Compiler Design for Accelerating Applications on Coarse-Grained Reconfigurable Architectures

by

Mahesh Balasubramanian

A Dissertation Presented in Partial Fulfillment
of the Requirement for the Degree
Doctor of Philosophy

Approved October 2021 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Chaitali Chakrabarti
Fengbo Ren
Laura Pozzi

ARIZONA STATE UNIVERSITY

December 2021

ABSTRACT

Coarse-Grained Reconfigurable Arrays (CGRAs) are emerging accelerators that promise low-power acceleration of compute-intensive loops in applications. The acceleration achieved by CGRA relies on the efficient mapping of the compute-intensive loops by the CGRA compiler onto the CGRA. The CGRA mapping problem, being NP-complete, is performed in a two-step process, scheduling, and mapping.

The scheduling algorithm allocates timeslots to the nodes of the DFG, and the mapping algorithm maps the scheduled nodes onto the PEs of the CGRA. On a mapping failure, the initiation interval (II) is increased, and a new schedule is obtained for the increased II. Most previous mapping techniques use the Iterative Modulo Scheduling algorithm (IMS) to find a schedule for a given II. Since IMS generates a resource-constrained ASAP (as-soon-as-possible) scheduling, even with increased II, it tends to generate a similar schedule that is not mappable and does not explore the schedule space effectively. The problems encountered by IMS-based scheduling algorithms are explored and an improved randomized scheduling algorithm for scheduling of the application loop to be accelerated is proposed.

When encountering a mapping failure for a given schedule, existing mapping algorithms either exit and retry the mapping anew, or recursively remove the previously mapped node to find a valid mapping (backtrack). Abandoning the mapping is extreme, but even backtracking may not be the best choice, since the root of the problem may not be the previous node. The challenges in existing algorithms are systematically analyzed and a failure-aware mapping algorithm is presented.

The loops in general-purpose applications are often complicated loops, i.e., loops with perfect and imperfect nests and loops with nested if-then-else's (conditionals). The existing hardware-software solutions to execute branches and conditions are inefficient. A co-design approach that efficiently executes complicated loops on CGRA

is proposed. The compiler transforms complex loops, maps them to the CGRA, and lays them out in the memory in a specific manner, such that the hardware can fetch and execute the instructions from the right path at runtime.

Finally, a CGRA compilation simulator open-source framework is presented. This open-source CGRA simulation framework is based on LLVM and gem5 to extract the loop, map them onto the CGRA architecture, and execute them as a co-processor to an ARM CPU.

To my parents, Balasubramanian & Rama,

to my wife, Shamini,

and to all my well-wishers.

ACKNOWLEDGEMENT

I would like to thank my advisor Dr.Aviral Shrivastava, who has been a great mentor. His support during my toughest times has helped me get through them and strive. This thesis would not have been possible without his guidance.

I would like to thank Dr.Chaitali Chakrabarti, Dr.Fengbo Ren, and Dr.Laura Pozzi, for supervising my thesis and for giving important ideas to develop my research and fine-tune the thesis.

I am grateful for the summer internships at Lawrence Berkeley National Laboratory (LBNL), where I had a chance to collaborate with Dr. Prabhat and Dr.Kris Bouchard. Their inputs and mentorship shaped my research direction and this thesis. Thanks to all the people that I have met in Berkeley, especially, Brandon Cook, Maximilian Dougherty, Pratik Sachdeva, Dr.Trevor Ruiz, and Dr.Sharmodeep Bhattacharyya. A special thanks to Dr.Grzegorz Muszynski for his friendship and intellectual discussions.

I would like to thank my lab mates, Shail Dave, Moslem Didehban, Dheeraj Lokam, Edward Andert, Mohammadereza Mehrabian, and Mohammad Khayatian, who have been great support and for providing a great research environment.

Finally, I would like to thank my family. My dad, who has been an emotional and financial support for three decades. My mom, Rama, for her prayers. My in-laws, Radhika and Rajaganesh, for believing in me. Most importantly, my wife, Shamini, whose patience and love is unparalleled.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

The advancement of the Internet and data collecting devices have increased the demand for high-performance, low-power computing alternatives. All mobile devices collect, process, and communicate data. Analyzing the collected data to extract meaningful information is compute-intensive [1] and often limited by the thermal, power, and resource constraints [2]. Efficiency in accelerating the compute-intensive sections is now being achieved through the use of custom accelerators, e.g., Application-Specific Integrated Circuits (ASIC), spatial architectures like Eyeriss [3], DianNao [4], EIE [5], MAERI [6], etc. for deep learning applications [7], NERO[8], SODA[9], Associative Processor[10] for accelerating stencil computation, SODA[11] for Software Defined Radio applications etc.

