree of loop-level and pipelined parallelism while satisfying the space-partition or time-partition constraints reduces to finding the null space of a system of equations. This affine partition mapping can be found easily with a set of simple algorithms.

## 2.2 Programming Models

In this section we go through some programming models and their pros and cons. For more information see the Tessier et al. survey paper [80]. In the early days of reconfigurable computing, there was no overlap between programming general purpose computers and FPGAs or other reconfigurable computing platforms. While procedural languages such as C were generally used to target microprocessors, most FPGAs application designers were still drawing schematics or using HDLs such as Verilog or VHDL. In order to make the reconfigurable computing platforms more accessible for general users without any hardware knowledge, several different programming models have been introduced.

A key goal in the early days was making the programming environment for reconfigurable computing platforms as similar as possible to microprocessor-based systems to make it attractive to general users. Since C was primary language of

the day, many C-based programming models were introduced. An initial C-to-hardware compiler [89] could do source to source transformation for a simple chain of operations (e.g. add, shift) into HDL code. Wo et al. [91] extended this idea by adding a simple state machine to execute multiple sequential hardware operations (considering data dependencies). To express parallelism better, features were added to the language. Early compilers often relied on users to manually express parallelism and synchronization using "pragma" statements [32]. Modern systems are closer to extracting parallelism from C source code [87] and consider the interface between the synthesized hardware and memory [99]. Application code can also be profiled using software execution [15] to determine target code for hardware acceleration. An important aspect of C-to-FPGA compilers is the estimation of functional unit area and performance [21], an issue made easier by the recent inclusion of hard macro blocks in FPGAs. Moreover, specialized systems have also been introduced to target synthesis with floating-point data types [85]. The amount of research and number of commercial procedural language-to-hardware tools (including Xilinx's Vivado, Calypto Catapult C, and Cadence's C-to-silicon) in recent years, show the demand of targeting users without hardware knowledge.

The similarity between objects in object-oriented programming models and instantiated hardware modules has led to a number of attempts to represent reconfigurable computing as communicating objects. Predictable communication flow and limited communication dependencies are key aspects of these models. This is similar to pipelined implementations. Streaming applications typically have coarse-grain compute objects that communicate with adjacent blocks via buffers or synchronized communication channels. Moreover, results are often sent along predetermined communication paths at predictable rates. This model is suitable for defining a series of signal processing blocks that require minimal control flow or global synchronization. PamBlox focused on the ability to define hierarchies of C++ objects [63] and the use of embedded block memories. These blocks could then be organized into streams. The Streams-C model [31] introduced a series of communicating sequential processes that used small local memories. Streams-C tools were later commercialized into the Impulse-C compiler. Perhaps the most comprehensive stream-based, object-oriented environment to date was the commercial Ambric model [12]. In this model, a Processing Element (PE) can exe-

cute one or more user-defined objects that communicate with objects running on other PEs via self-synchronizing, dataflow channels. The commercial Bluespec SystemVerilog [64] hardware synthesis system is also based on the manipulation of objects.

The ability to abstract away details of implementing reconfigurable platforms from the users makes stream-based environments attractive to general users. Several projects have considered the possibility of combining stream-oriented computation with run-time reconfiguration. JHDL [10] allows defining objects whose functionality can be dynamically changed. Development tools allow for evaluation of system performance using both simulation and in-circuit execution. The SCORE project [16] explores swapping stream-based objects on-demand at run-time. Objects can be swapped if the number of objects in an application is too large to fit in the hardware. As a result, the same application could be mapped to hardware platforms of many different sizes.

Data flow models are often used for specifying the behaviour of signal processing and streaming applications as a set of tasks, actors or processes with data and control dependencies. The differences between various dataflow models can be characterized by their expressive power and the availability of techniques for analyzing correctness and performance properties like absence of deadlock and throughput. The Kahn Processing Network (KPN) [30], for example, can capture many of the dynamic aspects of these systems, but evaluating their correctness and performance is in general undecidable. On the other hand, Synchronous Data Flow (SDF) [53] models do allow analysis of many correctness and performance properties but they lack support for expressing any form of dynamism.

## 2.3 Reconfigurable Computing Platforms

### 2.3.1 FPGA

FPGAs are Integrated Circuits (IC) that are used to implement digital logic functions. In contrast to an ASIC, an FPGA is field-programmable. This means that logic functions must be programmed after the device has been manufactured. On the other hand, an ASIC implements digital logic functions by placing and con-

necting transistors (layout), which cannot be changed once the chip is fabricated. An FPGA implements digital logic functions with Configurable Logic Block (CLB) modules (typically containing Look-Up Table (LUT)s and flip-flops (FF)) and configurable interconnect between them. Most modern FPGAs can be programmed by loading data into Static RAM (SRAM) cells, which can be reconfigured practically an unlimited number of times [51]. Since normal Integrated Circuit (IC) technology has been used to fabricate SRAM cells, CLBs and routing components on FPGAs, it's reasonable to describe an FPGA as a type of ASIC than emulates other ASICs. Traditionally FPGAs are used for applications such as ASIC prototyping, telecommunications equipment (where low volumes and changing standards make ASICs less attractive), and as interfaces between other ICs ("glue-logic"). In recent years, modern FPGAs have been introduced which provide more performance by adding different hard-core components such as memory blocks and multipliers. This makes them more desirable to run computationally-intensive applications such as computer vision.

### Architecture and Design Flow

A common model of an FPGA is a two-dimensional grid of blocks which are connected by a mesh routing network. The blocks may consist of CLBs or different hard blocks. The hard blocks are frequently used functions that are too expensive to implement as soft logic using CLBs. The most common hard blocks implemented by FPGA vendors are memories (Block RAM (BRAM) modules) and multipliers or other expensive arithmetic and logical functions (Digital Signal Processing (DSP) blocks). FPGA blocks communicate through a configurable routing network, usually using horizontal and routing wires meeting though reconfigurable switch blocks. Additionally, the wires within a routing channel to which a block input or output connect is configurable. The input/output blocks with pins that connect the FPGA to the outside world are also configurable, supporting multiple voltage levels and signalling styles.

FPGA architecture is relatively generic so user logic can be implemented in any set of CLBs. This means there are many possible ways to implement a general digital circuit on an FPGA. This process is not straightforward: the size of modern

FPGAs are large, so mapping a large design to a large FPGA is an optimal way is computationally intensive. To appreciate the problem size, consider that the largest Xilinx Virtex UltraScale+ FPGA has 3.8 million programmable logic cells, 94 Mb BRAM and 12,288 DSP slices.

The Computer Aided Design (CAD) flow for translating a design to an FPGA configuration bitstream varies for different vendors. For more information refer to a survey paper by Chen et al [17]. The differences are not important for the purposes of this dissertation, but it is necessary to understand a CAD flow to understand the reasons the traditional FPGA design cycle is long compared to the design cycle for software. RTL synthesis is the process of translating the input circuit as specified by the user to a netlist of Boolean functions and macros such as hard blocks. The Boolean functions get technology-mapped to FPGA programmable logic blocks. Placement then selects locations for each of these units, and routing determines how to configure the communication network to connect logic block inputs and outputs. Finally, assembly creates the bitstream that is used to program the FPGA. Note the term synthesis is often used to refer to the entire process; it includes the time taken by all of the steps. For large FPGAs, synthesis can take hours or even days. This fact makes automatically exploring the space/time tradeoffs and using a library of pre-synthesized bitstream implementations desirable for users.

### 2.3.2 Massively Parallel Processor Array

A Massively Parallel Processor Array (MPPA) is a type of embedded platform which has an array of hundreds or thousands of processing elements (processors) and memories (Random Access Memory (RAM)). Processors in the system pass work to one another through a reconfigurable interconnect of channels (e.g. First In, First Out (FIFO) buffers). A general MPPA is Multiple Instruction, Multiple Data (MIMD) architecture, often with local distributed memory. Communication between processors is realized in the configurable interconnect. Each processor can often run independently at its own speed. Several different architectures for MPPAs have been introduced by both industry and academia. Companies such as Aspex (Ericsson), Ambric, PicoChip, Intel, IntellaSys, GreenArrays, ASOCS, Tilera, Kalray, Coherent Logix, Tabula, and Adapteva have introduced their MPPAs

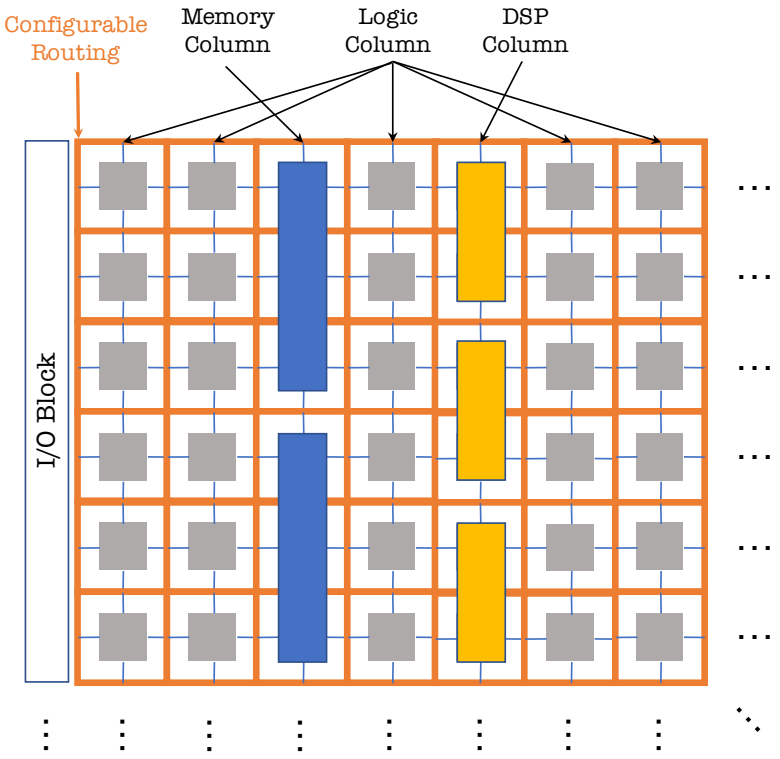


**Figure 2.1:** Example FPGA architecure

in recent years. Academia has introduced some architectures as well such as AsAP [100]. Below we discuss some popular MPPAs.

PicoChip (acquired by Intel) developed a multi-core digital signal processor, the picoArray [27]. This integrates 250-300 individual DSP cores onto a single die (depending on the specific product) and as such it can be described as an MPPA. Each of these cores is a 16-bit processor with Harvard architecture, local memory and 3-way Very Long Instruction Word (VLIW). The picoArray is programmed using a mixture of VHDL [2], ANSI/ISO C and assembly language. The VHDL is used to describe the structure of the overall system, including the relationship between processes, and the signals which connect them together. Each individual process is programmed in conventional C (albeit with additional communication functions), or in assembly language.

The Epiphany architecture consists of a low power, multi-core, scalable, parallel, distributed shared memory embedded system created by Adapteva [1]. The Epiphany IV Network on Chip (NOC) co-processor contains 64 cores (referred to as eCores) organized in a 2D mesh with future versions expected to house up to 4096 eCores. The Epiphany chip can be programmed using C, and has a Software Development Kit (SDK) but users need to manually find and express parallelism.

The Kalray MPPA was introduced as a single-chip many-core processor that integrates 256 user cores and 32 system cores in 28nm CMOS technology [24]. These cores are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them uses a NOC. This processor targets embedded applications whose programming models fall within the following classes: KPN, as motivated by media processing; single program multiple data (SPMD), traditionally used for numerical kernels; and time-triggered control systems.

University of California, Davis, introduced Asynchronous Array of Simple Processors (AsAP) [100] which contains an array of simple RISC processors with a nine-stage pipeline with small instruction and data memories. Processors communicate only with adjacent processors to permit full-rate communication with low energy. Each processor can receive data from any two neighbors and send data to any combination of its four neighbors. The first generation was introduced with 36 processors. The second generation has 167 processors [86] for DSP, communication, and multimedia workloads. It contains 164 programmable processors with dynamic supply voltage and dynamic clock frequency circuits, three algorithm-specific processors, and three 16 KB shared memories, all clocked by independent oscillators and connected by configurable long-distance-capable links.

### Ambric

Ambric Inc. was a high performance computing company founded in 2003 in Oregon state, United State of America. Their Am2045 MPPA chips have 336 32-bit RISC-DSP fixed-point processors and run up to 300 MHz. They were designed for high-performance embedded systems such as medical imaging, video, and signal-

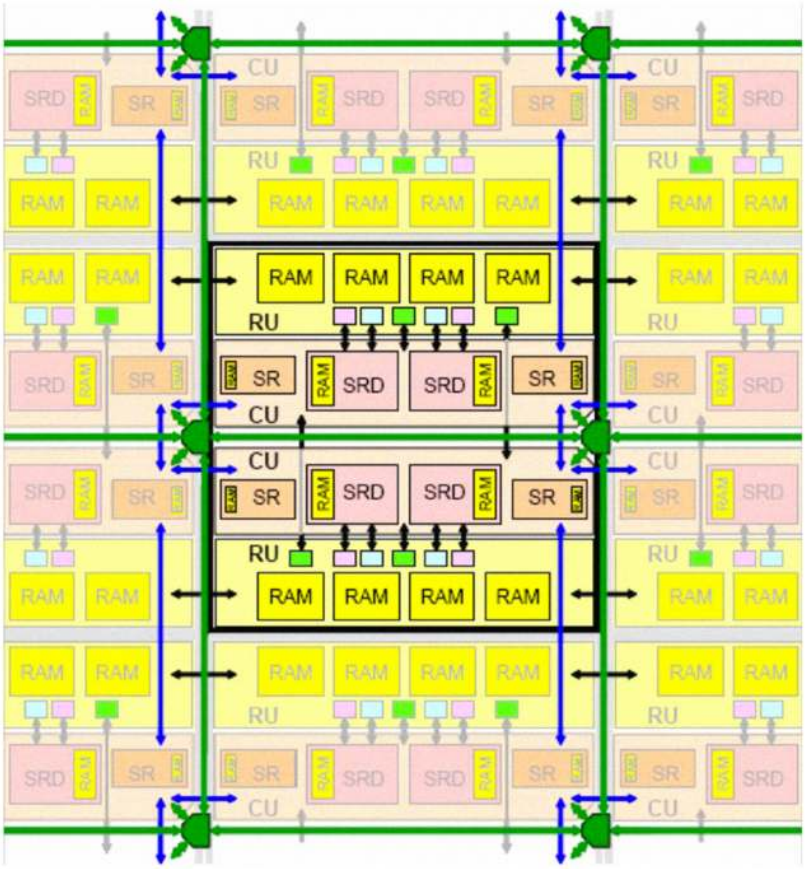


**Figure 2.2:** Ambric *bric* organization [12]

processing. The Am2045 is internally organized into a $5 \times 9$ array of *bric* modules. Figure 2.2 shows one *bric* and its neighbouring *brics*. Each *bric* contains two kinds of 32-bit Central Processing Unit (CPU)s. SRD processors contain 3 Arithmetic Logic Unit (ALU)s and provide math-intensive instructions to support DSP operations. Each SRD processor contains a dedicated 256-word RAM for instructions and data. This memory can be augmented though direct connections to *bric* memory objects. SR processors are lighter weight with only 1 ALU. They contain a dedicated 128-word memory for programs and data but do not have direct connections to memory objects. Each of the two memory objects (RU) in a *bric* is organized as 4 independent RAM banks.

Ambric introduced the Am2045 and its software tools in 2007, but fell victim to the 2008 worldwide financial crises [90].

Microprocessor Report gave a 2006 MPR Analysts' Choice Award for innovation for the Ambric architecture "for the design concept and architecture of its massively parallel processor, the AM2045". Although Ambric Inc. is defunct, in 2013 the Ambric architecture received the Top 20 award from the IEEE International Symposium on Field-Programmable Custom Computing Machines, recognizing it as one of the 20 most significant publications in the 20-year history of the conference.

Software written for Ambric devices is based on the Structural Object Programming Model [12]. Each processor is programmed in conventional Java (a strict subset) and/or assembly code (Figure 2.3). A programmed processor or memory is called a primitive object. Objects run independently at their own speeds. They are strictly encapsulated, execute with no side effects on one other, and have no implicitly shared memory. Objects intercommunicate through channels (FIFO-buffers) in hardware. Channels carry both data and control tokens. Channel hardware synchronizes its elements at each end, not at compile time but dynamically as needed at run time. Inter-processor communication and synchronization are combined in these channels. The transfer of a word on a channel is also a synchronization event. Processors, memories, and channel hardware handle their own synchronization transparently, dynamically, and locally, so it doesn't have to be done by the developer or the tools. Channels provide a common hardware-level interface for all objects. This makes it simple for nodes to be assembled into higher-level composite nodes. Because nodes are encapsulated and interact only through channels, composite nodes work the same way as primitive node objects. The developers express object-level parallelism using a block diagram. First the hierarchical structure of primitive and composite objects is defined, connected by channels. Then ordinary sequential software is written to implement the primitive objects. We have used an architecture and programming similar to Ambric in chapter 3.

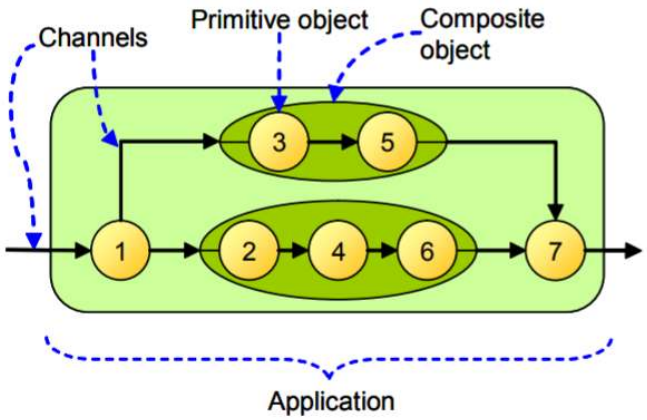


**Figure 2.3:** Structural object programming model [12]

## 2.4 OpenVX

OpenVX is a cross-platform, C-based API standard for Computer Vision applications. OpenVX is a good programming model for embedded systems because it enables performance and power-optimized CV processing to be written in a way that is independent of the target architecture. At the lowest level are OpenVX kernel functions; these are implemented as a library by developers with detailed knowledge of the target accelerator. In OpenVX, CV applications are implemented as a set of these vision kernel functions which communicate through streaming channels. An OpenVX application assembles these kernels, or nodes, into a graph, where edges convey an image passed between kernels. For example, Figure 2.4 shows the OpenVX code for Canny edge detection and Figure 2.5 shows the corresponding graph.

The OpenVX run-time system can break a large image into smaller image tiles, allowing each small tile to pass through the entire graph to completion before writing back to external memory. This provides excellent memory locality and improves both performance and power keeping the tile and its transformations on-chip for as long as possible. Most kernel functions can work with tiles because they need only local information. The few kernel functions that need global information cannot be executed until the full image output of the preceding kernel functions is computed; this can be achieved by cutting the graph at this point and

```
//Canny example
vx_node nodes[] = {
vxColorConvertNode(graph,rgb, gray),
vxGaussian3x3Node(graph,gray,gauss),
vxSobel3x3Node(graph,gauss,gradx,grady),
vxMagnitudeNode(graph,gradx,grady,mag),
vxPhaseNode(graph,gradx,grady,phase),
vxNonMaxima(graph,mag,phase,nm),
vxThreshold(grpah,nm,output)
};
```

**Figure 2.4:** OpenVX source code for Canny

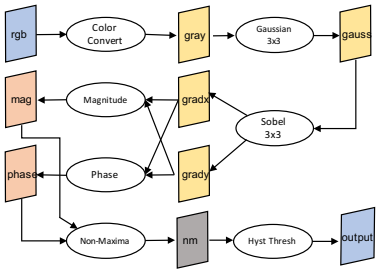


**Figure 2.5:** OpenVX graph for Canny

only operating on full images at these cut points.

OpenVX applications can be defined as streaming applications: each stage receives stream of image pixels, rows or frames, processes them, and sends the results as stream of data to the next stage. This means we can describe a computation as compute graphs or Synchronous Data Flow Graphs (SDFG) [53]. Previous studies have shown that custom hardware implementations on FPGAs as well as MPPAs have the potential to dramatically increase the performance/Watt for computationally intensive SDFGs [4, 40].

## 2.5 Partial Reconfiguration

The functionality of an FPGA is created by loading its configuration with a set of bits called a bitstream. In most applications, the entire FPGA is configured at once with a single bitstream. The Partial Reconfiguration (PR) feature available in most SRAM-based FPGAs allow only a portion of the FPGA to be reconfigured, with the rest of the bits staying intact. PR allows changing behaviour of partitions in the FPGA architecture while the remaining logic is still running.

There are two main uses for PR: to save parallel resources and power [42, 69], or to increase performance. There have been several studies on the use of PR. For example [19, 50] demonstrate using partial FPGA reconfiguration for image processing, [20, 83] propose self-adaptive control systems, and [28, 72] investigate the use of PR for High Performance Computing (HPC).

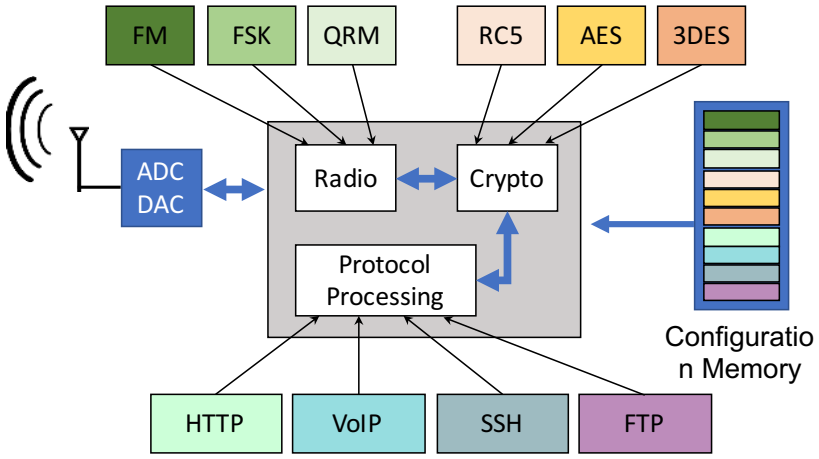


**Figure 2.6:** Area saving by reconfiguring only the currently required accelerator module to the FPGA. Configurations are fetched from the module repository at runtime.

Below we will go through two examples to illustrate the benefits of PR [47].

First, Figure 2.6 shows an adaptive communications device where each of three stages (the front-end radio, the cryptographic accelerator, and the protocol processing engine) can be loaded with three or four different 'algorithms'. The algorithms within each stage are never needed at the same time, i.e., they are mutually exclusive. In a traditional hardware system, all of these algorithms would be implemented at the same time and use considerable area. In a PR system, only enough area for the largest algorithm in each stage needs to be reserved, leading to considerable savings.

There are more potential benefits than only power and area savings. If we can implement a system with a smaller FPGA, we might be able to use a smaller package, which is especially important for mobile applications. In the case of more complex systems that demand multiple FPGAs, PR may reduce the total FPGA count, thereby simplifying PCB and system design. Due to higher integration, we may be able to perform more data processing on-chip, thereby reducing energy for chip-to-chip communications and/or memory writes.

Second, partial reconfiguration can help increase system performance. For example, in Figure 2.7, the PR system on the right can perform more work to achieve

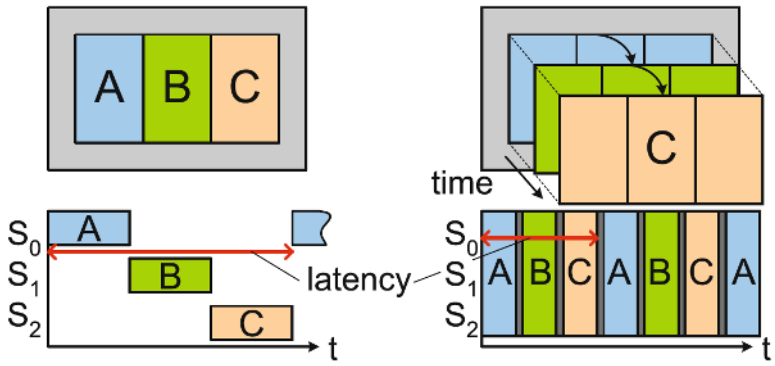


**Figure 2.7:** System acceleration due to partial reconfiguration. By spending temporarily more area for each function, the overall latency is reduced[47].

lower latency and higher throughput than the one on the left (without PR). If we assume that modules A, B and C are executed one after another, the system on the left requires 3 time steps to operate on a single block of data. In contrast, the system on the right can dedicate all resources to step A before using PR to move to step B.

A practical example for this is a hardware accelerated secure SSL connection. In this protocol, we first exchange a session key using asymmetric key exchange. After this, the session key is used by a symmetric cipher for the actual data transfer. Consequently, both steps are executed strictly sequentially and we can dedicate more resources for each step and reduce latency by using PR.

Partial run-time reconfiguration has been an active research field for more than two decades. Much research has been done on increasing the reconfiguration speed and reducing the reconfiguration time. Many different approaches and ideas have been proposed. Hauck introduced the concept of configuration prefetching to swap modules in the background to hide latency and improve the reconfiguration time [36]. Dittmann et al. used reconfiguration port scheduling as well as configuration preemption to improve the efficiency of the reconfiguration interface and to decrease the reconfiguration time [26]. Lange et al. introduced a hyper-reconfigurable architecture to reduce the amount of configuration

data needed to perform reconfiguration, which results in lower reconfiguration time and faster reconfiguration speed [52]. Different studies have investigated bitstream (de)compression to reduce the required bandwidth of configuration storage and to reduce the reconfiguration time [39, 48, 54]. Moreover, several different types of reconfiguration controllers, with or without Direct Memory Access (DMA) capabilities, have been investigated to improve the reconfiguration speed and reduce the reconfiguration time [18, 58, 61].

Although FPGA PR rates are relatively low, it is possible to obtain increased performance. For example, Xilinx FPGAs provides 400MByte/sec configuration speed with its on-chip ICAP controller ICAP. Hansen et al. showed it possible to achieve 2.2GByte/sec with existing Xilinx FPGAs [35] by overclocking the ICAP controller. In addition, other studies have suggested different architectures to improve PR time. For example, Trimberger et al. proposed a time-multiplexed FPGA architecture with a 33GByte/sec reconfiguration rate [84]. While faster reconfiguration times are possible, FPGA vendors have not yet seen sufficient need or demand to provide this feature.

# Chapter 3

# MPPA Space/Time Scaling

In this chapter, we automate space/time scaling for STGs by developing a Java-based compilation tool chain targeting a pipelined coarse-grained architecture that is very similar to the Ambric MPPA chip architecture. Similar to the Ambric tools, our compiler accepts input written in a subset of Java. Unlike the Ambric tools, our compiler analyzes the parallelism internal to each node and evaluates the throughput and area of several possible implementations. After finding different implementations for each node, it then analyzes the full graph for bottlenecks or excess compute capacity, and selects an implementation for each node while either minimizing area (for a fixed throughput target), or maximizing throughput (for a fixed area target). To find a better area/throughput tradeoff, we use node combining and splitting in the graph. We present two optimization approaches, a formal ILP formulation based on prior work and a novel heuristic solution. Results show that the heuristic is more flexible and can find design points that are computationally infeasible to find using the ILP, thereby achieving superior results with a faster runtime.

## 3.1 Introduction

In this chapter, we will investigate whether an array of ALUs or very lightweight processors, described best as a massively parallel processor array or MPPA, can achieve sufficient levels of performance, and make design entry sufficiently easy,

to make them an interesting alternative to more traditional design methods for running STGs. Moreover, we propose a novel approach to automatically explore space/time tradeoffs that can produce different optimized implementations for different throughput targets or different area budgets.

To explore the MPPA as an alternative target, we need a programming model and a tool flow that can compile algorithms into the target and use parallel resources efficiently. To make an MPPA a truly high performance platform, the programming model should support some of the strengths of FPGAs, especially pipelined parallelism. This led us to start with the explicit streaming model and architecture that was defined by Ambric [12, 13].

In the Ambric model, a Java object is created for each thread. Threads are nodes in a graphs defined as instances of objects that communicate together through explicitly defined blocking FIFO communication channels. The node can be a primitive node ranging from a single operation to multiple loop nests and complex conditions, or a composite node containing more than one primitive node. The objects and channels are placed and routed onto an array of 336 processors with a mesh NoC. Each object contains local state and a processing thread. Processing an object can be variable latency, but computation between objects is synchronized through the blocking FIFOs. Objects may be replicated, thus facilitating some re-use of a program, but all instances are explicitly allocated and defined by the programmer at compile-time. This is very similar to a KPN [30], except that in a KPN the FIFOs are assumed to be infinitely deep. The resulting process network exhibits deterministic behaviour that does not depend on the various computation or communication delays.

One of the drawbacks of the Ambric framework is the need for explicit allocation of all objects and channels. The number of objects, and the computational delays within each object, define the amount of parallelism and the throughput of the application. Thus, scaling a program to a larger or smaller processor array requires manually re-programming all objects and channels. For the Ambric commercial solution consisting of a single device, this is an acceptable trade-off. However, for a research platform, we must investigate a variety of array sizes, as well as simpler or more complex processors, which requires automatically transforming a streaming application to use more or less space, thereby increasing or

decreasing throughput.

In our model there is no long global interconnect. Instead, each PE can only send/receive data to/from its immediate neighbours. In order to send/receive data from/to more PEs, intermediate PEs are needed to disseminate the values. To evaluate functionality of our model and measure execution time, we developed a cycle-accurate simulator. Note, the simulator doesn't support 2D placement which needs to be addressed in the future work. Although the simulator doesn't support 2D placement, we manually placed sample benchmarks to make sure the model is not dependent on long placement delays.

In this chapter, we describe our compilation tool that can perform automated space/time tradeoffs. The user describes an initial program in Ambric-style Java, and then defines either a throughput target, or an area budget. The compiler analyzes the processing rate of each object (or thread), and propagates these throughputs across the entire computational graph (defined by the communication channels). It also analyzes each thread to determine the degree of internal parallelism. Using this information, it transforms the compute graph to meet the area or throughput target. There are a variety of transformations such as replicating objects (requiring a split/join on the data), subdividing objects into a deeper pipeline (increasing throughput), and merging objects together (decreasing area). At all times, a whole-program approach is taken to optimization, so portions of a program that are not performance-critical will be merged to use less area, and more area will be allocated to performance-critical regions. This alleviates some effort from the programmer, and creates a scalable/re-targetable implementation.

Our tool uses two internal optimization approaches. The first is based upon ILP and is based on prior work on task graph optimizations by Cong et al. [22]. The second, based upon a heuristic approach, is our own novel contribution. Although the ILP approach works well, it can be difficult or impossible to represent some types of optimizations as ILP constraints. In particular, our heuristic approach is able to perform object coalescing, which is difficult to model within the ILP formulation; doing so adds a large number of additional constraint variables which quickly make the ILP computationally infeasible. This additional optimization gives the heuristic considerable area savings over the ILP approach.

Figure 3.1 illustrates the flow for the proposed tool. It compiles a program

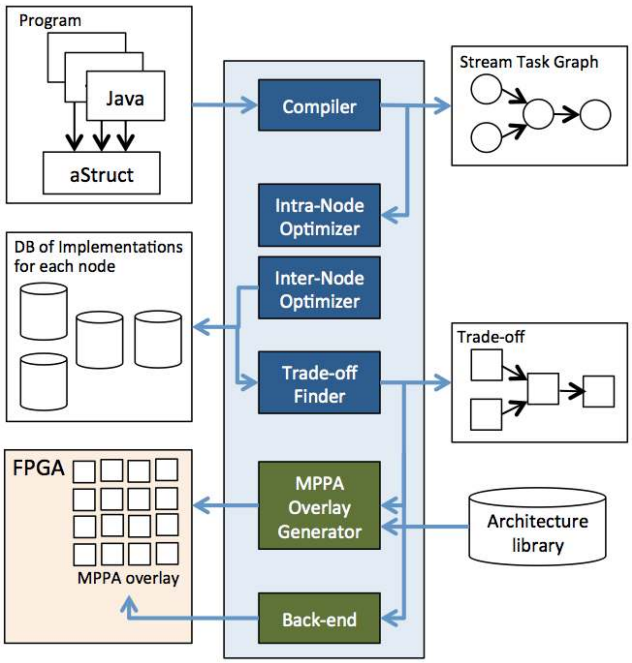


**Figure 3.1:** Tool flow

described in Ambric-style Java (compute) and aStruct (communication), and creates a STG complete with composite nodes communicating with each other through channels (edges of the graph). It uses Intra-Node Optimizer and Inter-Node Optimizer in order to find different implementations for each composite node. It uses Trade-off Finder to find a good trade off between throughput and area.

## 3.2 Finding Different Implementations

Consider an application with N composite nodes $f_1, f_2, ..., f_N$ in its STG. For each composite node $f_m$, our tool tries to find different implementations $P_m^1, P_m^2, ..., P_m^{S_m}$ where each implementation $P_m^s$ can perform the functionality of $f_m$ with area cost $A(P_m^s)$ and initiation interval $II(P_m^s)$. For node $f_m$ and its implementation $P_m^s$, the minimal input "inverse-throughput" $\vartheta_{in}(P_m^s)$ and output inverse-throughput $\vartheta_{out}(P_m^s)$ are calculated as

$$\vartheta_{in}(P_m^s) = \frac{II(P_m^s)}{In(f_m)}, \vartheta_{out}(P_m^s) = \frac{II(P_m^s)}{Out(f_m)} \tag{3.1}$$

where $In(f_m)$ and $Out(f_m)$ equal the number of data tokens that $f_m$ consumes on the input data channel and produces on the output data channel during each firing, respectively. Note that inverse-throughput shows the number of cycles used to consume/produce per datum in its input/output channel. Intra-Node Optimizer and Inter-Node Optimizer modules have been implemented in our tool to automatically find these abovementioned implementations for each composite node.

### 3.2.1 Intra-Node Optimizer

Affine loop transformation strategies in [6–8, 56, 74] are used in Intra-Node Optimizer to find the maximum degree of parallelism for each composite node. After finding the maximum degree of parallelism, Intra-Node Optimizer tries to find the best throughput (minimizing the inverse-throughput) for each node for different area costs. Since each operation needs a different number of clock cycles to provide its output (different inverse-throughput), Intra-Node Optimizer expands, combines, splits and, pipelines nodes regarding the inverse-throughput of operations inside the composite node in order to find an implementation with highest throughput for each composite node for different area costs. Below, the different optimization techniques used in Intra-Node Optimizer are listed.

- Loop Fusion
- Loop Fission
- Loop Re-indexing
- Loop Scaling
- Loop Reversal
- Loop Permutation
- Loop Skewing
- Node Replication
- Node Combining

- Node Splitting
- Pipelining

### 3.2.2 Inter-Node Optimizer

After finding the implementation with the highest throughput for each composite node for each different defined area cost, Inter-Node Optimizer is a clustering operation that finds different implementations. The clustering operation was implemented based on a previous study by Amit Singh et al. [77]. Each cluster will be mapped to one Processing Element. Inter-Node Optimizer sends operations back and forth between clusters to find optimum area cost for each overall inverse-throughput target. The example below illustrates how our tool works.

Here, we go through an example to demonstrate how Intra-Node and Inter-Node Optimizers work.

### 3.2.3 Example: N-Body Problem

The N-body Problem simulates a 3D universe, where each celestial object is a body, or particle, with a fixed mass. Over time, the velocity and position of each particle is updated according to interactions with other particles and the environment. In particular, each particle exerts a net force (i.e., gravity) on every other particle. The computational complexity of the basic all-pairs approach we use is $O(n^2)$. The runtime is dominated by the gravity force calculation, shown below:

$$\overrightarrow{F_{i,j}} = G\frac{M_i M_j}{r^2} = 0.0625\frac{M_i M_j}{|\vec{P_i} - \vec{P_j}|^3}(\vec{P_i} - \vec{P_j}) \tag{3.2}$$

Where $\overrightarrow{F_{i,j}}$ is the force particle $i$ imposes on particle $j$, $P_i$ is the position of particle $i$, and $M_i$ is the size or "mass" of particle $i$. When mapping the force calculation, because of the dependencies between instructions in this code, our tool first pipelines it. A simplified 2D pipeline STG (with inverse-throughput) for the gravity force calculation is shown in Figure 3.2 and its corresponding Java source code is shown in Listing 3.1.

Consider mapping each operation to a simple PE. Since everything is pipelined, we get the highest throughput per datum when each operation consumes/produces

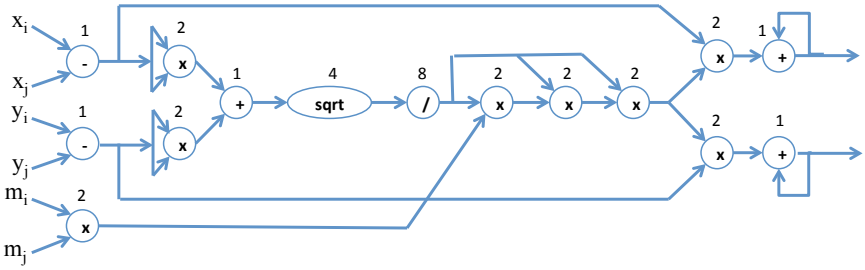


**Figure 3.2:** Pipelined force calculation

**Listing 3.1:** Force calculation Java code

```
void force_calc()
{
    for( int i = 0 ; i < NUMBER_OF_PARTICLES ; i++ ) {
        gmm = ref_gmm * m[i];
        dx = ref_px - px[i];
        dy = ref_py - px[i];
        dx2 = dx * dx;
        dy2 = dy * dy;
        r2 = dx2 + dy2;
        r = sqrt(r2);
        rr = 1/r;
        gmm_rr = rr * gmm;
        gmm_rr2 = rr * gmm_rr;
        gmm_rr3 = rr * gmm_rr2;
        dfx = dx * gmm_rr3;
        dfy = dy * gmm_rr3;
        result_x = result_x[i] + dfx;
        result_y = result_y[i] + dfy;
        result_x[i] = result_x;
        result_y[i] = result_y;
    }
}
```

**Table 3.1:** Different operations with their initiation intervals

| Operation | Initiation Interval |
|---|---|
| ADD / SUB | 1 |
| MUL | 2 |
| SQRT | 4 |
| DIV | 8 |

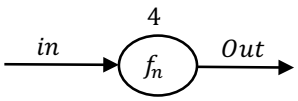


**Figure 3.3:** A node with inverse-throughput=4

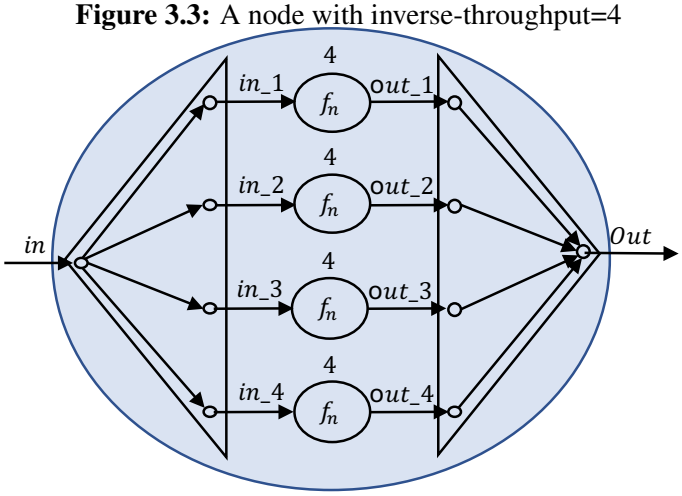


**Figure 3.4:** Expanding node using replication to improve throughput

one datum in one clock cycle (inverse-throughput=1). As shown in Table 3.1, each operator has a different initiation interval. Since each operation (primitive node) consumes/produces one data from/to its input/output in each firing, the inverse-throughput is equal to the initiation interval. For example, division needs eight cycles to provide its output (inverse-throughput equals to 8), which make it the slowest node in the STG. Because of dependencies between nodes, faster operations have to stall for division. This means the best overall inverse-throughput we can get with this mapping is 8. To get the highest throughput, Intra-Node Optimizer uses an "expanding" approach to parallelize those nodes with high inverse-throughput (slow nodes).

One approach to expanding is replicating. Figure 3.3 shows a node ($f_n$) with inverse-throughput equal to 4. This means it takes 4 clock cycles for the node to generate an output. To improve the throughput, it's possible to replicate node $f_n$, four times, pass the inputs in a round-robin order to each of them, and then gather the outputs coming from each replicated node in the same order. Figure 3.4 shows the expanded node for this situation. The expanded node improves throughput by a factor of 4 and is capable of receiving or sending one datum per clock cycle. Simi-

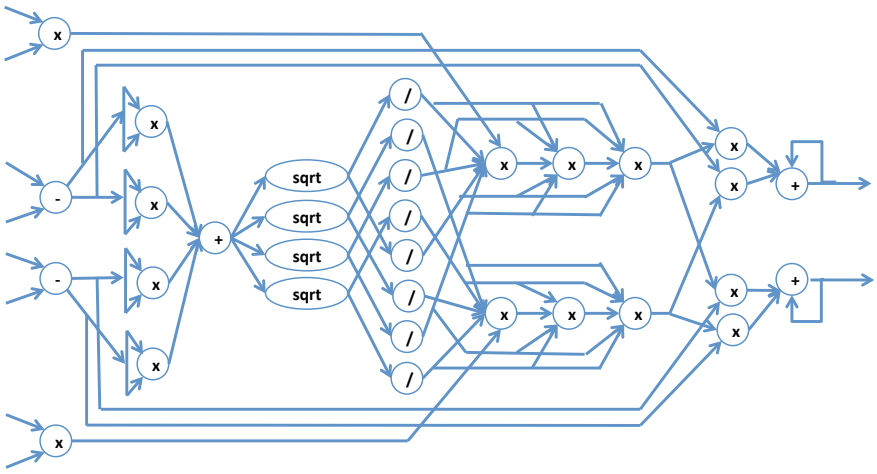


**Figure 3.5:** Expanded force calculation

larly, we can use this expanding method over all slow nodes in the force calculation STG. Figure 3.5 shows an improved expanded STG for force calculation after using Intra-Node optimizer, where the overall inverse-throughput equals to 1.

After finding the highest possible throughput, Inter-Node Optimizer tries to cluster and combine nodes again to find several implementations with different throughput and area. It means that Inter-Node Optimizer sacrifices the throughput to save area. It continues this procedure until it assigns the entire composite node to one PE (Area cost = 1). Figure 3.6 shows inverse-throughput and area relation for different implementations of the gravity force calculation function. Here, the inverse-throughput varies from 1 to 33. Moreover, to achieve the best throughput (inverse-throughput = 1), we can either replicate the slowest implementation (inverse-throughput=33) into 33 copies or use the fastest implementation directly (instruction level parallelism or data level parallelism).

We used the gravity force calculation example as a simple example to demonstrate how the tool deals with a general STG. Of course depending on the different STG topologies and data dependencies, the tool might not be able to use all the parallelism methods.

After finding several different implementations for an STG, the tool needs to find a suitable implementation for a given user restriction. There are different modes: a defined throughput target, or a defined area budget. To do so, the tool

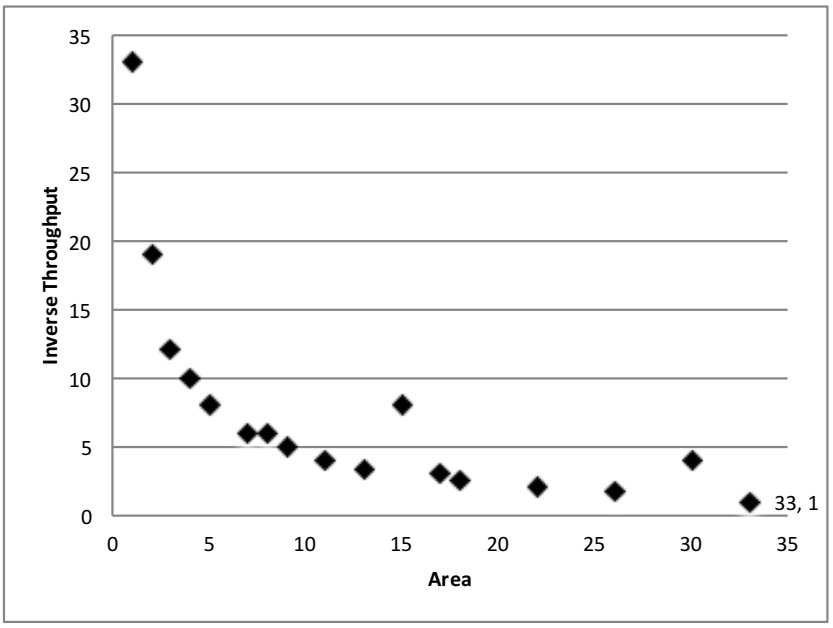


**Figure 3.6:** Inverse-throughput/area relation for different implementations of force calculation

needs to automatically explore space/time tradeoffs and find a suitable tradeoff for either of the modes. Below, we define the problem for these two modes and use two different approaches to solve the optimization problem.

## 3.3 Trade-off Finding Formulation and Solutions

To solve the trade-off finding problem we used an ILP approach as well as a heuristic approach. For each of those approaches, there are two different modes or objectives which a user can choose:

- Given an available area target $A_{tgt}$ and different implementations for each node $f_j$, which implementation $P_j^i$ should be selected and how many replicas $n_j^i$ are needed in order to minimize application inverse-throughput $\vartheta_A$ subject to the constraint the application area cost $A_A$ is not bigger than $A_{tgt}$.

- Given an inverse-throughput target $\vartheta_{tgt}$, and different implementations for each node $f_j$, which implementation $P_j^i$ should be selected and how many

replicas $n_j^i$ are needed in order to minimize area cost $A_A$ subject to the constraint the application inverse-throughput $\vartheta_A$ is not bigger than $\vartheta_{tgt}$.

### 3.3.1 Integer Linear Programming Algorithm

The first Trade-off Finding algorithm simply defines the problem as an Integer Linear Programming (ILP) problem based on prior work by Cong et al. [22]. The goal is to find binary integers $x_{j,1}, x_{j,2}, \ldots, x_{j,S_m}$ indicating the implementations to be selected, and integer $nr_j^i$ indicating the number of replicas needed. Equation 3.3 shows the formulation of finding a suitable implementation for a given area budget.

$$\begin{aligned} &\text{Minimize } \vartheta_A \text{ subject to:} \\ &\forall j \in \{1,...,N\} : \sum_{j=1}^{N} \sum_{i=1}^{S_m} nr_j^i A(P_j^i) x_{j,i} < A_{tgt} \text{ and } \sum_{i=1}^{S_m} x_{j,i} = 1. \end{aligned} \tag{3.3}$$

Equation 3.4 shows the formulation of finding a suitable implementation for a given throughput target.

$$\begin{aligned} &\text{Minimize } A_A \text{ subject to:} \\ &\forall j \in \{1,...,N\} : \sum_{i=1}^{S_m} \frac{1}{nr_j^i} \vartheta(P_j^i) x_{j,i} < \vartheta_{tgt} \text{ and } \sum_{i=1}^{S_m} x_{j,i} = 1. \end{aligned} \tag{3.4}$$

An ILP solver such as GNU Linear Programming Kit (GLPK) [60] could go through all the possibilities of $x_{j,i}$ and $nr_j^i$ and find the optimum solution for this problem, subject to the constraints. Although ILP solvers can solve these problems, the approach does have two shortcomings:

- Lack of flexibility: it is difficult and sometimes impossible to represent some types of optimizations as ILP constraints. In particular, combining or splitting nodes requires adding all combinations of merges and splits to the ILP problem formulation, which introduces a very large number of additional constraint variables. This quckly becomes computationally infeasible to solve.

- Time inefficient: In our experiments, ILP (without the node combining or splitting optimization) is usually slower than our heuristic algorithm.


**Figure 3.7:** Minimum and expected inverse-throughput

### 3.3.2 Heuristic Algorithm

Before describing our heuristic approach, we must first define Throughput Analysis, Throughput Propagation and Bottleneck Optimizer.