Due to the immense influx of the data, the performance of the application analyzing the data is of utmost importance [12]. The existing compilers and architectures exploit the data-level parallelism by vectorizing the applications. The resultant parallel code generated, which is of the Single Instruction Multiple Data (SIMD) fashion, is accelerated/streamed in vector units with variable width. Theoretically, as the vector width increases, the speedup achieved by such SIMD vector units should be proportional. But the speedup is restricted to highly parallel application, and many of the application with control-flow divergence does not benefit much acceleration from SIMD units [13].

The applications must be tuned to a particular architecture framework to achieve maximum performance. [12]. Graphics Processing Units (GPU) are successful in accelerating both graphics-based and general-purpose applications. Extensive research

is being carried out to handle the data-dependence and control-flow problem GPUs. The downside of the GPU accelerator technology is the programmers' ability to choose the *kernel* from the application to be accelerated on the GPU and especially program it for the hardware. OpenCL [14] and CUDA [15] are the widely used parallel-languages for accelerating kernels on GPUs. So accelerating an application using the GPU framework is not simple [16, 17, 18].

Along with the GPUs, Field Programmable Gate Arrays (FPGAs) are popular in accelerating applications at low power. For power-critical systems, FPGAs have been a promising accelerator solution. With the increasing use of data-centers and edge computing devices, FPGAs have permeated into the domain-specific acceleration like artificial intelligence, high-performance computing, etc. [19, 20]. Like GPUs, FPGAs suffer from programmability issues where the programmers need to write their applications in Hardware Descriptive Languages (HDL) [21].

Although custom accelerators can arguably achieve the highest acceleration efficiency (performance/power), using them may result in poor code portability. If the accelerator changes in the next generation of hardware, then the code-base needs to be ported to the new accelerator technology. There has been a surge of application/domain-specific accelerators like Eyeriss [3], Diannao [4], Marvel [22]. The compiler and hardware are optimized for performance, power, area, and energy for a particular application domain like Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), etc. As formulated in Marvel [22], the input application loops should have the following constraints: (1) Perfectly nested without any conditional statements. (2) Perfectly nested loops must not have any anti, flow, or output dependencies. (3) Can be freely reordered for compiler optimizations like tiling, loop reordering, etc. General-purpose applications have compute-intensive loops that do not conform to these restrictions. Accelerating them is challenging as they may contain conditional

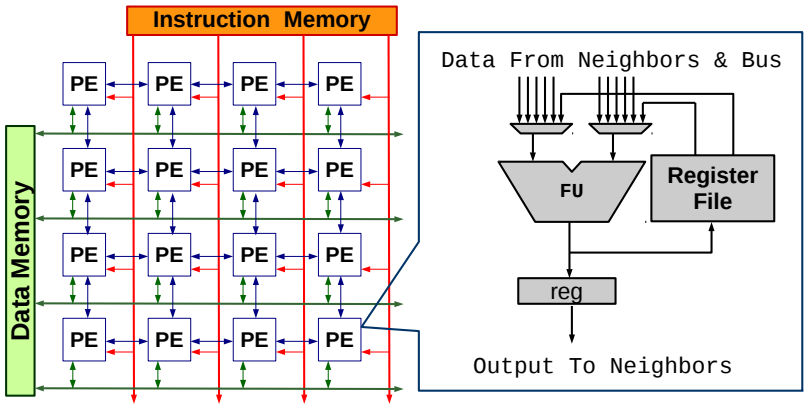


Figure 1.1: A 2-d Mesh 4×4 CGRA.

statements that are arbitrarily nested, loop carried dependencies, etc.

FPGA or Field-Programmable Gate Arrays is the popular general-purpose accelerator used. However, FPGAs achieve low efficiency in acceleration due to their bit-level programmability [23]. CGRAs, on the other hand, provide high efficiency at low power due to their word-level and arithmetic-operation level programmability [24].

In a heterogeneous computing paradigm, where the CPU core is equipped with vector processing and other accelerators, a single accelerator that can accelerator majority of the applications is still absent. The search for a generic accelerator that can accelerate all application domains with decent quality and at low-power consumption is the holy grail of the modern computing research. Coarse-grained Reconfigurable Arrays (CGRA) is a promising candidate for generic accelerators that may be able to accelerate a wide-range of applications with pretty good quality.

Figure 1.1 shows a 4x4 CGRA. A CGRA consists of a simple 2-D grid of Processing Elements or *PEs*. Each PE contains Functional Units (FUs), which can receive instructions from the instruction memory, compute arithmetic operations with the data received from the data memory or the neighboring PEs. Each PE consists of MUXes to select the inputs from its neighbors and a register file to store intermittent

data. The 4×4 CGRA shown in Figure 1.1, can perform 16 operations simultaneously, making it highly parallel. The attractive feature about CGRA is that the hardware is simple, and customizable since most of the effort for acceleration is shifted to the compiler.