#### Throughput Analysis

Each node achieves the maximum output throughput if and only if all its input data are ready when the node expects them. To make the throughput analysis more straightforward we use inverse-throughput instead of throughput. To achieve minimum output inverse-throughput $\vartheta_{mo}$, the input data have to be ready with expected inverse-throughput $\vartheta_{ei}$. We define inverse-throughput slack $\vartheta_s$ for each channel as:

$$\vartheta_s = \vartheta_{mo} - \vartheta_{ei} \tag{3.5}$$

Figure 3.7 shows a simple example in which two nodes A and B are connected together. Node A is a potential bottleneck if it doesn't provide data fast enough to satisfy node B's expectation ($\vartheta_{mo} > \vartheta_{ei} \iff \vartheta_s > 0$). Node B is a possible bottleneck if node A provides data faster than what node B consumes ($\vartheta_{mo} < \vartheta_{ei} \iff \vartheta_s < 0$).

Throughput analysis helps us to find possible bottleneck nodes in a system as well as unnecessary high-throughput nodes. Figure 3.8 shows an example with seven nodes with different $\vartheta_{mo}$ and $\vartheta_{ei}$. We calculated slack $\vartheta_s$ for each channel. As we can see, $\vartheta_s$ for input channels going to $f_3$ are smaller than $\vartheta_s$ for other input channels. Also, $\vartheta_s$ for output channel from $f_3$ is bigger than $\vartheta_s$ for other output channels. This shows $f_3$ is a potential critical bottleneck. To find potential bottlenecks in an STG, we define the weight $W_m$ for each node $f_m$ as:

$$W_m = \frac{\sum_{j=1}^{N_{out}} \vartheta_{sj} - \sum_{i=1}^{N_{in}} \vartheta_{si}}{N_{out} + N_{in}} \tag{3.6}$$

where $\vartheta_{si}$ is the input throughput slacks for incoming channels and $\vartheta_{sj}$ is the output

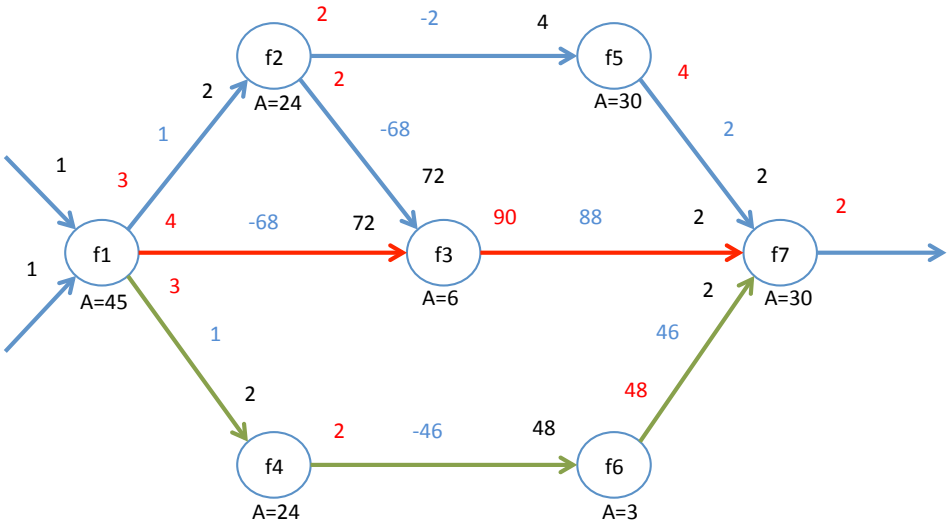


**Figure 3.8:** Throughput analysis example

throughput slacks for outgoing channels of $f_m$. $N_{in}$ denotes the number of inputs, and $N_{out}$ denotes the number of outputs for node $f_m$. A higher weight means that the node is not able to provide/consume expected outgoing/incoming data to/for its neighbors in most of its channels and the throughput differences between this node and its neighbor are critical which makes that node a potential bottleneck. A set of weights is calculated for all nodes in the graph and will be used in the heuristic to find critical bottlenecks in the graph. This weight set will be updated every time a bottleneck is improved by Bottleneck Optimizer.

### Throughput Propagation

Although it seems trade-off finder should select an implementation for each node in order to increase its throughput, increasing throughput of a node will not necessarily increase the overall throughput. For example, as shown in Figure 3.9, optimizing the block *B* doesn't increase the overall throughput of the STG (in either case) because the STG always has to wait for block *A* which takes 9 clock cycles to generates its output. In this case, either we need to improve the throughput of node *A*, or we can sacrifice the throughput of block *B* and make it slower to release some area. This example shows that we cannot consider improving a node inde-

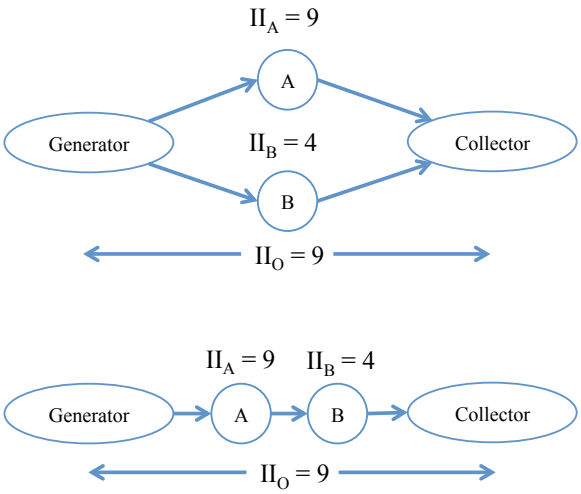


**Figure 3.9:** Throughput propagation and balancing

pendently without looking at the whole STG, examining each node's predecessors and successors. In other words, we need to have a balanced throughput throughout the STG.

To balance the throughput for each node and the entire STG, we have to propagate the target inverse-throughput to all nodes in the application. For propagating the input inverse-throughput to the output inverse-throughput for a single node, we used a similar strategy to Jason Cong's previous work [22]. For a node $f_m$, the number of input and output channels are denoted as $numIn(f_m)$ and $numOut(f_m)$, the number of data tokens that $f_m$ consumes/produces on the input/output channel is denoted as $In^j(f_m)/Out^k(f_m)$, and the inverse-throughput on the input/output channel is denoted as $\vartheta^j_{in}(f_m)/\vartheta^k_{out}(f_m)$, where $1 \leq j \leq numIn(f_m)$ and $1 \leq k \leq numOut(f_m)$. Given the input inverse-throughput target $\vartheta^j_{in}(f_m)$, the output inverse-throughput target $\vartheta^k_{out}(f_m)$ is calculated as Equation 3.7.

$$\vartheta^k_{out}(f_m) = \frac{\min_j\{\vartheta^j_{in}(f_m) In^j(f_m)\}}{Out^k(f_m)} \tag{3.7}$$

Equation 3.7 allows propagating throughput from input nodes in the STG, through other nodes all the way to output nodes. It helps the tool to find the overall application throughput as well as possible bottlenecks which helps the tool to balance the

throughput of all nodes in the STG in the heuristic approach. This helps the tool to eventually generate a balanced implementation for a throughput target or an area budget.

**Bottleneck Optimizer**

Bottleneck Optimizer is very similar to the ILP approach in that it makes replicas of the bottleneck to increase throughput. However, the ILP replicates the bottleneck without any attention to its neighbouring nodes so it can miss opportunities to have lower area overhead. To overcome this deficiency, we propose a method that relies on the fact that each node can send/receive data to/from up to FanIn/FanOut number of nodes without any area overhead cost. If more than FanIn/FanOut number of replicas are required, some overhead cost is inevitable. For example, in our Ambric replica, a FanOut beyond 4 requires extra area. In this situation, to connect these replicas to more successor/predecessor nodes, new fork/join nodes are needed to send/receive data to each replica. Let us go through a simple example to show the overall idea in our Bottleneck Optimizer approach. Figure 3.10.a shows an example in which two nodes, $S$ with inverse-throughput $\vartheta_s$, and $D$ with inverse-throughput $\vartheta_D$, are connected together. Node $S$ is sending data to $D$ over a channel. Assume the node $D$ is a bottleneck, and we want to match its throughput to node $S$'s throughput. To match the throughput, we need $nr$ replicas of node $D$, which is calculated as

$$nr = \left\lceil \frac{\vartheta_D}{\vartheta_S} \right\rceil \tag{3.8}$$

In order to connect node $S$ to $nr$ replicas of node $D$, we have to use several nodes in between to gather data and then send it to each replica in round-robin order. Assume each node has FanIn/FanOut equal to $nf$, which means each node can send data to a maximum of $nf$ nodes. We define $H$, which shows how many layers of nodes need to send data from one node to $nr$ nodes.

$$H = \left\lceil \log_{nf}(nr) \right\rceil \tag{3.9}$$

Assuming $nr = nf^H$ ( Figure 3.10.b), the area overhead $A_O$ for connecting node

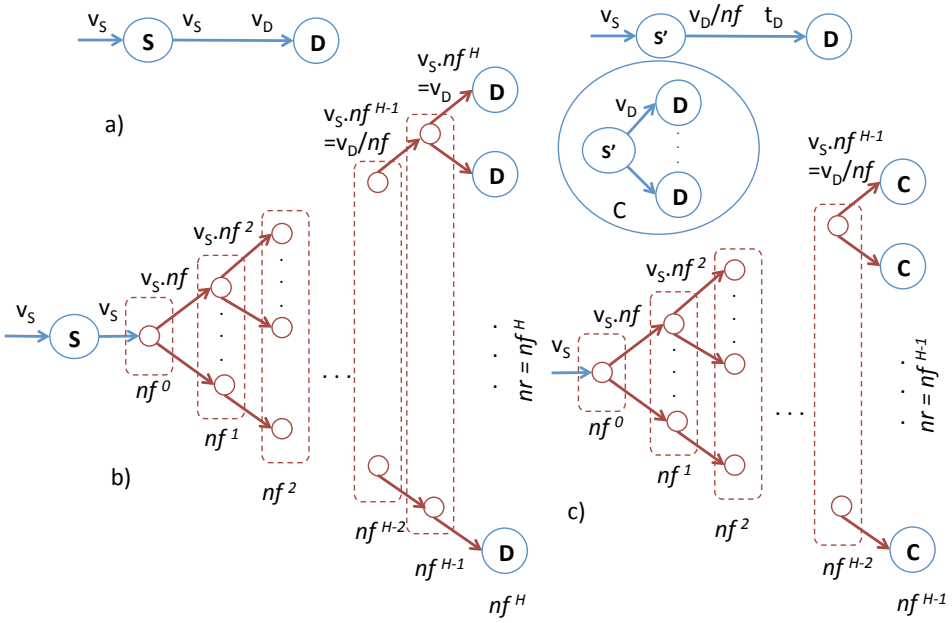


**Figure 3.10:** Node combining in Bottleneck Optimizer

$S$ to $nr$ replicas of node $D$ is calculated as

$$A_O = \sum_{i=0}^{H-1} nf^i \tag{3.10}$$

In our approach, we try to combine nodes together in order to save area overhead. As shown in Figure 3.10.b, in each layer $h$ there are $nf^{h-1}$ nodes with input inverse-throughput $\vartheta_{in}^h$ and output inverse-throughput $\vartheta_{out}^h$, which are calculated as

$$\vartheta_{in}^h = \vartheta_S \, nf^{h-1} = \frac{\vartheta_D}{nf^{H+1-h}} \tag{3.11}$$

$$\vartheta_{out}^h = \vartheta_{in}^h \, nf \tag{3.12}$$

So if we can find an implementation $S'$ of node $S$ with inverse-throughput equal to $\vartheta_{in}^h$, we can combine node $S'$ with $nf$ copies of node $D$ without any area overhead ( Figure 3.10.c) and name it node $C$ with input inverse-throughput $\vartheta_C$.

$$\vartheta_C = \frac{t_D}{nf} \tag{3.13}$$

To match the inverse-throughput of node $C$ to $\vartheta_C$ we have to make $nr'$ replicas of it with area overhead $A'_O$

$$nr' = \frac{nr}{nf} \tag{3.14}$$

$$A'_O = \sum_{i=0}^{H-2} nf^i \tag{3.15}$$

Assuming that inverse-throughput/area relation between node $S$ and node $S'$ is linear, we can save $nf^{H-1}$ nodes. For example in case $nf = 4$, more than 75% overhead area will be saved. As shown in this example it is possible to decrease area overhead by combining nodes, an approach that is computationally infeasible to model in the ILP formulation.

Now that the key modules of our heuristic approach have been defined, we will next explain how our heuristic approach works.

### Heuristic Approach Description

As mentioned before, the trade-off finding process has two modes. The user defines either an area target or a throughput target. Algorithm 1 shows pseudocode of the heuristic approach for a defined area target ($A_{tgt}$).

The Trade-off Finder heuristic starts by selecting an implementation with the highest throughput (lowest inverse-throughput) achieved per area unit for each node. It analyzes the full application and calculates the expected input inverse-throughput and minimum output inverse-throughput for each channel using Throughput Propagation. Then, it calculates slacks for all channels and weights for all nodes. Trade-off Finder finds the most critical bottlenecks as the nodes with the largest weights. Next, Trade-off Finder calculates the application area and available area for this implementation. Considering the defined area target, Trade-off Finder budgets the most critical bottleneck and propagates the throughput to other nodes. It continues budgeting other nodes considering propagated throughput. Af-

**Algorithm 1** Heuristic algorithm

$Application \leftarrow Compiler(sourcefiles)$
$A \leftarrow GlobalPartitioner(Application)$ ▷ $A = \{f_1, f_2, \ldots, f_N\}$
▷ Either using aStruct or MinCut for generating STG
**for** Each composite node $f_m$ **do**
    $MDP_m \leftarrow ParallelismFinder(f_m)$ ▷ Maximum degree of parallelism
    $P_m \leftarrow IntraNodeOptimizer(f_m, MDP_m)$ ▷ Different implementations
    $\tilde{P_m} \leftarrow InterNodeOptmizer(f_m)$
**end for**
$A_H \leftarrow SelectImplementationsWithHighestThroughput(\tilde{P})$
▷ $A_H = \{P_1^H, P_2^H, \ldots, P_N^H\}$ where $P_m^H = \{P_m^s \mid argmin(\frac{\vartheta_m^s}{A_m^s})\}$
**while** TRUE **do**
    **for** Each composite node $f_m$ **do**
        **if** $f_m$ and its successors and predecessors are visited **then**
            $(W_m) \leftarrow ThroughputAnalysis()$ ▷ Equation 3.5 and Equation 3.6
        **end if**
    **end for**
    Set of critical bottlenecks B $\leftarrow ThroughputPropagation()$ ▷ Equation 3.7
    **while** !$(\alpha_l A_{tgt} < A < \alpha_u A_{tgt})$ & applicationIsBalanced **do** ▷ Budgeting
        Budget the most critical bottleneck in the set
        Propagate the throughput
        Calculate estimated application area cost
    **end while**
    **while** (all Bottlenecks are visited) **do**
        $BottleneckOptimizer()$
        B $\leftarrow ThroughputPropagation()$
    **end while**
    $A_A \leftarrow AreaCost()$
    **if** $\delta < \frac{A_A}{A_{tgt}} < \beta$ **then**
        break
    **end if**
**end while**

ter budgeting all nodes, it calculates an approximate area cost for the application considering the new throughput for each node. Trade-off Finder accepts an area cost bigger than the target area on the chip within a margin. In other words, it overshoots and hopes to release area later in the process from fast nodes. If the approximate area cost is above the margin, Trade-off Finder decreases the target throughput budget and does the same procedure again.

After finding a budgeting which satisfies the targeted area, Trade-off Finder starts from the most critical bottleneck on the critical path (i.e., the path with the slowest throughput) and uses Bottleneck Optimizer to make replicas of that bottleneck to get better throughput. Trade-off Finder starts from the optimized bottleneck and goes toward the output until it reaches a node which is located on another critical path. After reaching this node, Trade-off Finder goes backward to visit the other bottleneck and uses Bottleneck Optimizer to match its throughput to satisfy the throughput expectations of other nodes. Note, the main idea is to prevent optimizing a bottleneck without addressing the expected inverse throughput for its predecessors. The process continues until it balances all the other nodes. Trade-off Finder sees the other nodes in breadth first search order and makes sure that each node doesn't affect nodes in other critical paths.

The Trade-off Finder use the same approach described above for a defined throughput target. The difference is in the budgeting process. The Trade-off Finder calculates an estimate application area cost based on the throughput target and budgets all the nodes based on that. Next, it goes through the Bottleneck Optimizer stage as before and makes sure all the bottlenecks are optimized in order to satisfy the throughput target. After finding an implementation which satisfies the throughput target, it budgets all the nodes again with a smaller area budget (considering the throughput per area unit achieved) and runs the process again. Next, it reduces or increases the overall area budget based on the success or failure of the last attempt to find alternative solutions. It continues this process until it finds the local optimum for the area cost function.

**Table 3.2:** Number of different implementations found by the tool for StreamIt benchmarks

| Benchmark | Number of different implementations |
|---|---|
| FFT | 34 |
| FIR | 42 |
| Radar Application | 73 |
| Filter Bank | 52 |
| FM Software Application | 39 |
| Vocoder | 92 |
| gsm | 82 |

## 3.4 Experimental Results

Our experiments are carried out in two steps. We first test our tool using StreamIt benchmarks [81]. Then, we examine implementation of a Joint Photographic Experts Group (JPEG) encoder produced using our tool.

### 3.4.1 StreamIt

In order to test our tool, 7 out of 9 benchmarks in the StreamIt benchmark set [33] are implemented as STGs using our Java-based programming model. For these test benchmarks, each benchmark is a single STG node. An architectural simulator has been implemented to validate the results generated by our tool. Our tool was able to find a number of different implementations for each benchmark with different area cost and throughput. Table 3.2 shows the number of different implementations found by our tool for the benchmarks. The functionality of all implementations has been verified with the simulator as well. Due to limited time, we didn't implement two of the benchmarks in the StreamIt benchmark set; the seven benchmarks used here were enough to evaluate the tool's ability to automatically find different implementations. Since the StreamIt benchmarks are small, we only use them here for verifying the front-end and finding different implementations. In the next section, we will use JPEG as our benchmark in order to examine different implementations provided by our Trade-off Finder.

**Table 3.3:** Implementation library for JPEG encoder

| module | Color Conversion | | | | DCT | | | | | Quantization | | | | | Encoding |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Version* | *v*1 | *v*2 | *v*3 | *v*4 | *v*1 | *v*2 | *v*3 | *v*4[2] | *v*5 | *v*1 | *v*2 | *v*3 | *v*4 | *v*5 | *v*1 |
| *InverseThroughput* | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 6 | 32 | 1 | 2 | 4 | 8 | 128 | 512 |
| *Area*[1] | 512 | 256 | 128 | 64 | 800 | 400 | 224 | 160 | 50 | 512 | 256 | 128 | 64 | 4 | 22 |

[1] Area units are the number of simple operations.
[2] The tool couldn't find an implementation with inverse throughput of 8, but it found a faster implementation instead.

**Table 3.4:** Heuristic vs ILP for many-core system

| Method | Inverse Throughput | Color Conversion | | DCT | | Quantization | | Encoding | | Fork/join Overhead | Total Area |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | impl | rep[1] | impl | rep | impl | rep | impl | rep | | |
| ILP | 1 | v1 | 1 | v1 | 1 | v1 | 1 | v1 | 512 | 10880 | 23968 |
| Heuristic | | v1 | 1 | v5 | 32 | v5 | 128 | v1 | 512 | 640 | 13888 |
| ILP | 2 | v2 | 1 | v2 | 1 | v2 | 1 | v1 | 256 | 5376 | 11920 |
| Heuristic | | v2 | 1 | v5 | 16 | v5 | 64 | v1 | 256 | 256 | 7456 |
| ILP | 4 | v3 | 1 | v3 | 1 | v3 | 1 | v1 | 128 | 2688 | 5984 |
| Heuristic | | v3 | 1 | v5 | 8 | v5 | 32 | v1 | 128 | 128 | 3600 |
| ILP | 8 | v4 | 1 | v4 | 1 | v4 | 1 | v1 | 64 | 1280 | 2976 |
| Heuristic | | v4 | 1 | v5 | 4 | v5 | 16 | v1 | 64 | 0 | 1736 |

[1] rep indicates how many replicas are used; overhead and area refer to the total number of simple operations.

### 3.4.2 JPEG

Figure 3.11 shows the block diagram of the JPEG compression algorithm [88]. The JPEG compression algorithm contains four major producer/consumer relationships shown as 4 blocks in the figure. The benchmark is written in this fashion, as four connected objects, with multiple loops and conditionals within each object. Despite this coarse level of implementation, our tool can break it down to a fully unrolled graph with one operation per node if necessary.

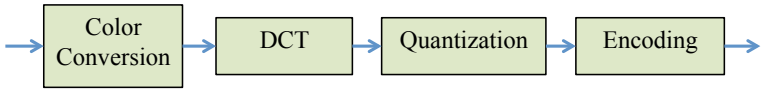


**Figure 3.11:** Node combining in Bottleneck Optimizer

The tool uses Intra-Node Optimizer and Inter-Node Optimizer modules on the STG, finding different implementations for each of them. In particular, our tool found 11 different implementations for "Color Conversion" and "Quantization" modules, 17 different implementations for "DCT", and only one implementation for "Encoding". Table 3.3 shows a selection of these implementations.

Both ILP and Heuristic approaches have been used by our tool in order to find

a trade off between area and throughput for different inverse throughput targets. Table 3.4 gives the results generated by these two approaches for given inverse throughput targets. We list the selected implementation and number of replicas for each module. The heuristic approach finds better area/throughput trade-off compared to the ILP approach. For example, for an inverse throughput target of 2, the heuristic approach used 37% less area compared to ILP. The ILP solver we use is GLPK [60] and the area cost unit is the number of primitive nodes, i.e. a simple operation such as add or subtract. A primitive node can be implemented with a very simple PE. This approach can also be used to map code to large processor arrays such as GRVI Phalanx [34]. Note, the complexity of PEs can be reduced while moving towards high throughput implementations. For example, in an implementation which all the loops are unrolled and parallelized, a primitive node can be a simple ALU (or a simple operator). This motivated us to explore space/time scaling for fine-grained architectures described in chapter 4.

## 3.5 Summary

This chapter investigates two ways of automatically finding area/throughput trade-off of streaming applications being mapped onto MPPA overlays. We introduce a new tool that compiles a streaming application written in Java, partitions it into composite nodes, finds all degrees of parallelism for each, finds different implementations for each node, and finally selects a good trade off between area and throughput. For optimization, we used both a classical ILP formulation as well as a novel heuristic. The heuristic combines module selection and replication methods with node combining and splitting in order to more quickly find a better area/throughput trade-off than what can be readily modelled in the ILP formulation; the ILP formulation can quickly become computationally infeasible. This approach has been verified with small designs in StreamIt and one larger design, a JPEG encoder. This approach can also be used to map code to large processor arrays. We stopped evaluating this approach after JPEG since our results showed the approach is reliable and it also can be more beneficial targeting fine-grained architectures. This leads us to investigate a better approach with OpenVX and FPGAs, described in chapter 4.

# Chapter 4

# FPGA Space/Time Scaling

In the previous chapter, we discussed how to automatically trade-off space vs time to implement an STG by targeting a given area budget or a given throughput budget. That architecture consisted of coarse-grained PEs.

In this chapter we consider a new approach that directly targets fine-grained parallelism available in FPGAs. We build a practical tool for automatically exploring space/time tradeoffs that is used together with a commercial HLS tool, Xilinx Vivado HLS [43, 96], for implementing Computer Vision (CV) applications.

## 4.1 Introduction

With the rise of FPGA-based CV applications, there is increasing need for a programming method that achieves the target throughput or area budget. HLS tools provide the opportunity to simplify design entry and debug. Unfortunately, a modern tool like Vivado HLS cannot automatically produce a range of implementations across the space/time spectrum from a single source file. However, back in 1994, in an attempt to address automatic space/time scaling, Synopsys Behavioral Compiler was introduced [46] with the promise that user can start with a high-level description containing few implementation details and the tool would handle resource allocation and scheduling automatically. Synopsys was not able to fully deliver and the project was shut down after 10 years. The idea of having a tool which automatically explores space/time tradeoffs motivated us to add this capa-

bility to the Vivado HLS tool.

In this chapter, we provide a framework for doing this with compute graphs specified in OpenVX, a C-based programming environment for computer vision. To do this, we build our own OpenVX system on top of Xilinx Vivado HLS [96], and add an algorithmic layer which allows the user to specify an area budget (while maximizing throughput) or a throughput target (while minimizing area).

Our OpenVX system consists of a series of compute kernels, prewritten in C++ for Vivado HLS and heavily parameterized. However, the key contribution in this chapter is an algorithm to automatically select from among these prewritten kernels. This is based upon the Intra-node and Inter-node Optimizers presented in chapter 3. With OpenVX, the implementations can also break an image down into multiple tiles, where the tile size is selected to maximize on-chip data reuse, thus improving delay and power. The runtime system must determine an appropriate tile size.

We evaluate the system on typical OpenVX benchmarks under a variety of fixed area constraints, and find that our system is able to automatically achieve between 92% and 100% of the target area utilization. We also evaluate the system with same benchmarks under variety of fixed throughput targets, and find our system saves up to 30% in area cost compared to manually parallelized implementations. Our heuristic approach is able to hit the same throughput targets and save 19% area on average compared to existing ILP approaches. In terms of efficient use of parallel resources on chip, the tool manages to satisfy different throughput targets, getting up to 5.5 GigaPixel/sec for the Sobel edge detection application on a small FPGA with only 53,200 LUTs and 220 DSP slices.

The most similar existing approach to ours is Xilinx's reVISION [95] which provides a set of hardware-accelerated OpenCV kernels. While existing methods can easily achieve a single design point, they are unable to automatically generate a set of solutions from the same source; a prominent capability embedded in our tool.

## 4.2 Approach

Existing methods of programming FPGAs with HLS require the user to explicitly manage resources at every stage in their algorithm in order to meet a specified area target or throughput target by manual loop unrolling or adding different "pragmas" to the code. Below, we describe a novel approach to explore the space/time trade-offs for OpenVX [45] compute graphs in order to find optimum solutions, meeting different area budgets or throughput targets.

We analyzed OpenVX kernels (nodes) to increase parallelism based on pipeline opportunities and different loop transformation strategies [55, 73]. After analyzing each kernel, we rewrote them in C++ while heavily parameterizing for HLS. Users can describe a CV computation as an OpenVX compute graph (STG) and then define either a throughput target, or an area budget. Our OpenVX system analyzes the compute graph and generates different implementations for each node with different area, IO and throughput characteristics by creating different HLS projects and passing them to Xilinx Vivado HLS. In order to get precise throughput/area information for different FPGA targets, and avoid implementations with deadlock, our tool automatically generates Vivado projects including a System-Verilog testbench for each implementation. In addition, our tool uses node replication and node combining to cover more possible solutions by either increasing the image tile size, or improving throughput for existing implementations.

Moreover, our tool uses the two internal optimization approaches discussed in chapter 3. The first, based upon ILP, is similar to previous work on task graph optimizations by Cong et al [22]. The second is a heuristic approach that we tuned and improved for implementing CV applications targeting FPGAs. Similar to chapter 3, we observed that although the ILP approach works well, maintaining the ILP optimization model within the tool precludes the use of certain optimizations. Instead, the heuristic approach is able to perform object coalescing which can be computationally infeasible in the ILP formulation. This leads to area saving and less runtime compared to the ILP approach.

There have been several recent studies on implementing image processing and OpenVX applications on FPGAs and exploring the area/throughput trade-off such as [78], [37] and [38]. These prior approaches either use a specific programming

model, which requires the user to learn a new programming language, or they implement a soft multi-core platform on the FPGA and then run the application on it. An OpenVX acceleration framework introduced by Taheri et al [79] can generate different hardware implementations by manually specifying the amount of pixel parallelism (different tile-size). Our approach is more general: to satisfy a given area target or throughput constraint, not only does it automatically explore different levels of pixel parallelism, but it can also automatically generate a variety of different implementations with different throughput or area cost at each level in order to meet the given constraint.

Next, we describe our proposed OpenVX based HLS system in detail.

## 4.3 Tool Flow for OpenVX-based HLS

Figure 4.1 illustrates the detailed flow of the proposed tool. It receives an OpenVX compute graph as an STG and analyzes it to obtain a matched kernel for each node. Each kernel is a heavily parameterized C++ function with Xilinx AXI-Stream [94] in/out arguments. Taking into account the user area budget or throughput target, our tool creates different Vivado HLS projects in order to generate different implementations for each kernel. To minimize the search space, our tool prunes the dominated points in the design space using an algorithm by Zhong et.al. [101].

Previous studies [57] have shown that the report generated by the HLS tool is not accurate and should therefore not be used for exploring the problem space. To obtain a precise throughput and area cost, our tool also generates Vivado projects using the HDL output generated from the HLS tool. Using Vivado design suite, the tool is able to find the throughput/area correlation for each kernel. This data will be eventually used later in the trade-off finding process. In addition, our tool uses a step called Intra-node Optimizer to generate more implementations to increase the trade-off solution space by filling gaps in the throughput/area solution space. Finally, the tool uses Trade-off Finder to find a good compromise between area and throughput. Below, we discuss all of these steps in detail.

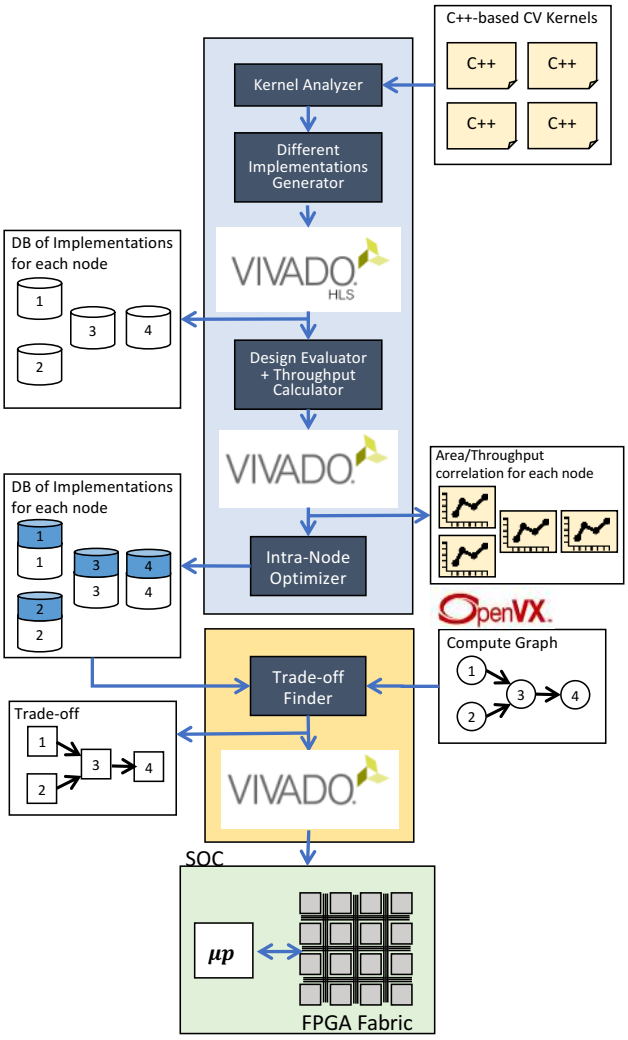


**Figure 4.1:** Tool flow

### 4.3.1 OpenVX Programming Model

OpenVX is a cross-platform, C-based API standard for Computer Vision. Most CV applications can be described as a set of vision kernels (nodes) which communicate through input/output data dependencies. OpenVX describes this set of vision kernels in a graph-oriented execution model based on DAGs. Figure 4.2 shows an OpenVX code example for the *Sobel* application and Figure 4.3 shows the corre-

```
//vxSobel3x3 example
vx_node nodes[] = {
  vxColorConvertNode(graph, rgb, gray),
  vxGaussian3x3Node(graph, gray, gauss),
  vxSobel3x3Node(graph, gauss, gradx, grady),
  vxMagnitudeNode(graph, gradx, grady, mag),
  vxPhaseNode(graph, gradx, grady, phase)
};
```

**Figure 4.2:** OpenVX source code

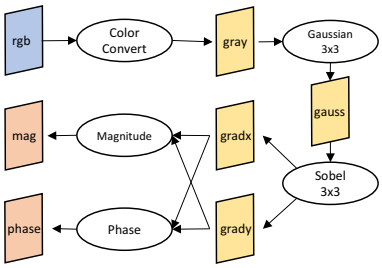


**Figure 4.3:** Sobel graph

sponding STG for it.

Since OpenVX compute graphs are DAGs, it makes them suitable candidates to be implemented as pipelined hardware accelerators on FPGAs. Below we discuss how our tool generates a variety of different implementations for those hardware accelerators. Then we describe our approach to find different implementations for each CV kernel.

### 4.3.2 Finding Different Implementations

Consider an application described as an STG with N nodes $f_1, f_2, ..., f_N$. For each node $f_m$ our tool tries to find different implementations $P_m^1, P_m^2, ..., P_m^{S_m}$ where each implementation $P_m^s$ can perform functionality of $f_m$ with area cost $A(P_m^s)$, number of pixels it can consume/produce $NP(P_m^s)$, and initial interval $II(P_m^s)$. For implementation $P_m^s$, the area cost on FPGAs is calculated as:

$$A(P_m^s) = w_{lut} \cdot LUT(P_m^s) + w_{dsp} \cdot DSP(P_m^s) + w_{bram} \cdot BRAM(P_m^s) \qquad (4.1)$$

where $LUT(P_m^s)$, $DSP(P_m^s)$ and $BRAM(P_m^s)$ are the LUT, DSP and BRAM count for implementation $P_m^s$. Note LUT weight ($w_{lut}$), DSP weight ($w_{dsp}$) and BRAM weight ($w_{bram}$) are different for various FPGA architectures and are chosen to reflect the relative size of each resource in silicon. We have set these weights to slightly different values for Xilinx, Altera and VPR architectures.

For node $f_m$ and its implementation $P_m^s$, input "inverse throughput" $\vartheta_{in}(P_m^s)$ and

output inverse throughput $\vartheta_{out}(P_m^s)$ for each input/output are calculated as:

$$\vartheta_{in}(P_m^s) = \frac{II(P_m^s)}{In(f_m)}, \vartheta_{out}(P_m^s) = \frac{II(P_m^s)}{Out(f_m)} \tag{4.2}$$

where $In(f_m)$ and $Out(f_m)$ are the number of data tokens that $f_m$ consumes on the input data channel and produces on the output data channel during each firing, respectively. Note that inverse throughput shows the number of cycles to consume/produce per datum in its input/output channel. For most CV kernels, their input/output channels have matched inverse throughput, $\vartheta_{IO}(P_m^s)$. Kernel throughput is number of pixels consumed/produced in each clock cycle:

$$\Theta(P_m^s) = \frac{NP(P_m^s)}{\vartheta_{IO}(P_m^s)} \tag{4.3}$$

The Different Implementation Generator (DIG) module in our tool automatically finds the above mentioned implementations of each node using the heavily parameterized C++ based kernels. The DIG needs to be able to automatically find a wide range of different implementations to cover the solution space as much as possible.

To have a better understanding of the complexity of this problem and the variety of possible solutions, let's look at the simple example of a 3-node graph shown in Figure 4.4a. Figure 4.4.b and Figure 4.4.c show two different approaches to satisfy a target throughput of 5; one reads 20 pixels and picks implementations with $II = 4$ for each node, the other reads 5 pixels and picks implementations with $II = 1$. Further, Figure 4.4.c shows another approach for node $f_2$ in which instead of picking an implementation with $NP = 5$, it picks two implementations with $NP = 3$ and $NP = 2$ and splits the data between them. Figure 4.4.b and Figure 4.4.c are just two examples of various instances of $II$, $NP$ and splitting/joining nodes which can be utilized to find the solution. In addition, the strategy of reading image data from the main memory can vary for different implementations when $NP$ and $II$ change. The strategy impacts DMA configuration and data alignment network design, which leads us to a different overhead cost for each.

The above-mentioned example shows that for every area budget or throughput target, there are a variety of different acceptable solutions. This makes the trade-off

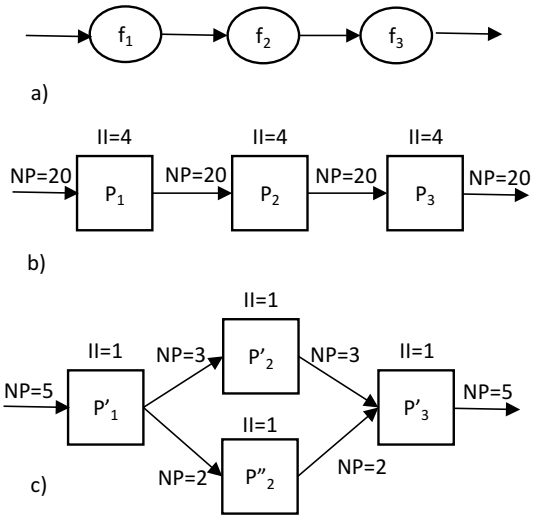


**Figure 4.4:** Two different approaches for satisfying $\Theta = 5$

finding problem a rich design space. It also shows the importance of generating a variety of different implementations with different *NP*, *II* and area costs to cover the solution space as much as possible.

### 4.3.3 CV Accelerator on FPGA

Before describing the tool flow in more detail, it is beneficial to go through the overall system description first. As mentioned earlier, the main goal of this study is to automatically find a good area/throughput trade-off for CV applications by generating different implementations which is done through changing the image tile width and/or using different function implementations inside the kernel.

Figure 4.5 gives a high-level system view of a CV accelerator implemented on Xilinx FPGAs. The host processor (i.e., ARM based processor [88]) is responsible for configuring DMA (e.g., Xilinx AXI DMA IP core [98]) to read/write image data as vertical strips from the main memory. On the other end, DMA sends/receives image data to/from the accelerator using the AXI-Stream protocol. Since the DMA data-width should be a power of two and the CV accelerator may read data at a different width in pixels, there needs to be a Data Alignment Network implemented

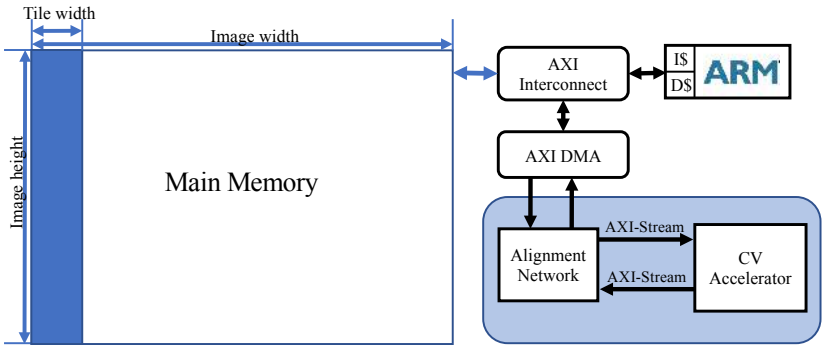


**Figure 4.5:** System view implemented on Xilinx FPGA

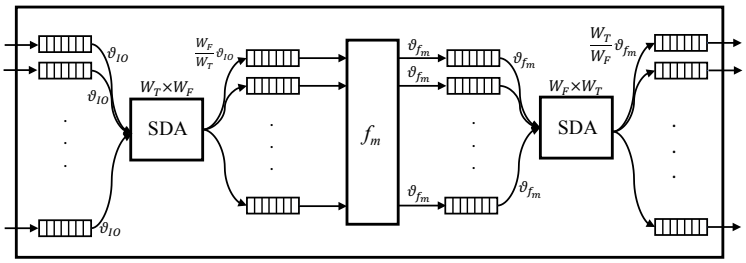


**Figure 4.6:** Internal view of a general node in CV hardware accelerator

as mixed-width FIFOs in between to align the data sent back and forth between DMA and accelerator.

Figure 4.6 provides an internal view of a general node in a CV hardware accelerator. Representing the image tile width with $W_T$, a general node $m$ with implementation $P_m^s$ consumes $W_T$ pixels ($NP_{in}(P_m^s) = W_T$) as stream-in and generates $W_T$ pixels as stream-out with inverse-throughput equal to $\vartheta_{IO}$. Since the hardware function inside the node might consume/produce a different number of pixels, two Stream Data Adjuster (SDA)s are added to either end. Assuming the hardware function can consume/produce $W_F$ pixels in its input/output channels, the input SDA should get $W_T$ pixels from the input and pass $W_F$ pixels to the function with inverse-throughput equal to $\frac{W_F}{W_T}\vartheta_{IO}$. On the other end, the output SDA gets $W_F$ pixels with inverse throughput $\vartheta_{f_m}$ and provides $W_T$ pixels with inverse-throughput equal to $\frac{W_T}{W_F}\vartheta_{f_m}$. As Figure 4.6 shows, four layers of FIFO are added in between in order to match different throughputs in various stages. To prevent data loss, FIFO

depths should be carefully selected.

### Stream Data Adjuster in more detail

SDA can deal with two different types of kernels; *Pixel*2*Pixel* kernels and *Window*2*Pixel* kernels.

*Pixel*2*Pixel* kernels such as *vxConvertColor* produces one pixel for each pixel received:

$$\tilde{P_{i,j}} = f(P_{i,j}) \tag{4.4}$$

Figure 4.7 shows how the SDA functions as an adjuster for a simple Pixel2Pixel kernel which receives/produces 4 pixels every 2 clock cycles and its hardware function consumes/produces 2 pixels in every clock cycle. In order to match the stream rate between IO and the function, SDA simply uses upstream to downstream transformation by splitting data in its input and sending it to the function. On the other end it joins data coming from the function and sends it to the output. In this example $\frac{W_T}{\vartheta_{IO}}$ is equal to $\frac{W_F}{\vartheta_{fm}}$, so it only needs to capture $W_T$ pixels in its $2 \times 2$ FIFO every two clock cycles.

*Window*2*Pixel* kernels such as *vxSobel*3*x*3 consumes a window of pixels for every produced output pixel as follows:

$$\tilde{P_{i,j}} = f(\begin{bmatrix} P_{i-\delta,j-\delta} & \cdots & P_{i-\delta,j} & \cdots & P_{i-\delta,j+\delta} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ P_{i,j-\delta} & \cdots & P_{i,j} & \cdots & P_{i,j+\delta} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ x_{j+\delta,j-\delta} & \cdots & P_{i+\delta,j} & \cdots & P_{i+\delta,j+\delta} \end{bmatrix}), \delta = \tfrac{w-1}{2} \tag{4.5}$$

A kernel with an image tile width of $W_T$ and filter window size of $w \times w$ consumes $W_T + w - 1$ pixels and produces $W_T$ pixels in every firing. For each firing, it slides down an image by one row, thus producing a vertical stripe of output, $W_T$ pixels wide.

Figure 4.8 shows how SDA handles the stream adjustment for a Window2Pixel kernel with $W_T$ equal to 4, $W_F$ equal to 2 and filter window size equal to $3 \times 3$. This kernel receives 6 pixels and produces 4 pixels every 2 clock cycles. The hard-

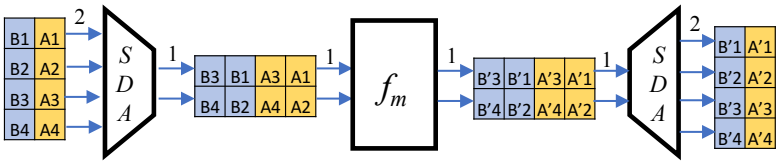


**Figure 4.7:** Pixel2Pixel kernel example, $W_T = 4, W_F = 2$

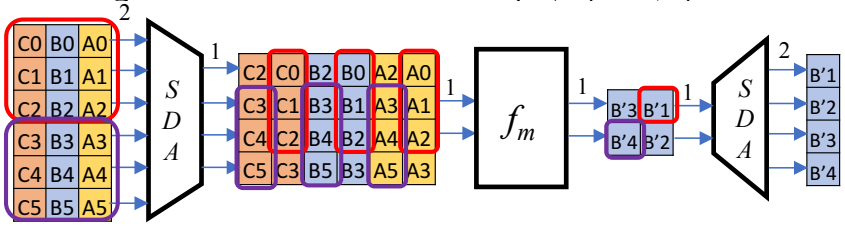


**Figure 4.8:** Window2Pixel kernel example, $W_T = 4, W_F = 2$

ware function produces 2 pixels every clock cycle which means it needs to get 4 pixels every clock cycle. In this case, SDA splits the input stream data maintaining some data overlap. This data overlap has two consequences: overhead of 2 duplicated pixels for every 6 pixels consumed by the kernel in each firing, and the need for a line-buffer with minimum depth of 5. Figure 4.9 illustrates a general Window2Pixel kernel with an image tile width of $W_T$ and a filter window size of $3 \times 3$ with $W_F$ equals to $\frac{W_T}{N}$. Since the function needs to receive $\frac{W_T}{N} + 2$ pixels in its firing, the overhead is $2N$. Also, to produce the first output, the function needs to have a line buffer with a minimum depth of $2N + 1$.

### 4.3.4 Heavily Parameterized C++-based OpenVX Kernels

A set of heavily parameterized C++ based OpenVX kernels with AXI-Stream input/output have been implemented to generate different implementations for each kernel. Each kernel is parameterized at two levels, the IO level and the core level. The number of pixels that a kernel consumes/produces in its IO as well as the number of pixels needed to provide/gather to/from its core are parameterized in the IO level. For each kernel, the main core function has been manually analyzed to find all degrees of parallelism and then heavily parameterized. This can be done by

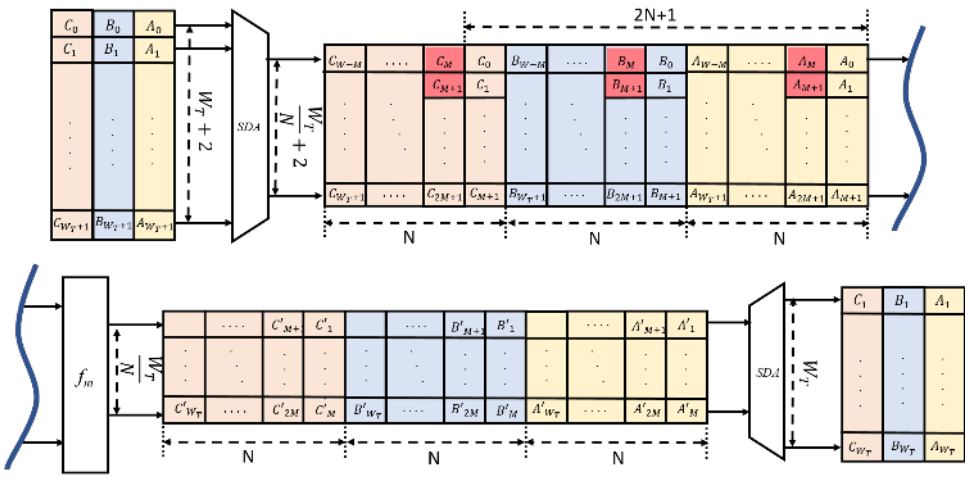


**Figure 4.9:** Window2Pixel kernel

labeling all loops and generating a set of suitable *pragmas* saved as a JSON file for each kernel. Considering this parameterization, the tool can generate different implementations with different number of inputs ($NP(P_m^s)$), area cost ($A(P_m^s)$) and initiation intervals ($II(P_m^s)$). Figure 4.10 shows the area, initiation interval and tile-width (number of pixels input) correlation of different implementations for the *Gaussian*3*x*3 kernel. Each dot represents an implementation.

### 4.3.5 Intra-node Optimizer

In addition, an Intra-node Optimizer step in the tool generates a wider range of implementations. Intra-node Optimizer replicates and combines existing implementations in order to fill gaps in the solution space. Node replication can be used to either increase the throughput or widen the tile-width. Figure 4.11.a demonstrates a general *Pixel*2*Pixel* kernel with inverse throughput $\vartheta_{IO}$ and tile-width $W_T$. In order to improve the throughput (reduce the $\vartheta_{IO}$), the tool replicates the node and sends data to each replica with a round-robin order. Figure 4.11.b shows how the tool improves the throughput $N$-times by making $N$ replicas of the original kernel. Figure 4.11.c shows how the tool increases the tile-width by replicating the kernel. Because of data dependencies in *Window*2*Pixel* kernels, replication can only be used for increasing tile-width. Our tool replicates *Window*2*Pixel* kernels considering the window size and handles the data passing. Figure 4.12 demonstrates

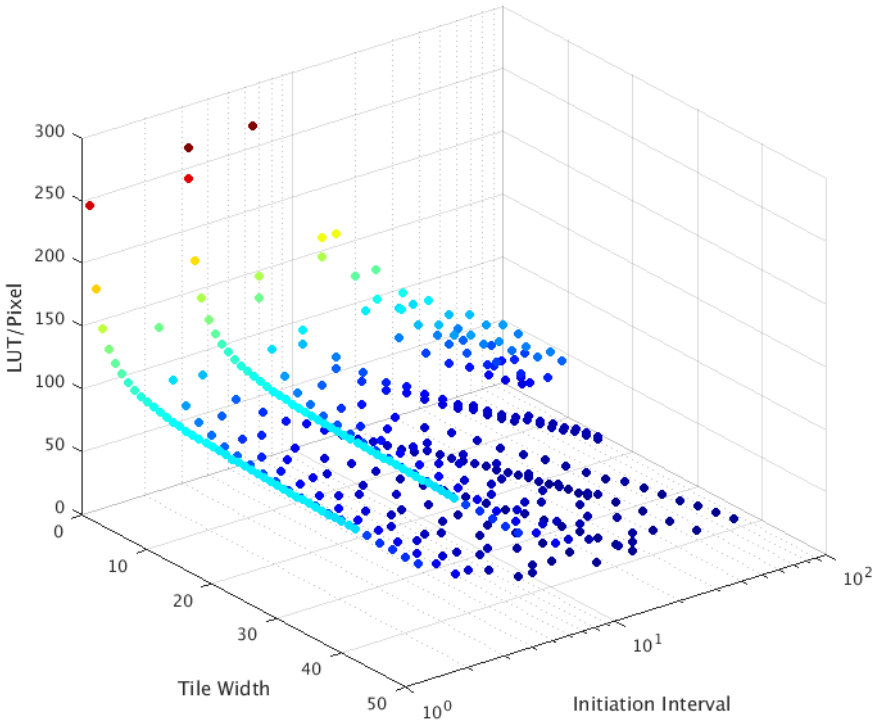


**Figure 4.10:** Area, throughput and tile-width correlation for *Gaussian*3*x*3 kernel

how the tool passes data to each replica when windows must overlap, e.g. in 2D convolutions.

### 4.3.6 Inter-node Optimizer

Moreover, we also have added the capability of Inter-node Optimizer which combines existing implementations and then replicates the combined node on the fly. Figure 4.13 shows a simple node combining example. Assume node $f_n$'s throughput is N times bigger than node $f_m$'s throughput. Two different approaches are shown in Figure 4.13 to match the throughputs: the first approach is replicating node $f_m$, $N$ times and using a $1 \rightarrow N$ data splitter so that node $f_n$ can send data to those replicas in a round-robin order (Figure 4.13.a). In the second approach

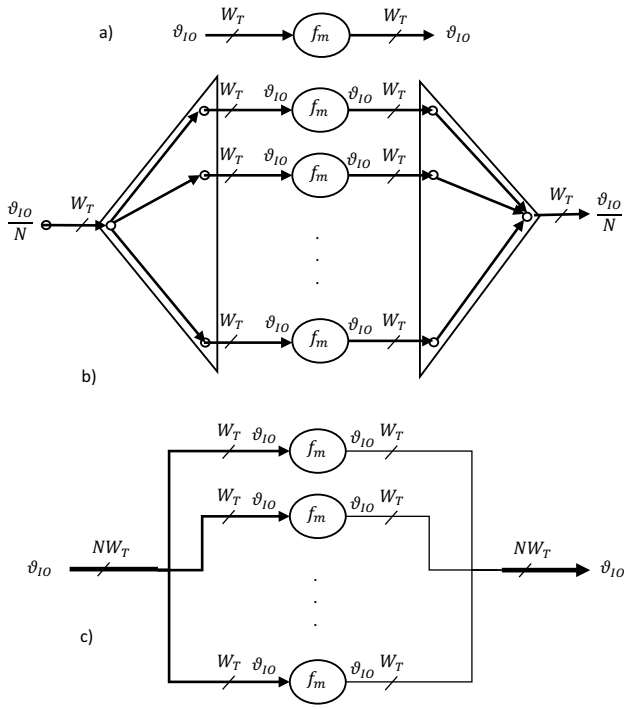


**Figure 4.11:** Pixel2Pixel replication

shown in Figure 4.13.b, another implementation for node $f_n$ with a throughput equal to twice node $f_m$'s throughput is found ($f'_n$). Then the nodes $f'_n$ and $f_m$ are combined and the combined node is replicated $\frac{N}{2}$ times. Note that second approach needs a $1 \rightarrow \frac{N}{2}$ data splitter and is much smaller than the $1 \rightarrow N$ data splitter in the first approach.

All the above mentioned techniques are used in Intra-node and Inter-node Optimizer steps to find a wide range of implementations (either in the pre-synthesis step or during tradeoff finding process) for each kernel which widens the solution space for the area/throughput scaling problem. Below we discuss our trade-off formulation and solutions.

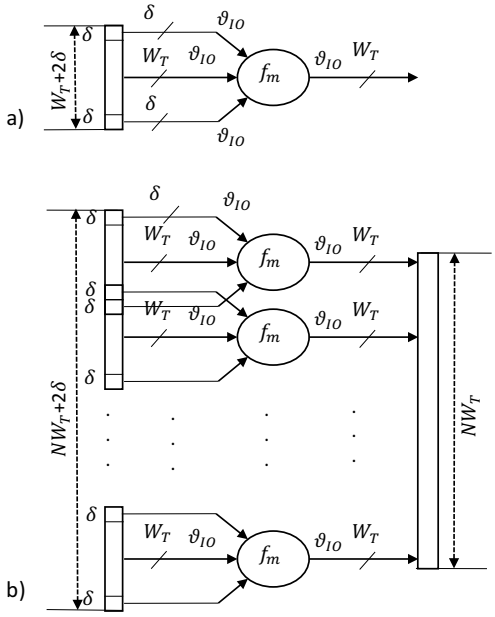


**Figure 4.12:** Window2Pixel replication

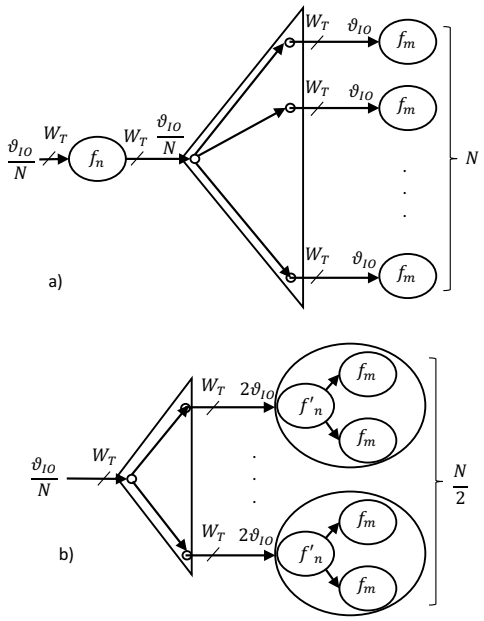


**Figure 4.13:** Node combining

### 4.3.7 Trade-off Finding Formulation and Solutions

To solve the described trade-off finding problem we used an ILP approach as well as a heuristic approach. For each of these approaches, there are two different modes or objectives:

- Given an area target $A_{tgt}$ and different implementations for each node $f_j$, which implementation $P_j^i$ should be selected and how many replicas $nr_j^i$ are needed in order to maximize application throughput $\Theta_A$ subject to the constraint the application area cost $A_A$ is not bigger than $A_{tgt}$.

- Given a throughput target $\Theta_{tgt}$, and different implementations for each node $f_j$, which implementation $P_j^i$ should be selected and how many replicas $nr_j^i$ are needed in order to minimize area cost $A_A$ subject to the constraint the application throughput $\Theta_A$ is bigger than $\Theta_{tgt}$.

#### Integer Linear Programming Algorithm

This problem can be defined as an ILP model similar to the previous chapter's formulation and solved with GLPK; it goes through all the possibilities in the solution space and finds the optimum solution, subject to the constraints.

Similar to the previous chapter, we can define an ILP problem formulation for both modes. The goal is to find binary integers $x_{j,1}, x_{j,2}, ..., x_{j,S_m}$ indicative of the implementations to be selected, and integer $nr_j^i$ indicative of the number of replicas needed. Equation 4.6 shows the formulation of the finding an implementation with maximum throughput for an area budget.

Maximizing $\Theta_A$ subject to:

$$\forall j \in \{1,...,N\} : \sum_{j=1}^{N} \sum_{i=1}^{S_m} nr_j^i A(P_j^i) x_{i,j} < A_t gt \ and \ \sum_{i=1}^{N} x_{i,j} = 1 \tag{4.6}$$

Equation 4.7 shows the formulation of finding an implementation with minimum area cost for a given throughput target. The formulation of the second problem is:

Minimizing $A_A$ subject to:

$$\forall j \in \{1,...,N\} : \sum_{i=1}^{S_m} \frac{NP(P_j^i)\vartheta(P_j^i)}{nr_j^i} > \Theta_{tgt} \; and \; \sum_{i=1}^{N} x_{i,j} = 1 \tag{4.7}$$

**Heuristic Algorithm**

To overcome the shortcomings of ILP approach mentioned in subsection 3.3.1, we also used a heuristic approach similar to the approach we discussed in subsection 3.3.2. We use throughput analysis and throughput propagation as well as node replication and node combining to find space/time tradeoffs for CV applications on FPGAs for a defined throughput target or a defined area budget.

## 4.4 Experimental Results

Our experiments are carried out in two steps. First, we evaluate our strategies of finding a good area/throughput tradeoff by targeting different FPGA architectures with an area target. Then we evaluate our tool by setting different throughput targets.

In our evaluation, we utilize the following benchmarks implemented as OpenVX compute graphs:

- *Sobel*, a Sobel-filter based edge detection with 5 nodes.
- *Canny*, a Canny edge detector with 6 nodes.
- *Harris*, a Harris corner detector with 6 nodes.

All the kernels inside each of the aforementioned benchmarks are analyzed and rewritten as parameterized C++ based kernels with stream-in/stream-out arguments. Using our tool, a library of different implementations for each kernel is generated. To examine whether our tool can cover a wide range of FPGA sizes, we evaluated it with VPR [11], a part of the academic Verilog-To-Routing project [59]. Using VPR, different size FPGAs were generated based on Altera Stratix IV [41], with logic cluster size $N = 10$, look-up table size $K = 6$, and channel segment

length $L = 4$. Then we passed each FPGA's size as an area budget to our tool. Figure 4.14 shows the percentage of LUTs used for implementing *Sobel* on different device sizes. As shown, our tool was able to automatically find suitable implementations for different architecture targets and fill over 95% of the chip area on average.

Figure 4.15 shows how efficiency scales with FPGA size in terms of LUTs required per unit of throughput. For smaller FPGAs, there is some overhead, so up to 50% more LUTs are required to achieve the same throughput. This overhead quickly reduces as FPGA size is scaled.

We also evaluated our tool with different Xilinx FPGAs. Figure 4.16 shows the percentage of LUTs used for implementing *Sobel* and *Harris* benchmarks on different Xilinx 7 series devices [97]. As shown, our tool was able to automatically find suitable implementations for different architecture targets and fill over 97% of the chip area on average.

Finding an optimum implementation for different throughput targets was the second goal. To evaluate that, we tested our tool by setting different throughput targets for different benchmarks and compared our tool to a fully manual HLS approach. Since we implemented heavily parameterized kernels for our OpenVX approach first, we learned which parallelization strategies worked better. Using that knowledge, we generated a manual HLS version. Due to limited time (as all designers will experience), we had to choose just a handful of implementation strategies which gave similar optimal area and throughput as our tool. However, to achieve designs with throughputs in between the optimal points, these implementations were scaled in the most logical way possible, but as they were scaled, they became less efficient; naturally they used more area as we moved further away from the optimal design points. Figure 4.17 shows how our tool covers a large design space and hits all targets more efficiently for the *vxMagnitude* kernel. We compared our tool results for different CV kernels with our manual HLS and observed that our approach found an implementation for each throughput target that saves up to 30% area.

Figure 4.18 shows area per throughput, normalized by the median value for each benchmark, for a range of throughput targets. As shown, for throughput targets larger than roughly 5 Pixel/clk, our tool finds good area/throughput tradeoffs

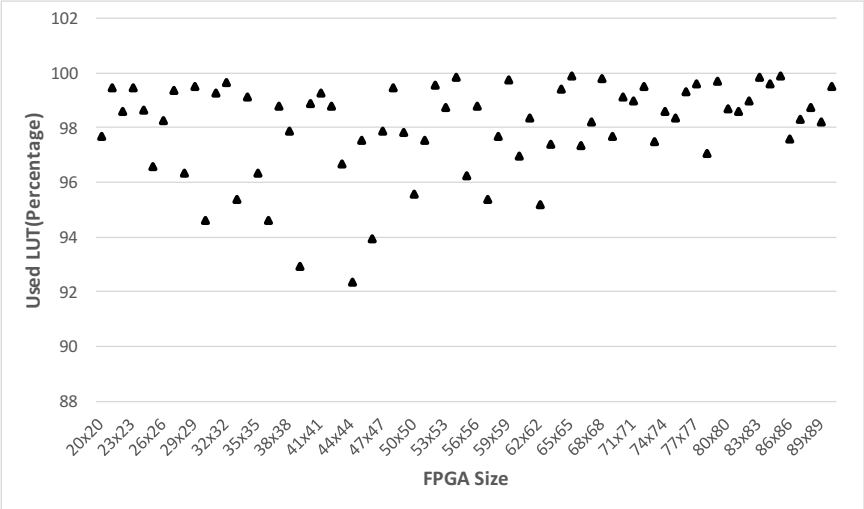


**Figure 4.14:** LUT usage percentage for Sobel implementations on different FPGA sizes

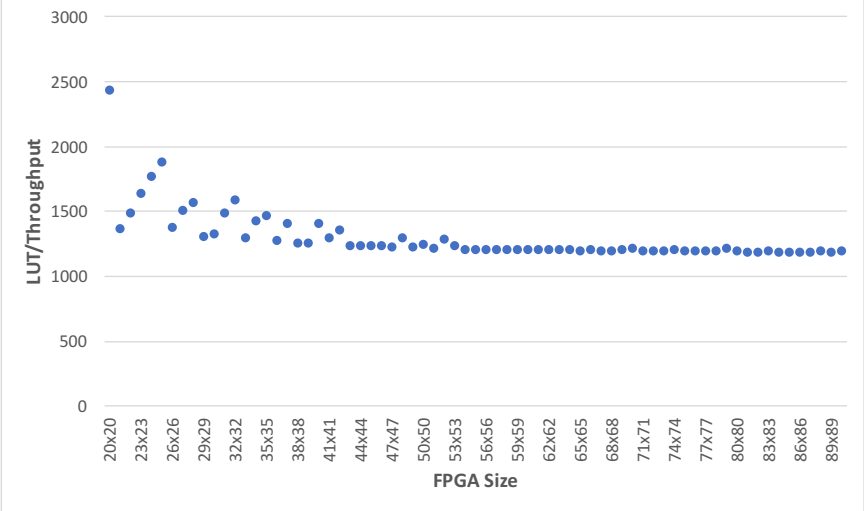


**Figure 4.15:** Throughput achieved for Sobel on different FPGA sizes

for each throughput target. For throughput targets less than 5, line buffer and SDA overhead became a big portion of the area cost and increased the area per throughput ratio.

To solve the trade-off finding problem, we used both ILP (similar to prior work by Cong et al. [22]) and the heuristic approaches. We also evaluated both ap-

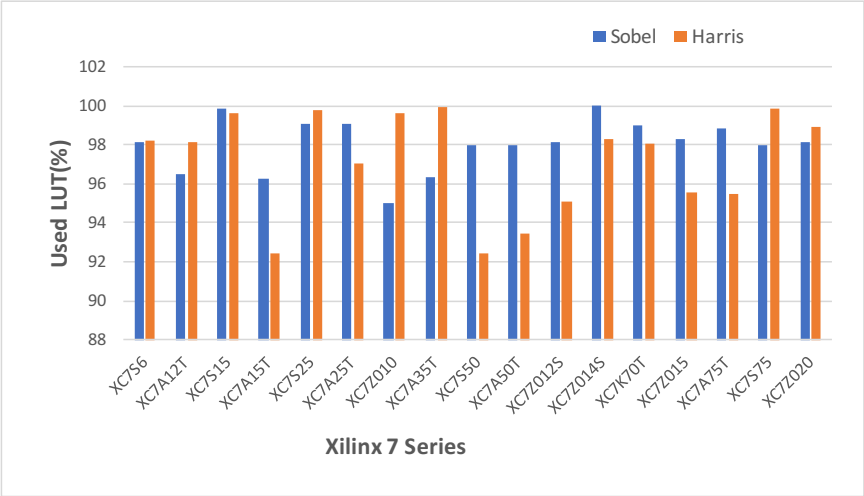


**Figure 4.16:** Percentage of LUT usage for different Xilinx FPGAs

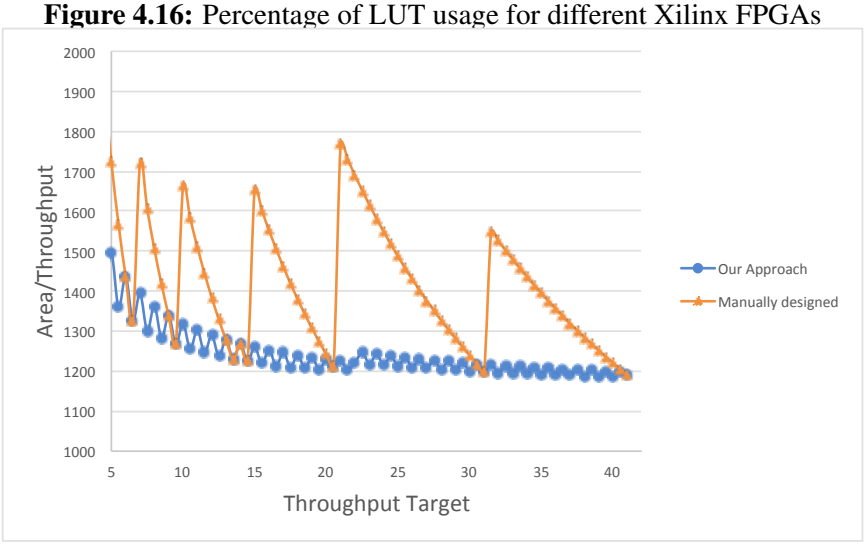


**Figure 4.17:** *vxMagnitude* Area/Throughput results for different throughput targets

proaches with different Xilinx 7-series FPGAs. Both heuristic and ILP approaches could fill over 95% of the chip area on average. The heuristic approach, however, improves the runtime up to 3.6x compared to the ILP approach. Figure 4.19 shows the runtime results for both heuristic and ILP approaches for implementing *Harris* application on different Xilinx FPGAs. Although the results are no shown, the heuristic was also able to fill closer to the area target on average.

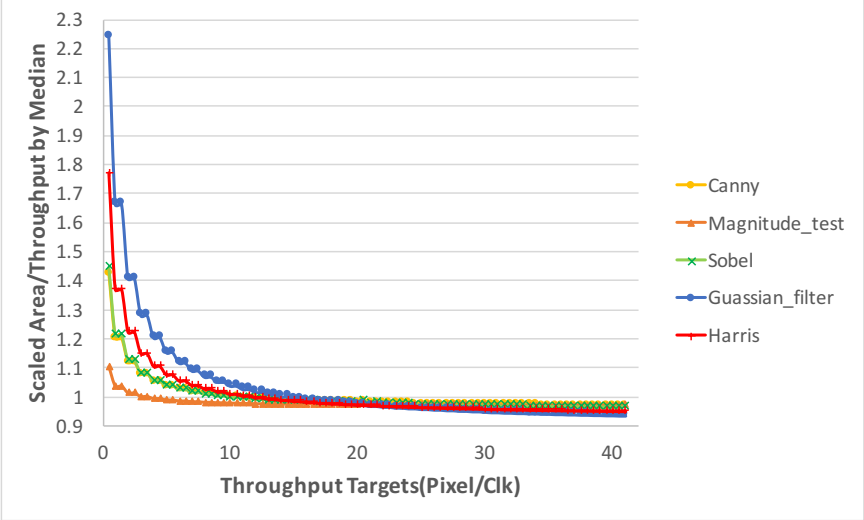


**Figure 4.18:** Area cost results for different throughput targets

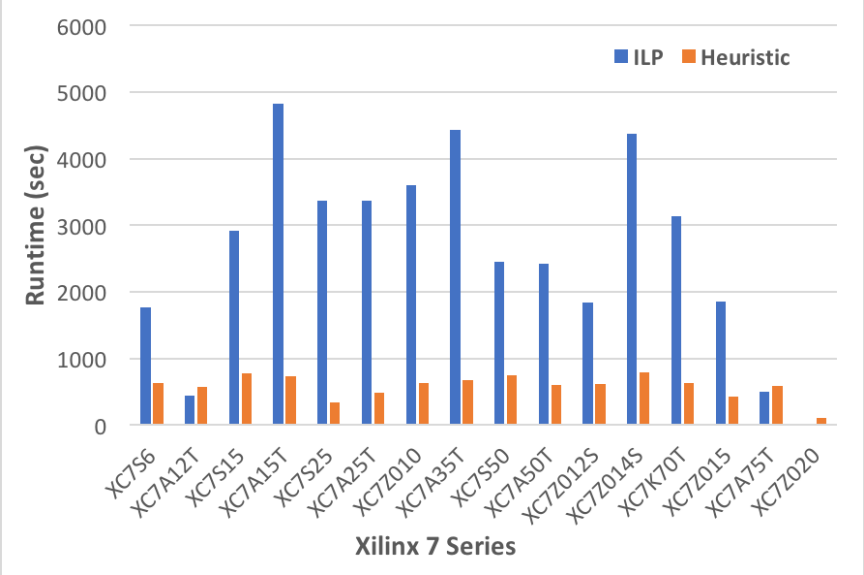


**Figure 4.19:** Heuristic vs ILP runtime speedup for Harris corner detection

To better demonstrate the ability of the heuristic to save area, we first used the ILP approach to meet the area target for *Harris*. Then, we used the ILP's achieved throughput on each FPGA size as a throughput-target for the heuristic approach. Figure 4.20 shows the area cost comparison between the heuristic and the ILP. The heuristic approach saves 19% area on average while decreasing the throughput by

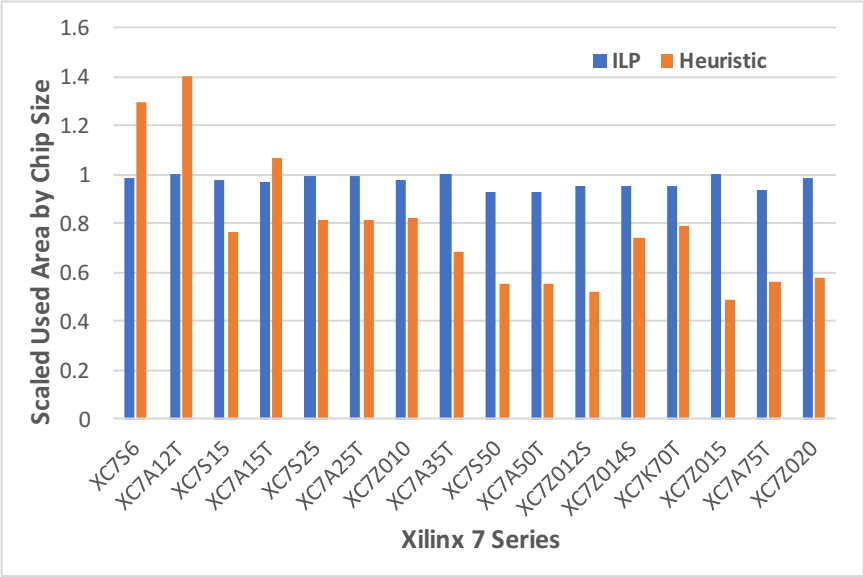


**Figure 4.20:** Area cost results for Harris using Heuristic and ILP approaches

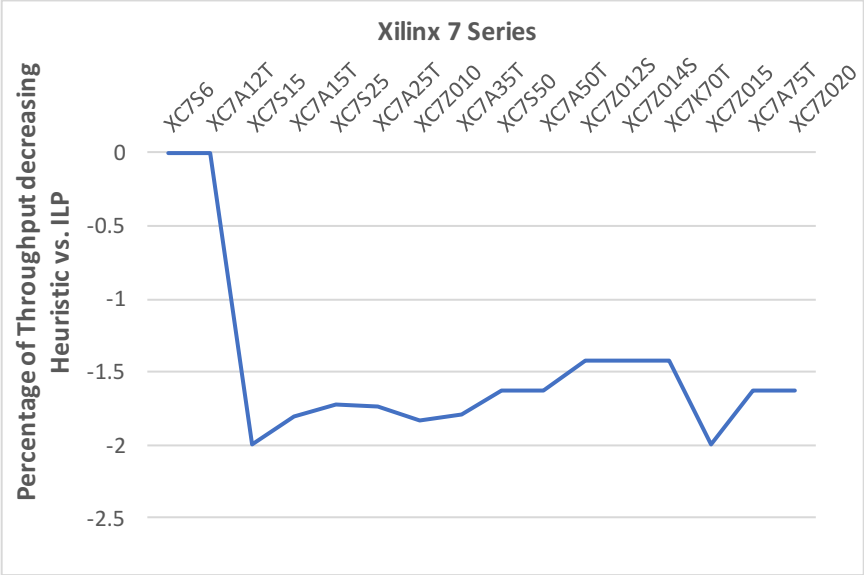


**Figure 4.21:** Heuristic vs ILP throughput results for Harris corner detection

less than 2%. Unfortunately, however, it also failed to fit for three of the FPGA sizes. The corresponding (small) drop in throughput is shown in Figure 4.21.

Next, to evaluate the real performance of the tool, we used the ZedBoard de-

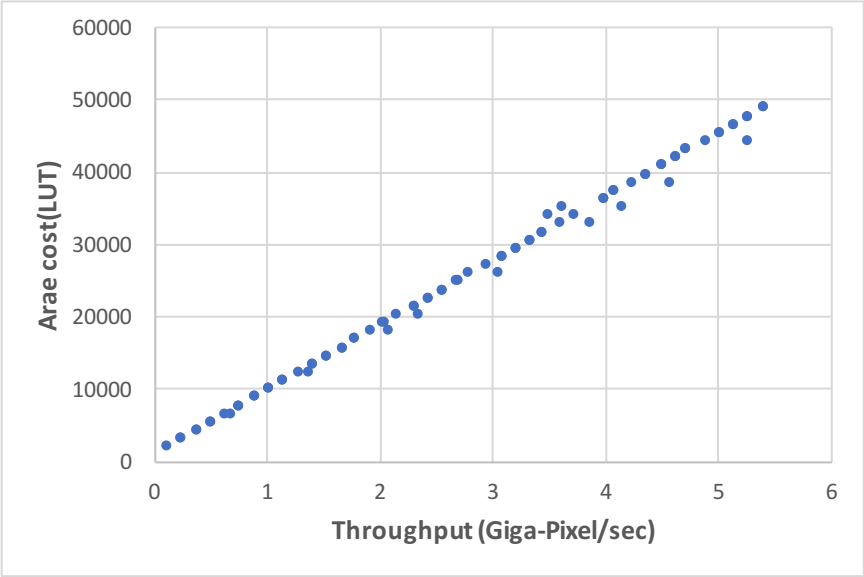


**Figure 4.22:** Area/throughput results for implementing Sobel on Xilinx Zed-Board

velopment platform [3] and set up different throughput targets for implementing *Sobel*. Figure 4.22 shows the results. As shown, our tool is able to hit all the throughput targets while increasing the area cost linearly. The tool is able to achieve up to 5.5GigaPixel/sec throughput for *Sobel* running at 105MHz. Since the *Sobel* benchmark requires about 70 operations per pixel, this is equivalent to roughly 350 GOPS.

## 4.5 Summary

In this chapter, we studied the problem of automatically finding an area/throughput trade-off of CV applications by mapping OpenVX compute graphs onto FPGAs. We proposed a framework on top of the Xilinx Vivado HLS tool which receives C++ based CV kernels and uses different approaches in order to find many different implementations for each kernel. It compiles an OpenVX compute graph, analyzes it and finds a good trade-off between area and throughput. Our approach is differentiated from the existing HLS approaches as 1) it automatically investigates finding different implementations, and 2) it combines module selection and repli-

cation methods as well as changing tile-size width, node combining, and splitting in order to automatically find a better area/throughput tradeoff. This approach was verified with different OpenVX benchmarks targeting several different FPGA sizes. Our tool is able to automatically achieve over 95% of the target area budget on average while maximizing the throughput. Our tool also can automatically satisfy a variety of throughput targets while minimizing the area cost. The proposed system saves up to 30% of the area cost compared to manually written and heavily parallelized implementations. Using Inter-node Optimizer step, our heuristic tradeoff finder is able to hit the same throughput targets as the ILP algorithm while saving 19% area on average.

# Chapter 5

# FPGA Overlay Space/Time Scaling with Custom Instructions

So far, this dissertation has investigated space/time scaling to implement STG applications on MPPA and FPGA architectures. In this chapter, we investigate an FPGA-based overlay architecture for accelerating OpenVX applications without creating any application-specific logic. The overlay consists of a Soft Vector Processor (SVP) for general acceleration, and the ability to implement Vector Custom Instruction (VCI) on the fly using a Partial Reconfiguration Region (PRR). The PRR is a reserved area of the chip that is initially blank and made available at run-time to allow instances of VCIs to be defined as they are needed. A pipeline of compute kernels can sometimes be realized by chaining multiple VCI together using a custom multiplexer network within the PRR. Using PR, this method obtains speedups far beyond what a plain SVP can accomplish. For example, on the *Canny-blur* application, an 8-lane SVP is 18 times faster than the plain ARM Cortex-A9. However, using ultra-fast PR, which is technically feasible but not yet supported on modern FPGAs, a speed of 106 times faster is possible. This allows OpenVX programmers, who have no FPGA design knowledge, to achieve hardware-like speeds within their vision application.

**Table 5.1:** *vxMagnitude* kernel throughput running on different platforms

| Running Platform | Throughput (MegaPixel/Sec) | Speedup vs ARM |
|---|---|---|
| ARM Cortex-A9 (667MHz) | 10.31 | 1.0 |
| VectorBlox SVP-V4 (100MHz / 12,989 LUTs) | 65.54 | 6.3 |
| VectorBlox SVP-V8 (100MHz / 22,517 LUTs) | 128.92 | 12.5 |
| Custom hardware (100MHz / 3,000 LUTs) | 1176.04 | 114 |

## 5.1 Introduction

Previously in chapter 4, we augmented Vivado HLS tools to implement CV applications described using OpenVX with maximum throughput for a given area target, or with minimum area for a given throughput target. These solutions are custom-generated for the OpenVX application, and require running the full Xilinx place-and-route tool flow to generate the solution. While they are highly optimized in terms of performance and area, the overall design flow still requires the expertise of hardware designers who are familiar with FPGAs.

In this chapter, we wish to consider accelerating OpenVX applications on FPGAs by software developers with no hardware experience, while still exploiting the ability to make space/time tradeoffs. The simplest implementation method would simply use a host processor – either a fast hard ARM processor present on many modern FPGAs, or a slower soft processor built using FPGA logic. To provide scalable performance for area, an SVP implemented in the FPGA logic achieves greater performance per unit area than multi-core soft processors [76].

Unfortunately, the total performance per unit area of an SVP still falls short of what can be achieved with the dedicated hardware generated in chapter 4. For example, consider Table 5.1 which shows throughput achieved with the *vxMagnitude* OpenVX kernel on different platforms. The vectorized software implementation running on a SVP with four vector lanes (V4) uses about 13,000 LUTs and achieves 6.3 times higher throughput compared to a basic software implementation running on the Cortex-A9. Note the SVP achieves this high throughput while running at a frequency that is approximately one-sixth that of the ARM processor, making it an efficient platform. Increasing the number of vector lanes to 8 nearly doubles the speedup to 12.5 and uses 23,000 LUTs. However, a custom hardware implementation of the *vxMagnitude* kernel that is area-constrained to just 3,000 LUTs achieves 114 times higher throughput than Cortex-A9.

While it may appear that custom hardware is the way to go, there are three main constraints. First, and most importantly, it requires the expertise of an FPGA hardware engineer. Second, the total area available is limited, so a system like the one proposed in chapter 4 is required to achieve a proper space/time tradeoff. Third, unlike the SVP, it is completely inflexible and cannot be 'reprogrammed' to do other tasks.

Instead, we can take advantage of the SVP's ability to add specialized vector custom instructions or VCI to provide greatly increased performance for a small increase in area. Unlike custom instructions in a traditional processor, where performance gains are limited by the amount of data available, a VCI has access to a significant amount of high-bandwidth data. It naturally accepts two wide, streaming data sources and produces one wide, streaming data result in a pipelined fashion. Thus, a VCI is a natural candidate for an OpenVX compute kernel, which also streams data and can operate in a pipelined fashion. One major restriction of the VCI, however, is a limit of only two input operands and one output operand.

Thus, to accelerate OpenVX compute graphs for software developers, we can produce an FPGA overlay consisting of a processor resource, such as a host ARM processor and SVP for some acceleration, and a pool of VCI modules. However, choosing the right VCI modules is a difficult problem. Not all OpenVX compute kernels will be used by an application, so it would be wasteful to dedicate area to implement all of them at once. Instead, to fit a limited budget, we can reserve some FPGA area as a partially reconfigurable region and dynamically load the partial bitstream for each VCI as needed. Using an SVP with multiple VCI modules and partial reconfiguration, we can create an FPGA overlay for software programmers using a limited area budget, yet still greatly accelerate OpenVX compute graphs.

Figure 5.1 shows four different scenarios (a, b, c and d) of running a simple application consisting of three nodes (A, B and C) on the proposed system. One way, shown in Figure 5.1.a, is to run all the nodes sequentially as software. Implementing one node as a VCI module, shown in Figure 5.1.b, increases throughput. A second VCI module might improve things further, but the two modules will compete for area if they must be instantiated at the same time. Instead, if we can reconfigure the reserved FPGA region from supporting node A to node C while the

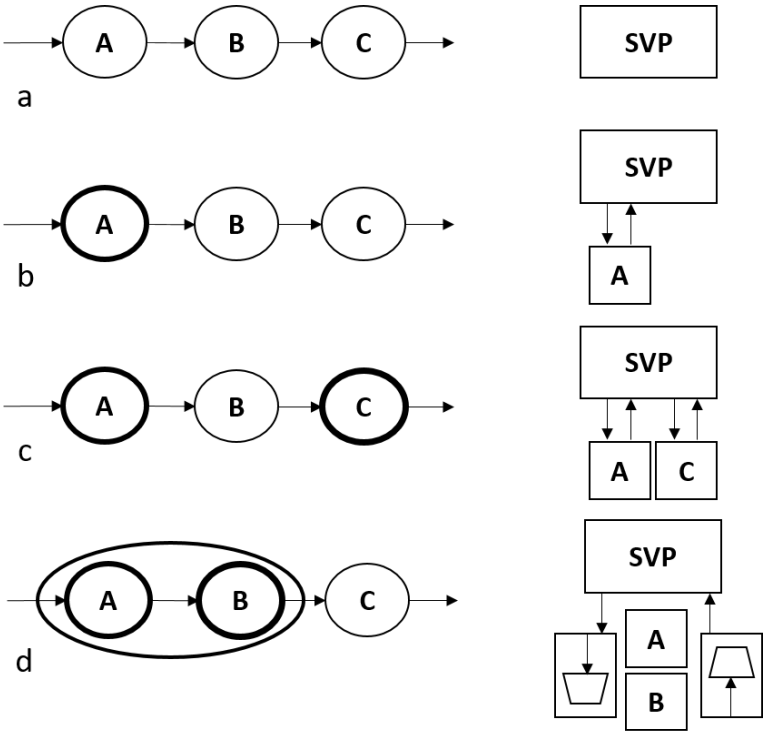


**Figure 5.1:** Running an application on the hybrid system

SVP runs node B, we can realize a greater speedup. Note that, after each node is run, the results are written back into the SVP scratchpad, and must be read out again by the next node. This wastes time. To speed things up, it may be possible to directly stream results from A to B by connecting two VCI modules in a pipelined fashion, as shown in Figure 5.1.d. This is called VCI chaining. Since we do not know beforehand which two modules must be connected, we can build a multiplexer network to forward the data, effectively bypassing the scratchpad in between.

As mentioned before, chip area is limited, so not all compute nodes can be implemented at the same time. However, with timesharing these resources through dynamic partial reconfiguration, we can better dedicate area to nodes that are needed to improve throughput.

There are various ways for running an application on such a system. The key question here is which nodes should be selected to run as software, and which ones

as hardware. Since the VCI needs to achieve fixed levels of throughput, the tool used in chapter 4 is needed to produce an implementation with minimum area. Furthermore, when different VCI modules are chained, implementations with matched throughputs are required to avoid the complications of internal buffering.

This chapter presents a method for the run-time acceleration of an OpenVX application on an SVP system that uses a PRR which can host VCIs. To do this, we pre-generate a library of different VCI implementations that fit the PRR using the tool from chapter 4. Next, using the knowledge of PR speed and how long it will take to instantiate a VCI, we determine which OpenVX nodes should be run as software and which should use a VCI. For a very slow PR speed, we cannot dynamically reconfigure and must select a static set of nodes for acceleration by VCIs that fit the PRR area. Each VCI instance takes a different load time depending upon the bitstream size needed to program that region. If the underlying FPGA can support a faster PR speed, it may become possible to dynamically switch the VCI loaded on the fly. Conceptually, for an infinitely fast PR speed, we can change the VCI for each node in the graph and produce the highest speeds. When VCI chaining is possible, matched throughput implementations must be selected and loaded into the PRRs, and the multiplexer network must be configured accordingly.

Thus, the speed of the PR, the bitstream size of the VCI, the size of the PRR supported, the size of the image, the tile size used, and the OpenVX graph (eg, which nodes, how they are arranged, and whether VCI chaining is possible) will all affect performance.

Since we cannot easily modify the speed of the PR (it is fixed by the FPGA vendor), we build a framework that takes these variables into account and estimates overall performance.

There have been several recent studies on implementing image processing and OpenVX applications on reconfigurable platforms [37, 38, 70, 78]. The contributions listed below distinguish our work from previous approaches:

1. Adding automated space/time tradeoffs to the process of generating VCI for SVP in order to use parallel resources more efficiently.

2. Adding PR to VCI implementation to time-share the parallel resources.

3. Adding scratchpad bypass capability to improve performance with VCI chaining.

4. Using pre-synthesized node fusion as well as a heuristic approach to save area compare to classic ILP approaches.

## 5.2 System Overview

The FPGA-based overlay consists of an ARM Cortex-A9 host processor, the VectorBlox MXP SVP [75], and an empty PRR reserved for vector custom instructions. To use a VCI, the associated bitstream must first be loaded into the reconfigurable region. Static PR loads bitstreams for VCIs until the PRR is full. Dynamic PR allows the bitstream to change quickly on the fly, enabling the use of more VCIs than the PRR can hold at once. Figure 5.2 provides a view of the overall system.

The MXP has its own scratchpad and supports up to 16 different VCI opcodes. The scratchpad provides two streaming source operands on PortA and PortB, and accepts results on PortC. Each VCI can be implemented within the reconfigurable region and connect to these ports. The Multiplexer (MUX) network is implemented as part of each VCI and connects it with the SVP; the VCIs can forward their results directly to each other through the MUX network in a way that bypasses writing the result to the scratchpad.

The VCIs for OpenVX are generated using the synthesis system described earlier in chapter 4, thus ensuring maximum throughput is achieved within the area constraints provided. The OpenVX compute graph is analyzed and each compute kernel is run as either regular software or a custom instruction in an accelerated hardware pipeline.

According to Xilinx, the tool support for partial reconfiguration is based upon large, non-overlapping regions with predefined boundaries. However, this is a current limitation of the tools, and not the underlying device architecture. For example, tools such as GoAhead [9] enables fine-grained boundaries, fine-grained vertical relocation of the bitstream, and the ability to lock down interconnect wires feeding these reconfigurable regions from non-reconfigurable regions. These tools

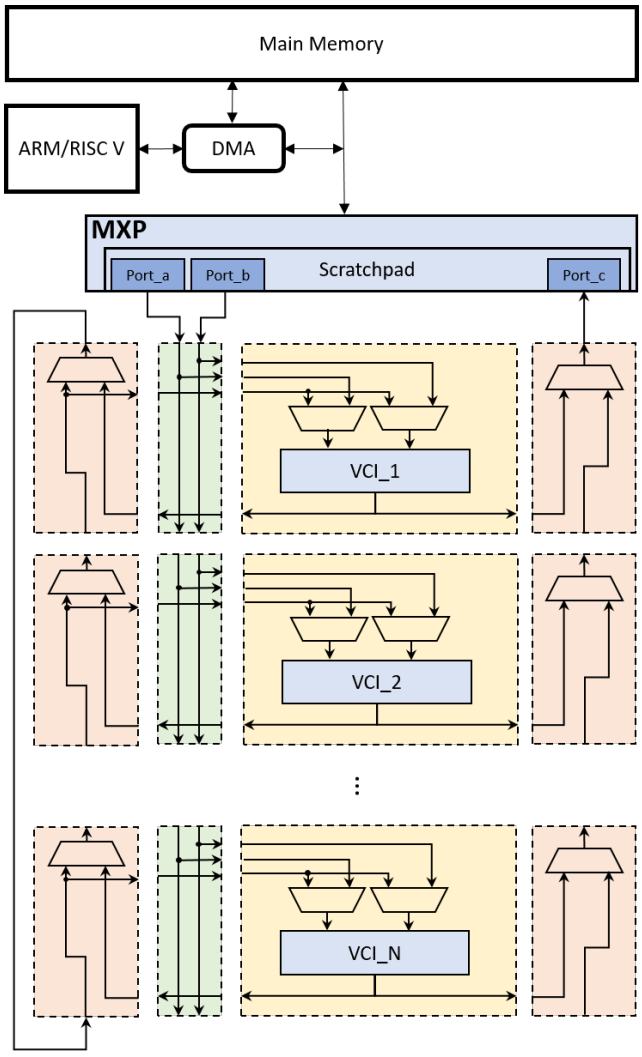


**Figure 5.2:** System overview

can be also extended to modern devices under Vivado. [1] In our system, we assume tools like GoAhead will be available for modern devices, but until then this work is limited to being a conceptual study.

[1] Dirk Koch, private communication.

## 5.3 Mapping OpenVX Applications to FPGA Overlay

The process of mapping an OpenVX application involves two steps. The first step is similar to chapter 4 and done in advance by FPGA experts: the OpenVX kernels, written in C++ for Vivado HLS, are mapped for defined levels of throughput. The throughput levels are defined to match the width of the SVP VCI, which can be anywhere from 1 to N lanes, where N is a power of 2 that matches the largest SVP to be used. This produces a minimal-area VCI for each width. Each VCI implementation is noted with its throughput, area requirement, and bitstream size. Each OpenVX kernel also has a pure software implementation on the SVP or host processor, where the throughput, in terms of pixels per second it can process, is recorded at each image tile size.

The second step is executed at application time by the OpenVX run-time system. Given an OpenVX compute graph and a target image size, it uses an execution time model to determine the best tile size, which nodes should be implemented as a VCI, and the VCI implemention to use. It also determines whether to use bypassing and node fusion.

All of these steps are described below.

### 5.3.1 Finding Different Implementations

Consider an application described as an STG $G$ with $N$ nodes $f_1, f_2, ..., f_N$.

$$G = (V, E) \tag{5.1}$$

$$V = \{f_1, f_2, ..., f_N\} \tag{5.2}$$

For each node $f_m$ we can find $N_{SVP}$ different SVP implementations $S_m^1, S_m^2, ..., S_m^{N_{SVP}}$ as well as $N_{VCI}$ different VCI hardware implementations $P_m^1, P_m^2, ..., P_m^{N_{VCI}}$. Each SVP implementation $S_m^s$ can perform functionality of $f_m$ on an image tile, in $t(S_m^s)$ time. Each VCI implementation $P_m^s$ can perform functionality of $f_m$ with area cost $A(P_m^s)$, number of pixels it can consume/produce $NP(P_m^s)$ each firing (i.e., tile width), and initiation interval $II(P_m^s)$.

Considering available resources, such as the size of the PRR, scratchpad capacity, and PR reconfiguration speed, the OpenVX run-time system decides which

node should be implemented as SVP software and which one should be implemented as VCI hardware. After enumerating different implementations for each OpenVX node, to minimize the search space, the OpenVX run-time system prunes any dominated implementation points. Moreover, it uses "execution time analysis" to consider other factors, namely DMA time and PR time, to find out the overall execution time for each implementation.

### 5.3.2 Execution Time Analysis

In order to execute a general OpenVX compute graph, the run-time system needs to fetch each image tile from main memory to the scratchpad and execute the whole compute graph one node at a time (either as SVP or VCI implementations). In every stage of traversing the graph, the intermediate data is saved in the scratchpad. For large graphs, a significant amount of intermediate data may need to be buffered. This means the tile size must be calculated based on the available scratchpad size and amount of buffering required. After finishing executing all the nodes in the graph on a tile, results are written back to main memory before fetching the next tile. The execution time for these components is discussed below.

#### SVP Software Implementation

The execution time of executing a compute graph $G$ with $N$ nodes $f_1, f_2, ..., f_N$ and subset of selected SVP implementations $S_1, S_2, ..., S_N$ on image tile $T_j$ is $t_{T_j}$. This is calculated as:

$$t_{T_j} = t_{DMA_{M2S}} + [\sum_{i=1}^{N} t(Si)] + t_{DMA_{S2M}} \tag{5.3}$$

The overall execution time for $N_T$ tiles in the image is calculated as:

$$t_A = \sum_{j=1}^{N_T} t_{T_j} \tag{5.4}$$

### Accelerated VCI Implementation

Consider node $f_m$ in the compute graph $G$. Instead of using SVP implementation $S_m$ with execution time $t(S_m)$, it is possible to use a VCI hardware implementation $P_m$ with execution time $t(P_m)$ and PR reconfiguration time $t_{PR}$ to improve the execution time. Assuming we need to reconfigure this node $N_{PR}$ times during the processing of the entire image, the execution time can be improved if:

$$N_T t(S_m) > N_{PR} \cdot t_{PR} + N_T \cdot t(P_m) \tag{5.5}$$

In chapter 4 we showed that for most CV kernels implemented as pipelined hardware accelerators, we can define kernel throughput $\Theta(P_m)$ as number of pixels consumed/produced in each clock cycle. The same formulation can be used here to calculate VCI execution time $t(P_m)$:

$$t(P_m) = SetupTime(P_m) + \frac{TileSize}{\Theta(P_m) \cdot F_{max}} \tag{5.6}$$

where the $F_{max}$ is the speed of the SVP (here, 100MHz). In addition:

$$t_{PR} = \frac{PR_{size}}{PR_{rate}}. \tag{5.7}$$

### Node Chaining

Each VCI normally implements one OpenVX kernel, and only one VCI is executing at a time. However, if the graph topology allows, it's possible to find a cluster of nodes where a series of VCIs can chain together, sending the output of one directly to the input of the next, without writing intermediate results to the scratchpad. This scratchpad bypassing allows us to take advantage of pipeline parallelism by overlapping VCI execution.

Although chaining improves performance, it requires all VCI implementations to be active at the same time. That is, there must be sufficient area in the PRR to hold the entire VCI chain. In addition, the overall VCI chain still needs to follow VCI topology restrictions: overall, there can be a maximum 2 input operands (PortA and PortB) and one destination operand (PortC).

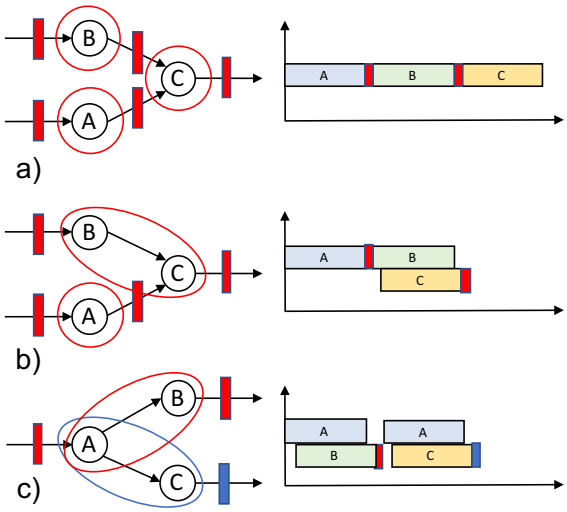


**Figure 5.3:** Node clustering and bypassing the scratchpad

For example, Figure 5.3.a shows a graph with three nodes and its execution timeline. Standalone VCI implementations are used for each node in the graph. This means each VCI implementation needs to write its results to the scratchpad for its successors. It also means each node must wait for its predecessors to finish their jobs before it can begin. Since the execution of nodes *A*, *B* and *C* do not overlap, the system only needs to keep one VCI configured at a time within the PRRs.

In contrast, nodes *B* and *C* can be chained as shown in Figure 5.3.b. The chain bypasses the scratchpad for writing, so the result of node *B* bypasses the scratchpad and is sent directly by the MUX network to the VCI implementing node *C*. For this to work, the VCI for node *A* must be executed first, and the MUX network must be configured to stream data through the VCIs for both *B* and *C* which must be active simultaneously.

In a different example, shown in Figure 5.3.c, nodes *B* and *C* are both using the result of node *A*. This concept of fan-out was not present in the previous two examples. To avoid writing the result of *A* to the scratchpad, two VCI chains must be formed: *A* and *B*, as well as *A* and *C*. This example shows that clustering needs to consider all uses of the intermediate data between nodes.

Now that we have explained VCI chaining, we will describe the execution time

analysis for standalone VCIs as well as VCI chains.

### Pruning Slow Standalone VCIs

To enhance performance and reduce the search space, standalone VCI implementations that are slower than SVP implementations are pruned. Hence, we will only keep VCIs that satisfy the following equation:

$$t(S_m) > \frac{PR_{size}.N_{PR}}{PR_{rate}.N_T} + \frac{TileSize}{\Theta(P_m).F_{max}} + SetupTime(P_m). \tag{5.8}$$

### Bypassing the Scratchpad

Similarly, we will prune VCI chains that are slower than the SVP implementations. Assuming we can implement a sequence of $N_C$ nodes as a VCI chain, we will only keep VCI chains which that satisfy the following equation:

$$\begin{aligned}\sum_{j=1}^{N_C} t(S_M^j) > \sum_{j=1}^{N_C} \frac{PR_{size}^j.N_{PR}^j}{PR_{rate}^j.N_T} + \max[\frac{1}{\Theta(P_m^j)}].\frac{TileSize}{F_{max}} \\ + \sum_{j=1}^{N_C} SetupTime(P_m^j).\end{aligned} \tag{5.9}$$

We prune the problem space by eliminating all implementations that cannot satisfy Equation 5.5, Equation 5.8 and Equation 5.9.

### Pre-synthesized Node Fusion

Instead of VCI chaining, it is possible to fuse nodes together. This accomplishes a similar result, but the VCI must be pre-synthesized, so the pair of nodes to be fused must be known in advance. Hence, this can be done as long as there is sufficient time to precompute a library of all pairs of OpenVX nodes.

The difference between VCI chaining and node fusion is shown in Figure 5.4. With chaining, the MUX network is used to steer the output of $A$ to the input of $B$. With node fusion, the connection is made directly and the entire fused function is synthesized into a single VCI. This can yield higher performance within an area budget; for example, with node chaining, each VCI might be limited to 2 pixels per

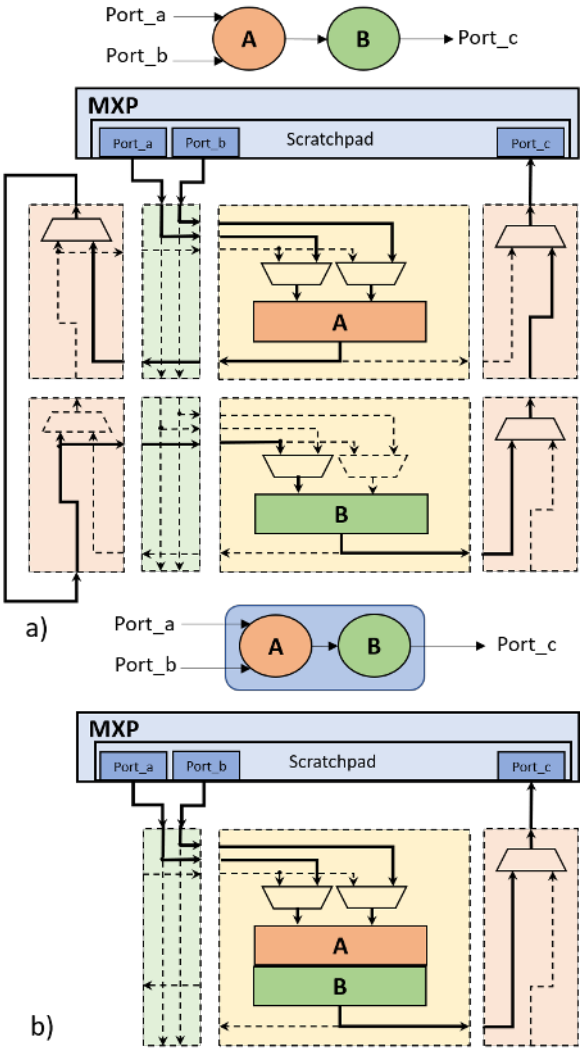


**Figure 5.4:** VCI chaining versus node fusion

**Table 5.2:** Some of common patterns used for pre-synthesized node fusion

| Pattern |
|---|
| ConvertColor, Gaussian |
| Gaussian, Sobel_X |
| Gaussian, Sobel_Y |
| ConvertColor, Gaussian, Sobel_X |
| ConvertColor, Gaussian, Sobel_Y |
| Sobel_X, Sobel_Y, Magnitude |
| Sobel_X, Sobel_Y, Phase |
| Gaussian, Sobel_X, Sobel_Y, Magnitude |
| Gaussian, Sobel_X, Sobel_Y, Phase |
| ConvertColor, Gaussian, Sobel_X, Sobel_Y, Magnitude |
| ConvertColor, Gaussian, Sobel_X, Sobel_Y, Phase |
| Magnitude, Phase, Non-Maxima |
| Sobel_X, Sobel_Y, Magnitude, Phase, Non-Maxima |

firing, whereas node fusion might be able to support 4 pixels per firing within a similar budget.

It may not be feasible to pre-synthesize all pairs of OpenVX nodes. Fusing more than two nodes is even more computationally difficult. However, a subset might be feasible if the graph topology is known in advance. In particular, there are several common sub-graph patterns in OpenVX applications which can be anticipated. For example, the *Color Convert* kernel followed by *Guassian Filter* kernel is a common 2-node sequence. Table 5.2 lists a few common patterns with up to 5 nodes that might be useful. In this case, these patterns must still conform to the overall two-input, one-output operand structure, but with node fusion it is possible to encapsulate more complex internal structures.

### 5.3.3 Solving the Space/Time Tradeoff

After pruning the problem space, the next step is exploring space/time tradeoffs to find suitable implementations for each OpenVX node and solving the scheduling problem by finding which ones need to be implemented as SVP and which ones need to be implemented as VCI or VCI chains.

Previous studies have shown the scheduling problem can be defined as an ILP

problem and be solved by ILP solvers [49, 82]. In chapter 4, we also demonstrated an ILP approach for automatically exploring space/time tradeoffs. Combining these two ILP formulation approaches, we can easily produce an ILP model and use a solver such as GLPK [60].

Although ILP solvers can solve these problems, they lack flexibility. In particular, the ILP problem cannot model all possible constraints, and it can quickly become computationally infeasible to solve. In other words, as we have shown elsewhere, combining or splitting nodes are not feasible while using ILP. In addition, as shown in chapter 4, ILP solvers can be slower than heurstics.

To overcome those mentioned shortcomings, we developed a heuristic approach which allows us to use node fusion. The heuristic approach is similar to the approach we used in chapter 4 in the sense that it uses the "node combining" ability (explained subsection 3.3.2) to allow for node fusion. Going through different possible nodes to fuse, the heuristic uses exhaustive search similar to ILP to find which nodes need to be implemented as SVP and which ones need to be implemented as VCI.

## 5.4 Experimental Results

In this section, we will investigate the speedup provided by the SVP and various VCI configurations. The performance of three different hardware configurations are measured: the baseline, consisting of OpenVX kernels written in C and running on the ARM Cortex-A9 processor at 667MHz; the SVP running at 100MHz without any VCI, consisting of OpenVX kernels written in C using the VectorBlox MXP API and running on a combination of the Cortex-A9 and MXP; and the SVP with different VCI options at 100MHz.

The OpenVX kernels implemented for this study are shown in Table 5.3. For the HLS versions, a library of different PR bitstream implementations for each VCI instance were generated to facilitate the trade-off finding process. These VCIs were generated using the HLS tool described in chapter 4.

The baseline SVP and ARM code was implemented by Nick Ivanov of VectorBlox. Mr. Ivanov generated two versions of the SVP hardware, and using that hardware he gathered runtime data for the OpenVX kernels. The two SVP versions

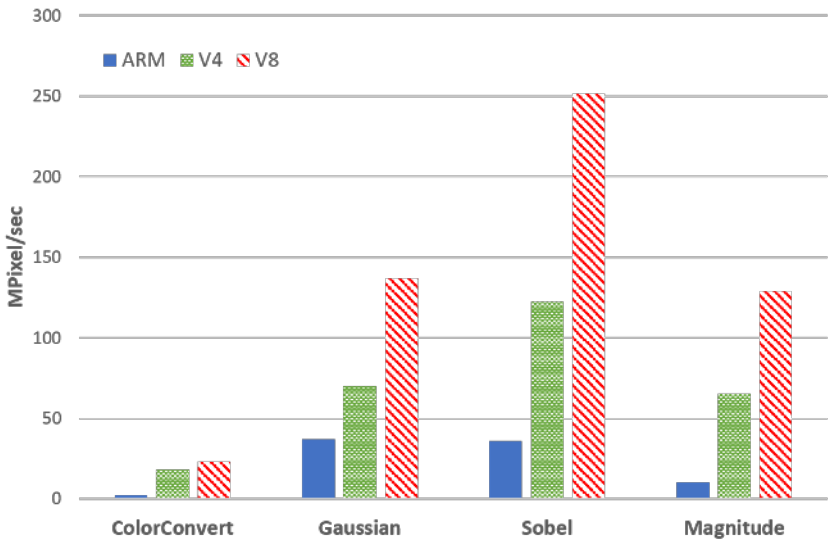


**Figure 5.5:** ARM Cortex-A9 (667MHz) vs SVP-V4 and SVP-V8 (100MHz)

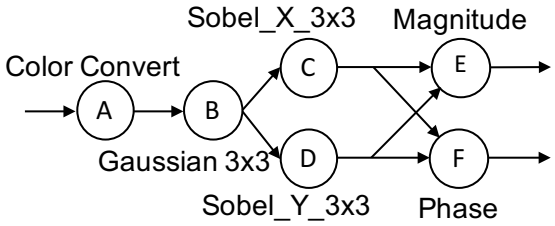


**Figure 5.6:** Graph representation of *Sobel* application with 6 nodes

generated were a 4-lane configuration, where four 32-bit ALUs are assembled and called SVP-V4, and an 8-lane configuration, called SVP-V8.

The throughput results for four specific kernels (ColorConvert, Gaussian, Sobel and Magnitude) running on 3 different platforms (ARM Cortex-A9, SVP-V4 and SVP-V8) are shown in Figure 5.5. On average across those kernels, the SVP achieves a 4.6 times speedup on SVP-V4 over the Cortex-A9, and an 8 times speedup using SVP-V8.

Unfortunately, only a few general algorithms have been released as public benchmarks for OpenVX. This means there is no standard OpenVX benchmark suite available. To show the capabilities of our approach, we implemented following simple benchmarks as OpenVX compute graphs:

**Table 5.3:** List of CV kernels

| CV kernel | ARM | SVP | HLS |
|---|---|---|---|
| Color Conversion | ✓ | ✓ | ✓ |
| Channel Extract | ✓ | ✓ | ✓ |
| Gaussian filter | ✓ | ✓ | ✓ |
| Sobel filter | ✓ | ✓ | ✓ |
| Phase | ✓ | ✓ | ✓ |
| Magnitude | ✓ | ✓ | ✓ |
| Non-maxima Suppression | ✓ | ✓ | ✓ |
| Thresholding | ✓ | ✓ | ✓ |
| Median Filter | ✓ | ✓ | ✓ |
| Box Filter | ✓ | ✓ | ✓ |
| Channel Combine | ✓ | ✓ | ✓ |
| Absolute Difference | ✗ | ✓ | ✓ |
| Accumulate | ✗ | ✓ | ✓ |
| Accumulate Squared | ✗ | ✓ | ✓ |
| Accumulate Weighted | ✗ | ✓ | ✓ |
| Arithmetic Addition | ✗ | ✓ | ✓ |
| Arithmetic Subtraction | ✗ | ✓ | ✓ |
| Bitwise AND | ✗ | ✓ | ✓ |
| Bitwise OR | ✗ | ✓ | ✓ |
| Bitwise XOR | ✗ | ✓ | ✓ |
| Bitwise NOT | ✗ | ✓ | ✓ |
| Convert Bit Depth | ✗ | ✓ | ✓ |
| Dilate Image | ✗ | ✓ | ✓ |
| Erode Image | ✗ | ✓ | ✓ |
| LBP | ✗ | ✓ | ✓ |
| Pixelwise Multiplication | ✗ | ✓ | ✓ |
| Magnitude-cordic | ✗ | ✗ | ✓ |
| Sqrt-cordic | ✗ | ✗ | ✓ |
| Arctan-cordic | ✗ | ✗ | ✓ |

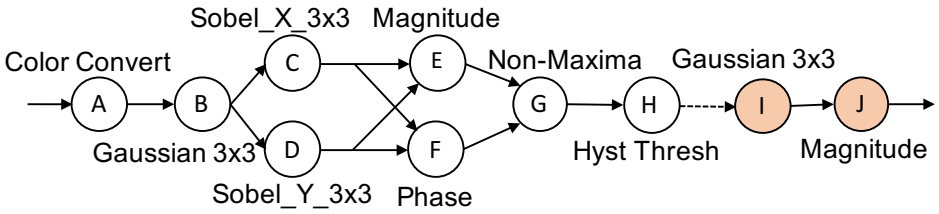


**Figure 5.7:** Graph representation of *Canny – Blur* application with 10 nodes

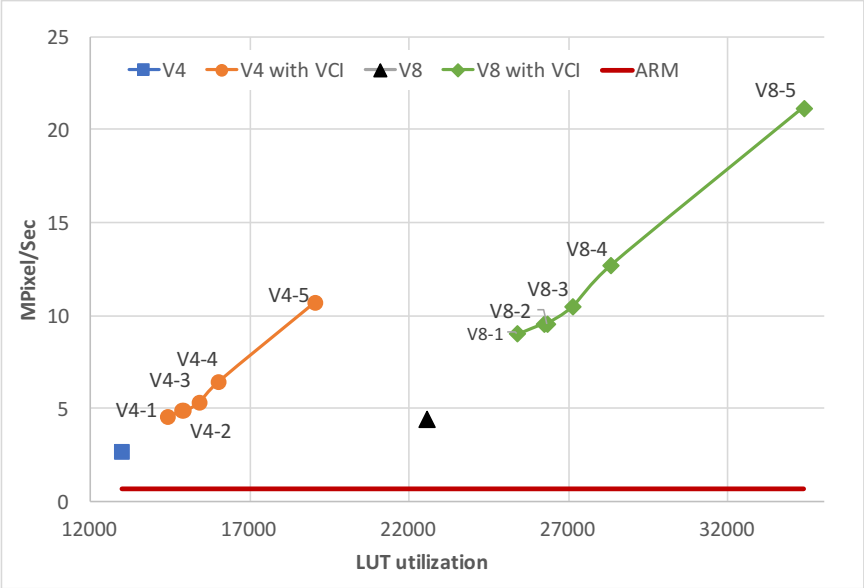


**Figure 5.8:** Throughput vs area for V4 and V8 with/without VCI (*Canny – Blur* Figure 5.7)

- *Sobel* application with 6 nodes, shown in Figure 5.6.
- *Canny-blur* application with 10 nodes, shown in Figure 5.7.

All the kernels in both applications can be run using image tiles except the hysteresis thresholding kernel (node H) in *Canny – blur*. The hysteresis thresholding kernel needs the global image perspective, so it must DMA all tile results prior to that node before running the subsequent nodes. Thus, to run *Canny – blur*, we need to run the first part on whole image, save to memory, read the results back and then run the second part on the whole image generated by first part.

**Table 5.4:** List of CV kernels implemented as VCIs in Figure 5.8

| Implementation | Kernels implemented as VCIs |
|---|---|
| V4-1 | (AB) |
| V4-2 | (AB), D |
| V4-3 | (ABC), (ABD) |
| V4-4 | (ABC), (ABD), H, I |
| V4-5 | (ABC), (ABD), H, (IJ) |
| V8-1 | (AB) |
| V8-2 | (AB), C |
| V8-3 | (ABC), (ABD) |
| V8-4 | (ABC), (ABD), H, I |
| V8-5 | (ABC), (ABD), H, (IJ) |

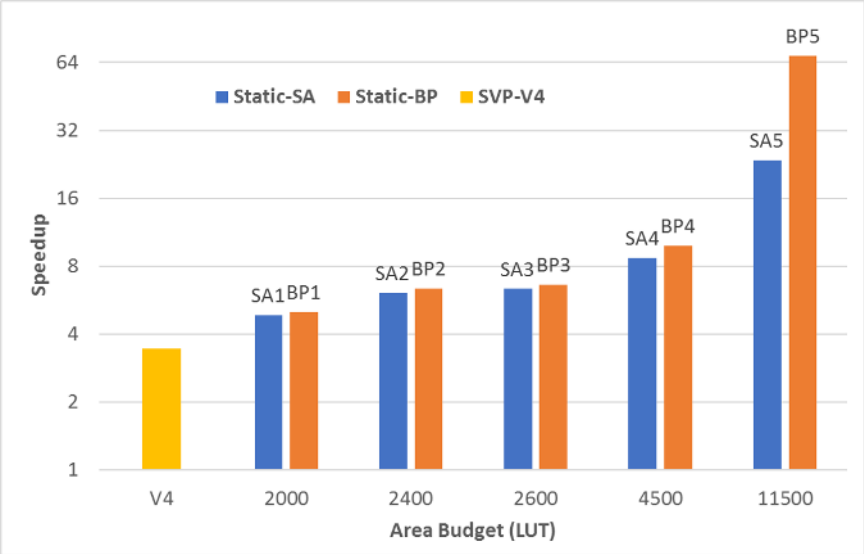


**Figure 5.9:** *Sobel* speedup by adding static VCIs (standalone and bypassing) to SVP-V4 compared to ARM

### Benefits of Static VCIs

Although SVPs are able to achieve better throughput than ARM, the use of VCIs offers significant improvement. Figure 5.8 shows the results of running the *Canny – blur* benchmark on SVP-V4 and SVP-V8, both without and with VCIs. The Y-axis shows the throughput achieved in Mega-Pixels per second, while the X-axis shows the LUT utilization, i.e. the amount of FPGA area used by each implementation.

We set different PRR sizes for implementing VCIs connecting to V4 or V8. Considering the area budget, our tool explored the solution space and decided which nodes need to be implemented as VCIs and which ones as SVPs in order to maximize the throughput. When PRR size increases, throughput increases. Five VCI solutions were generated for each of V4 and V8; these are shown in Table 5.4. In this table, the letter indicates which node(s) from the graph in Figure 5.7 are accelerated with a VCI; a grouping with parentheses indicates a VCI chain that bypasses the scratchpad.

Note that in this figure, all of the VCIs are implemented in a fully static manner. That is, $N_{PR} = 0$ in Equation 5.5, so the VCI is only configured once at application load time. Below, we will demonstrate the benefits and limitations of using dynamic PR, thereby changing which VCI is implemented as an image tile is passed from node to node in the graph.

### Impact of Bypassing

The previous subsection included bypassing to achieve better results. In this subsection, we remove bypassing to show the gains it provides. Results in Figure 5.9 show speedup before and after bypassing on an SVP-V4 when a fixed area budget is given to VCIs. For a small area budget, bypassing does not yield significant improvements. However, as the area budget grows, the benefit of bypassing increases; in the version with the largest area budget, the speedup with bypassing is over 2 times faster. A breakdown of the VCIs used in this figure is shown in Table 5.5.

### Impact of Dynamic Partial Reconfiguration

Instead of statically assigned VCIs within the PRR, it is also possible to dynamically reconfigure each VCI while evaluating a graph. To simplify discussion, suppose $N_{PR} = N_T$ in Equation 5.5. This might be the case when the first use of a VCI incurs latency, but future uses within a graph can hide the latency (e.g., through prefetching). In this case, the time to reconfigure and run on the VCI must also be faster than the software-only SVP implementation. That is,

$$t(S_m) > t_{PR} + t(P_m). \tag{5.10}$$

**Table 5.5:** List of CV kernels implemented as VCIs on SVP-V4 in Figure 5.9

| Implementation | Kernels implemented as VCIs |
|---|---|
| SA1 | A, B |
| BP1 | AB |
| SA2 | A, B, C |
| BP2 | (AB), C |
| SA3 | A, B, D |
| BP3 | (AB), D |
| SA4 | A, B, C, D |
| BP4 | (ABC), (ABD) |
| SA5 | A, B, C, D, E, F |
| BP5 | (ABCDE) , (ABCDF) |

When we allocate a new VCI to the PRR, we need to select a precise location. First, we do a simple first-fit search strategy in the free space. If that fails, we swap out the VCI that has been idle for the longest time. We find that most VCIs with the same bandwidth require similar amount of space within the PRR.

This allocation heuristic is far from perfect, and it may lead to internal fragmentation within the PRR. Since the entire graph is known, a better scheduler can look at whether kernels are used multiple times in the graph. Also, since the multiple image tiles will be passed through the entire graph, the reconfiguration schedule is cyclic. Using these properties, a better scheduler can optimize for the fewest reconfigurations while also avoiding fragmentation. Rather than attempting to create such an optimal heuristic, we went with a simple one (which has more reconfigurations than necessary, thereby conservatively underestimating the performance impact) and ignored the fragmentation problem (since it is likely solvable). These practical issues should be addressed in future work.

### Impact of PR Rate and Image Size

One concern of using dynamic PR is the PR rate, or the speed at which a new configuration can be loaded. Considering Equation 5.7 and Equation 5.10, this rate can significantly influence the benefits of using dynamic PR. Although in this study we used Xilinx FPGAs, the overall PR rate for Intel FPGAs is similar [93]. As a starting point, Xilinx documents a maximum 400MByte/sec PR rate using their on-chip

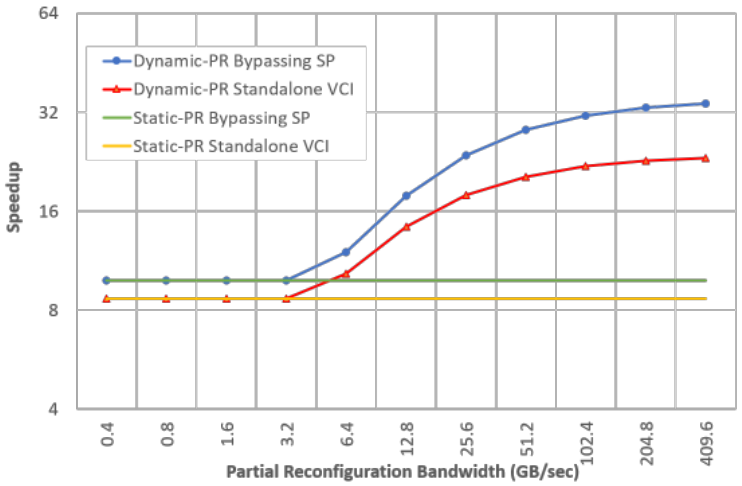


**Figure 5.10:** V4 Dynamic PR and Static PR speedup vs ARM for Sobel Application (4500 LUT budget, image size $1920 \times 1080$)

ICAP interface; this is specified as a 32b value every 100MHz clock cycle. However, Hansen et al. demonstrated this can be overclocked by more than a factor of 5 to achieve 2.2GByte/sec on real devices [35]. Other studies have suggested different architectures to achieve even faster PR rates. For example, Trimberger et al. proposed a time-multiplexed FPGA architecture which provides 33GByte/sec reconfiguration rate [84]. Unfortunately, FPGA vendors have not made it a priority to provide fast PR rates; we believe this work provides some of that missing incentive.

Figure 5.10 shows the impact that PR rate has on speedup. In this figure, the *Sobel* application is run on an SVP-V4 with both with dynamic PR and static PR on an image of size $1920 \times 1080$. As the PR rate increases along the X-axis, the overall speedup (relative to the ARM) also increases. At low PR rate values of 3.2GB/s or lower, dynamic PR is not considered because it is slower than static. At 6.4GB/s and above, dynamic PR becomes faster than static. Bypassing benefits more from fast PR because it covers a greater proportion of the total runtime. These same trends are demonstrated on the *Canny-blur* application on both an SVP-V4 version with area budget of 4500 LUTs in Figure 5.11, and an SVP-V8 version with area budget of 14000 LUTs in Figure 5.12.

Similar to how increasing the PR rate improves bypassing more, increasing the

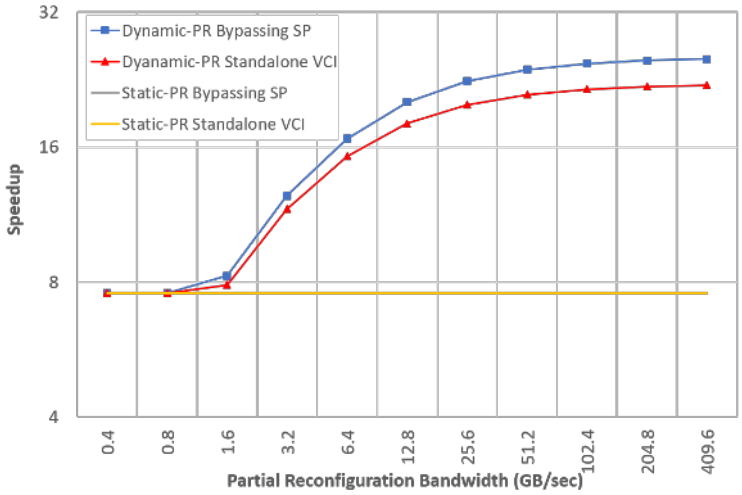


**Figure 5.11:** V4 Dynamic PR and Static PR speedup vs ARM for Canny-blur Application (4500 LUT budget, image size $1920 \times 1080$)

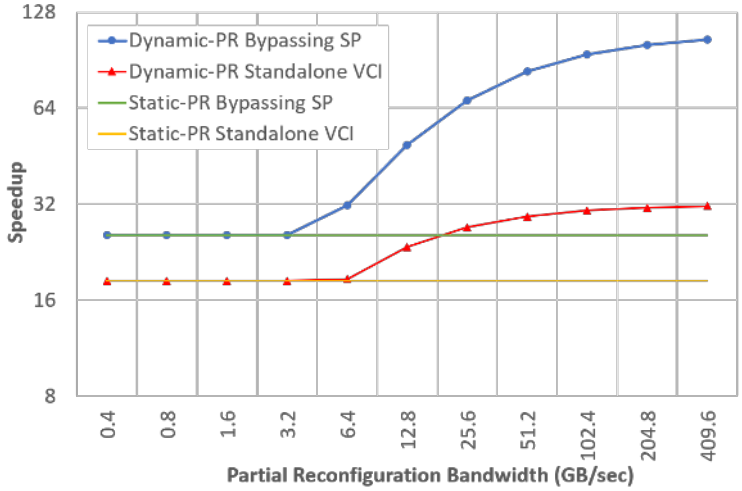


**Figure 5.12:** V8 Dynamic PR and Static PR speedup vs ARM for Canny-blur Application (14000 LUT budget, image size $1920 \times 1080$)

image size can also improve performance, particularly at low PR rates. This is shown in Figure 5.13 where speedup improves over larger images. This is because the larger image size allows the use of a taller image tile, allowing the current node computation to better hide the latency of configuration for future nodes. At low PR rates, the proportion of time spent doing PR relative to the total computational size

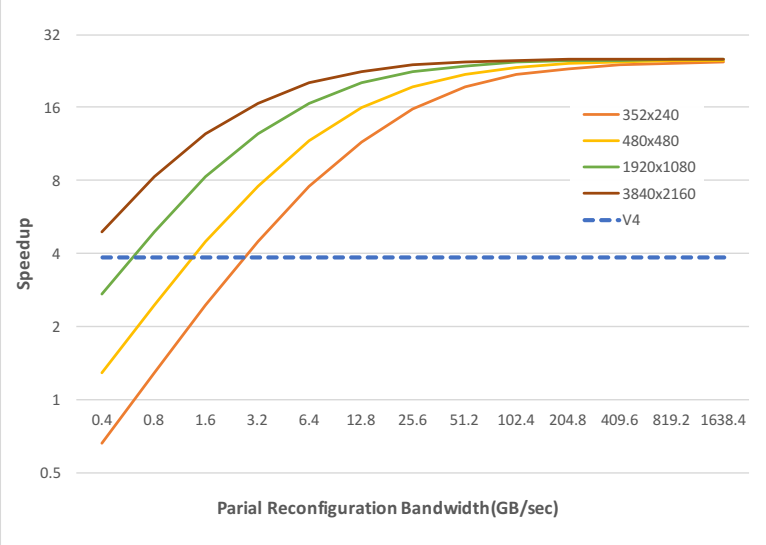


**Figure 5.13:** V4 Dynamic PR speedup vs ARM for Canny-Blur for different image sizes (4500 LUT budget)

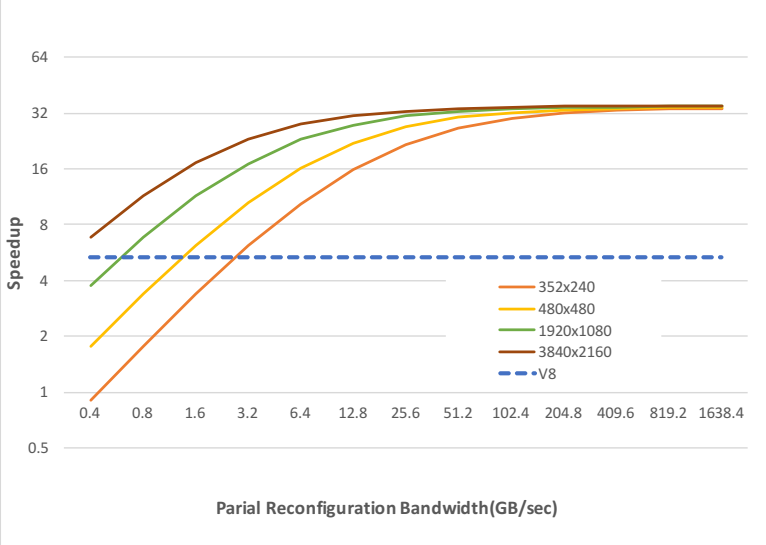


**Figure 5.14:** V8 Dynamic PR speedup vs ARM for Canny-Blur for different image sizes (4500 LUT budget)

goes down with larger images. At high PR rates, the PR overhead has already been minimized so changing the image size has little impact on performance.

**Impact of Node Fusion Heuristic**

So far, all of the above results have been generated using the ILP approach. We also tested the heuristic approach, which enables node fusion as a new optimization, using the same throughput targets or area budgets. Just like the individual kernels, the candidates for fusion, taken from Table 5.2, were presynthesized. We have omitted the detailed results showing that the heuristic is able to match all of the throughput results achieved by the ILP while using 9% less area. This area savings could be useful to reduce the number of reconfigurations, thereby reducing the reconfiguration time, or to increase the number of VCIs or their width, thereby improving performance.

## 5.5 Summary

This chapter presented a method for the run-time acceleration of an OpenVX application on an FPGA-based overlay system that uses a PRR which can host VCIs. To do this, we pre-generated a library of different VCI implementations that fit the PRR profile using the automated FPGA space/time scaling tool we introduced in chapter 4. The OpenVX application's compute graph is analyzed and each compute kernel is run as either regular software or as a custom instruction in an accelerated hardware pipeline configured on the PRR. A pipeline of compute kernels can sometimes be realized by chaining multiple VCI together using a custom multiplexer network. With partial reconfiguration, this method can obtain speedups far beyond what a plain SVP can accomplish. For example, on the *Canny-blur* application, an 8-lane SVP is 18 times faster than the plain ARM Cortex-A9. However, using ultra-fast partial reconfiguration which is technically feasible but not yet supported on modern FPGAs, a speed of 106 times faster is possible. This allows OpenVX programmers, who have no FPGA design knowledge, to achieve hardware-like speeds within their vision application.

# Chapter 6

# Conclusions

This work broadens the overall usability of parallel resources by providing an environment which allows users to automatically explore space/time tradeoffs and find suitable implementations regarding a throughput target or an area budget on a wide range of different pipelined architectures. The contributions made in this dissertation are summarized below.

First, we added automated space/time scaling for streaming applications to compilation tools for MPPAs (a coarse-grained architecture). This was done by proposing a Java-based HLS tool chain which finds all degrees of parallelism in a general stream application and then uses throughput analysis and throughput propagation to find possible bottlenecks. The proposed approach combines module selection and replication methods with node combining and splitting in order to automatically find a better area/throughput tradeoff. It also presents a heuristic approach which is more flexible and can find design points that are computationally infeasible or difficult to model using a classic ILP formulation. We examined different benchmarks and the tool satisfied a variety of different throughput targets and area budgets. Our heuristic approach could achieve the throughput targets using 30% less area compared to the ILP approach.

Second, automated exploration of space/time tradeoffs was added to a commercial HLS tool for implementing CV applications targeting FPGAs. OpenVX was used to define CV applications as STGs. This approach was verified with different OpenVX benchmarks targeting several different FPGA sizes. Our tool is able to au-

tomatically achieve over 95% of the target area budget on average while improving throughput. Our tool also can automatically satisfy a variety of throughput targets while minimizing the area cost. The proposed system saves up to 30% of the area cost compared to manually written implementations. Using the Inter-node Optimizer step, our heuristic tradeoff finder is able to hit the same throughput targets while saving 19% area on average compared to existing ILP approaches.

Third, an automated space/time tradeoff approach was used for implementing CV applications targeting an FPGA-based overlay architecture that combines a processor, a vector execution unit (SVP), and reconfigurable custom vector instructions. The SVP allows us to target more general applications and coupling it with the FPGA fabric provides us an environment to accelerate computationally intensive applications. Automated space/time scaling was used to automatically generate different VCIs for the SVP in order to use parallel resources more efficiently. Also, Partial Reconfiguration (PR) was used for implementing VCIs in order to time-share the parallel resources. Scratchpad bypassing capability was added to VCIs in order to improve the performance. Moreover, node clustering and node combining approaches are used to avoid local memory access as much as possible. Last, pre-synthesized node fusion allows the heuristic approach to use additional optimization opportunities which are difficult to model or computationally infeasible in classic ILP approach. The heuristic approach matches all the throughput results achieved by the ILP approach while using 9% less area on chip.

Overall, the performance results achieve speedups far beyond what a plain SVP can accomplish. For example, an 8-lane SVP achieves a speedup of 5.3, whereas a VCI version is another 3.5 times faster, with a net speedup of 18.5 versus ARM Cortex-A9 for running the $Canny - blur$ application. This was achieved by using automated space/time scaling, node clustering and dynamic PR (considering today's device restrictions). These speedup results can be significantly improved up to 106 times faster than ARM if the PR rate increases in the future.

The continuing trend toward designing larger pipelined architectures, as well as introducing more complex streaming applications, requires smarter and more capable approaches in order to find all degrees of parallelism in the application and use available parallel on-chip resources efficiently. In this work, we proposed an environment in which users can explore automated space/time tradeoffs for STGs

however we only addressed the STG with DAG topologies. Addressing automated space/time tradeoffs for cyclic graphs needs to be investigated. Moreover, we believe our approach can be used not only for CV applications but also for a wide range of streaming applications which we did not address because they were out of scope of this study.

# Bibliography


[1] Adapteva-Inc. Epiphany-iv 64-core 28nm microprocessor. 2014. URL http://www.adapteva.com/products/silicon-devices/e64g401/. → page 19

[2] P. J. Ashenden. *The designer's guide to VHDL*, volume 3. Morgan Kaufmann, 2010. → page 18

[3] Avnet-inc. Zedboard product briefs. 2017. URL http://www.zedboard.org/sites/default/files/product_briefs/5066-PB-AES-Z7EV-7Z020-G-V1b.pdf. → page 71

[4] M. Awad. Fpga supercomputing platforms: a survey. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 564–568. IEEE, 2009. → page 23

[5] E. Ayguadé and J. Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proceedings of the 7th international conference on Supercomputing*, pages 407–415. ACM, 1993. → page 11

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4): 345–420, 1994. → pages 11, 31

[7] U. Banerjee. *Unimodular transformations of double loops*. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990. → page 11

[8] U. Banerjee. *Loop transformations for restructuring compilers: the foundations*. Springer Science & Business Media, 2007. → pages 11, 31

[9] C. Beckhoff, D. Koch, and J. 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. → page 78

[10] P. Bellows and B. Hutchings. Jhdl-an hdl for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175–184. IEEE, 1998. → page 15

[11] V. Betz and J. Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997. → page 65

[12] M. Butts, A. M. Jones, and P. Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 55–64. IEEE, 2007. → pages xii, 2, 14, 20, 21, 22, 28

[13] M. Butts, B. Budlong, P. Wasson, and E. White. Reconfigurable work farms on a massively parallel processor array. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 206–215. IEEE, 2008. → pages 2, 28

[14] D. Callahan, K. Kennedy, et al. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 63–76. ACM, 1987. → page 11

[15] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson. Legup high-level synthesis. In *FPGAs for Software Programmers*, pages 175–190. Springer, 2016. → page 14

[16] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). In *International Workshop on Field Programmable Logic and Applications*, pages 605–614. Springer, 2000. → page 15

[17] D. Chen, J. Cong, P. Pan, et al. Fpga design automation: A survey. *Foundations and Trends® in Electronic Design Automation*, 1(3):195–330, 2006. → page 17

[18] C. Claus, F. H. Muller, J. Zeppenfeld, and W. Stechele. A new framework to accelerate virtex-ii pro dynamic partial self-reconfiguration. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–7. IEEE, 2007. → page 26

[19] C. Claus, W. Stechele, M. Kovatsch, J. Angermeier, and J. Teich. A comparison of embedded reconfigurable video-processing architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 587–590. IEEE, 2008. → page 23

[20] S. Commuri, V. Tadigotla, and L. Sliger. Task-based hardware reconfiguration in mobile robots using fpgas. *Journal of Intelligent and Robotic Systems*, 49(2):111–134, 2007. → page 23

[21] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011. → page 14

[22] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou. Combining module selection and replication for throughput-driven streaming programs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1018–1023. IEEE, 2012. → pages 29, 37, 40, 51, 67

[23] L. Daoud, D. Zydek, and H. Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In *Advances in Systems Science*, pages 483–492. Springer, 2014. → page 2

[24] B. D. de Dinechin. Kalray mppa®: Massively parallel processor array: Revisiting dsp acceleration with the kalray mppa manycore processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, pages 1–27. IEEE, 2015. → page 19

[25] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. → page 1

[26] F. Dittmann and S. Frank. Caching in real-time reconfiguration port scheduling. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 740–744. IEEE, 2007. → page 25

[27] A. Duller, G. Panesar, and D. Towner. Parallel processing-the picochip way. *Communicating Processing Architectures*, 2003:125–138, 2003. → page 18

[28] E. El-Araby, I. Gonzalez, and T. El-Ghazawi. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(4):21, 2009. → page 23

[29] P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21 (5):313–347, 1992. → page 13

[30] K. Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974. → pages 15, 28

[31] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56. IEEE, 2000. → page 14

[32] M. B. Gokhale and J. M. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 126–135. IEEE, 1998. → page 14

[33] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, et al. A stream compiler for communication-exposed architectures. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 291–303. ACM, 2002. → page 46

[34] J. Gray. Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 17–20. IEEE, 2016. → pages 7, 48

[35] S. G. Hansen, D. Koch, and J. Torresen. High speed partial run-time reconfiguration using enhanced icap hard macro. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 174–180. IEEE, 2011. → pages 26, 94

[36] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 65–74. ACM, 1998. → page 25

[37] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling

high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, pages 144:1–144:11, 2014. → pages 51, 77

[38] J. Hegarty, R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz, and P. Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Trans. Graph.*, pages 85:1–85:11, 2016. → pages 51, 77

[39] M. Huebner, M. Ullmann, F. Weissel, and J. Becker. Real-time configuration code decompression for dynamic fpga self-reconfiguration. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 138. IEEE, 2004. → page 26

[40] B. Hutchings, B. Nelson, S. West, and R. Curtis. Optical flow on the ambric massively parallel processor array (mppa). In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 141–148. IEEE, 2009. → pages 2, 23

[41] Intel-inc. Overview for the stratix iv device family. 2016. URL https://www.altera.com/en_US/pdfs/literature/hb/stratix-iv/stx4_siv51001.pdf. → page 65

[42] C. Kao. Benefits of partial reconfiguration. *Xcell journal*, 55:65–67, 2005. → page 23

[43] R. Kastner, J. Matai, and S. Neuendorffer. Parallel programming for fpgas. *arXiv preprint arXiv:1805.03648*, 2018. → pages 5, 49

[44] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 6th international conference on Supercomputing*, pages 323–334. ACM, 1992. → page 11

[45] Khronos-Group. Openvx. 2017. URL https://www.khronos.org/openvx/. → page 51

[46] D. W. Knapp. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. Prentice Hall PTR, 1996. → page 49

[47] D. Koch. *Partial reconfiguration on FPGAs: architectures, tools and applications*, volume 153. Springer Science & Business Media, 2012. → pages xii, 24, 25

[48] D. Koch, C. Beckhoff, and J. Teich. Bitstream decompression for high speed fpga configuration from slow memories. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 161–168. IEEE, 2007. → page 26

[49] H. Kooti and E. Bozorgzadeh. Reconfiguration-aware task graph scheduling. In *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference on*, pages 163–167. IEEE, 2015. → page 87

[50] B. Krill, A. Ahmad, A. Amira, and H. Rabah. An efficient fpga-based dynamic partial reconfiguration design flow and environment for image and signal processing ip cores. *Signal Processing: Image Communication*, 25 (5):377–387, 2010. → page 23

[51] I. Kuon, R. Tessier, and J. Rose. Fpga architecture: Survey and challenges. *Foundations and trends in electronic design automation*, 2(2):135–253, 2008. → page 16

[52] S. Lange and M. Middendorf. Hyperreconfigurable architectures for fast run time reconfiguration. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 304–305. IEEE, 2004. → page 26

[53] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, pages 1235–1245, 1987. → pages 15, 23

[54] Z. Li and S. Hauck. Don't care discovery for fpga configuration compression. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 91–98. ACM, 1999. → page 26

[55] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, 1997. ISBN 0-89791-853-3. → page 51

[56] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214. ACM, 1997. → pages 11, 31

[57] D. Liu and B. C. Schafer. Efficient and reliable high-level synthesis design space explorer for fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2016. → page 52

[58] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 498–502. IEEE, 2009. → page 26

[59] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, pages 6:1–6:30, 2014. → page 65

[60] A. Makhorin. Glpk (gnu linear programming kit). *http://www. gnu. org/software/glpk/*, 2008. → pages 37, 48, 87

[61] P. Manet, D. Maufroid, L. Tosi, G. Gailliard, O. Mulertt, M. Di Ciano, J.-D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, et al. An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. *EURASIP Journal on Embedded Systems*, 2008:1, 2008. → page 26

[62] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009. → page 2

[63] O. Mencer, M. Morf, and M. J. Flynn. Pam-blox: High performance fpga design for adaptive computing. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 167–174. IEEE, 1998. → page 14

[64] R. Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004. → page 15

[65] H. Omidian and G. G. Lemieux. Janus: A compilation system for balancing parallelism and performance in openvx. In *Journal of Physics: Conference Series*, volume 1004, page 012011. IOP Publishing, 2018. → page vi

[66] H. Omidian and G. G. F. Lemieux. Automated space/time scaling of streaming task graph. *International Workshop on Overlay Architectures for FPGA (OLAF)*, abs/1606.03717, 2016. → page vi

[67] H. Omidian and G. G. F. Lemieux. Exploring automated space/time tradeoffs for openvx compute graphs. In *2017 International Conference on Field-Programmable Technology (FPT)*, Dec 2017. → page vi

[68] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72. ACM, 2006. → page 1

[69] K. Paulsson, M. Hübner, S. Bayar, and J. Becker. Exploitation of run-time partial reconfiguration for dynamic power management in xilinx spartan iii-based systems. In *ReCoSoC*, pages 1–6, 2007. → page 23

[70] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, pages 26:1–26:25, 2017. URL http://doi.acm.org/10.1145/3107953. → page 77

[71] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13. IEEE Computer Society Press, 1998. → page 1

[72] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, P. Chow, R. Wittig, H. Styles, and A. Putnam. Mpi as a programming model for high-performance reconfigurable computers. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4):22, 2010. → page 23

[73] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. *SIGPLAN Not.*, pages 175–187, 1992. → page 51

[74] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *ACM SIGPLAN Notices*, volume 27, pages 175–187. ACM, 1992. → pages 11, 31

[75] A. Severance and G. G. F. Lemieux. Embedded supercomputing in fpgas with the vectorblox mxp matrix processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013. → page 78

[76] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 117–126, 2014. → page 74

[77] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in fpgas. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(4):643–663, 2002. → page 32

[78] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini. Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 289–296, 2015. → pages 51, 77

[79] S. Taheri, J. Heo, P. Behnam, P. Veidenbaum, and A. Nicolau. Acceleration framework for fpga implementation of openvx graph pipelines. Technical report, Center for Embedded and Cyber-Physical Systems, University of California, Irvine, 2018. → page 52

[80] R. Tessier, K. Pocek, and A. DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015. → page 13

[81] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002. → page 46

[82] M. F. Tompkins. *Optimization techniques for task allocation and scheduling in distributed multi-agent operations*. PhD thesis, Massachusetts Institute of Technology, 2003. → page 87

[83] S. Toscher, T. Reinemann, and R. Kasper. An adaptive fpga-based mechatronic control system supporting partial reconfiguration of controller functionalities. In *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*, pages 225–228. IEEE, 2006. → page 23

[84] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., the 5th Annual IEEE Symposium on*, pages 22–28. IEEE, 1997. → pages 26, 94

[85] J. L. Tripp, M. B. Gokhale, and K. D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3), 2007. → page 14

[86] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, C. Watnik, A. T. Tran, Z. Xiao, et al. A 167-processor computational platform in 65 nm cmos. *IEEE Journal of Solid-State Circuits*, 44(4):1130–1144, 2009. → page 19

[87] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134. IEEE, 2010. → page 14

[88] G. K. Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992. → pages 47, 56

[89] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. Prism-ii compiler and architecture. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 9–16. IEEE, 1993. → page 14

[90] L. Wirbel. Ambric lives on, in a parallel universe, 2011. URL https://www.edn.com/electronics-blogs/fpga-gurus/4409421/Ambric-Lives-On-in-a-Parallel-Universe. → page 21

[91] D. Wo and K. Forward. Compiling to the gate level for a reconfigurable co-processor. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 147–154. IEEE, 1994. → page 14

[92] M. E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Citeseer, 1992. → page 11

[93] Z. Xiao, D. Koch, and M. Lujan. A partial reconfiguration controller for altera stratix v fpgas. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE, 2016. → page 93

[94] Xilinx-inc. Axi4 stream interconnect. 2017. URL https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html. → page 52

[95] Xilinx-inc. revision enables responsive and reconfigurable vision systems. 2018. URL https://www.xilinx.com/products/design-tools/embedded-vision-zone.html. → page 50

[96] Xilinx-inc. Vivado high-level synthesis. 2018. URL https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. → pages 5, 49, 50

[97] Xilinx-inc. 7 series fpgas data sheet: Overview. 2018. URL https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. → page 66

[98] Xilinx-inc. Dma v7. 1, logicore ip product guide, vivado design suite, 2018. 2018. URL https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html. → page 56

[99] Y. Yankova, G. Kuzmanov, K. Bertels, G. Gaydadjiev, Y. Lu, and S. Vassiliadis. Dwarv: Delftworkbench automated reconfigurable vhdl generator. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 697–701. IEEE, 2007. → page 14

[100] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An asynchronous array of simple processors for dsp applications. In *Solid-State Circuits Conference, 2006. ISSCC 2006. Digest of Technical Papers. IEEE International*, pages 1696–1705. IEEE, 2006. → pages 18, 19

[101] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar. Design space exploration of multiple loops on fpgas using high level synthesis. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 456–463, 2014. → page 52