Previous CGRA designs have demonstrated high degrees of acceleration and high power efficiency (performance/power) for signal processing applications. The ADRES [23] experimentally showed the power-efficiency CGRAs to be 60 GOps/W using 32nm technology. Good compilers are needed to obtain a good quality mapping of performance-critical loops from applications from a wide variety of domains to demonstrate the usefulness and applicability of CGRAs as a general-purpose accelerator. However, existing state-of-the-art CGRA compilers still suffer from problems such as not finding a mapping for a loop, achieving poor mapping (low performance) for a loop, and yielding high compilation time.

The most common way to use CGRAs is to employ them as co-processors to CPU cores or processors, to speed up and power-efficiently execute compute-intensive applications – similar to GPUs. The compute-intensive loops of an application are offloaded to CGRAs for parallel execution. The remaining un-parallel code section is executed on the CPU core. CGRA compilers achieve this roughly in the following steps: (i) identify the loops to be accelerated on the CGRA, (ii) extract and convert the loops into a Data Flow Graph (DFG) honoring the data and control dependencies, and (iii) schedule the nodes of the DFG onto a time extended CGRA, (iv) map the nodes of the DFG onto the PEs for execution. The DFG is software pipelined onto the CGRA graph, so the PEs can communicate the computed value to achieve a correct execution in a parallel fashion. The earliest time at which the next iteration of the loop can start in software pipelining is called the *initiation interval* (II).

Many techniques have been proposed to solve NP-complete[25] mapping problem

of CGRAs efficiently[26, 27, 28, 29, 25, 30, 31, 32]. A mapping failure can occur in the mapping step due to the limited connectivity among the PEs of the CGRA. When dependent operations are scheduled in non-contiguous timeslots, the value computed by the source PE should be routed to the destination PE. This is commonly referred to as the routing problem. One solution is to route the operands through the PEs in the intermediate timeslots. Since routing and mapping attempts often fail, existing CGRA mapping techniques have heavily focused on solving the problem encountered in the mapping and routing step.

My thesis statement is: Effective compiler technology and co-design approach is crucial in achieving efficient acceleration of applications on CGRAs.

This dissertation aims to explore and analyze the following problems when accelerating an application on CGRA:

- **Randomized Scheduling Algorithm:** Iterative Modulo Scheduling (IMS) algorithm is widely used by the existing state-of-the-art CGRA mapping techniques to schedule the application loops' Data-Flow Graph (DFG) onto the CGRA. The IMS-generated schedule does not change much, even when more resources are added towards the bottom of the CGRA graph. The resource-constrained ASAP schedule will be almost identical to the one obtained before, and the extra resources are not used! As a result, the mapping algorithm keeps on exploring the schedule space with the same schedule, and often no mapping can be found, even after huge increases in the II. Hence, this creates a need for an enhanced scheduling algorithm that explores the schedule space to increase the mappability of the compute-intensive loops. The steps to mitigate this issue is addressed in chapter 4.

- **Failure-Aware Mapping Algorithm:** The problem with existing mapping techniques is that they do not investigate the reasons behind a mapping failure. Many of the existing techniques abandon the current mapping and retry from scratch every time they are unable to map a node. Some mapping techniques use a more systematic method e.g., backtracking, to solve the problem. Backtracking recursively unmaps the last mapped node and then tries to obtain a mapping. However, the order in which the backtracking algorithm unmaps the nodes may not be efficient – in the sense that the problematic node may be very deep in the backtracking stack. As a result, existing techniques may require a long time to obtain a mapping or may fail to find a mapping in a limited time. The only reason for a mapping failure in step (iv) is when there is no connection between the PE of the failed node and the PEs to which predecessors and successors of the failed node are mapped. This issue can be solved by: changing the mapping of the failed operation in its time slot, or changing the mapping of the successor of the failed node, or by changing the mapping of the predecessor of the failed node, and so on. The existing techniques are failure-unaware that worsens the compilation time and sometimes the performance. A failure-aware mapping algorithm is presented in chapter 5

- **Efficient Mapping of Loops with Conditionals:** Complicated loops like nested loops and loops with arbitrary conditionals are hard to map by the current compilers. Although full and partial predication can map arbitrarily nested conditional loops, they suffer from poor performance and increasing DFG size, respectively. Co-design approaches like Dual-Issue and PSB cannot map nested loops due to their rigid hardware. A compiler-architecture co-design is presented in chapter 6 to map and execute arbitrarily nested conditional loops.

- **CGRA Compilation Simulation Framework:** There is very few open-source compiler simulator framework for CGRA in the research community. The available frameworks like CGRA-ME [33] are restrictive (takes a lot of compilation time) and not cycle-accurate. In chapter 8 we present a CGRA compiler based on LLVM and CGRA architecture simulator based on gem5 for accelerating various compute-intensive application loops.

Chapter 2

BACKGROUND AND TERMINOLOGY

2.1 Mapping a loop DFG onto the CGRA

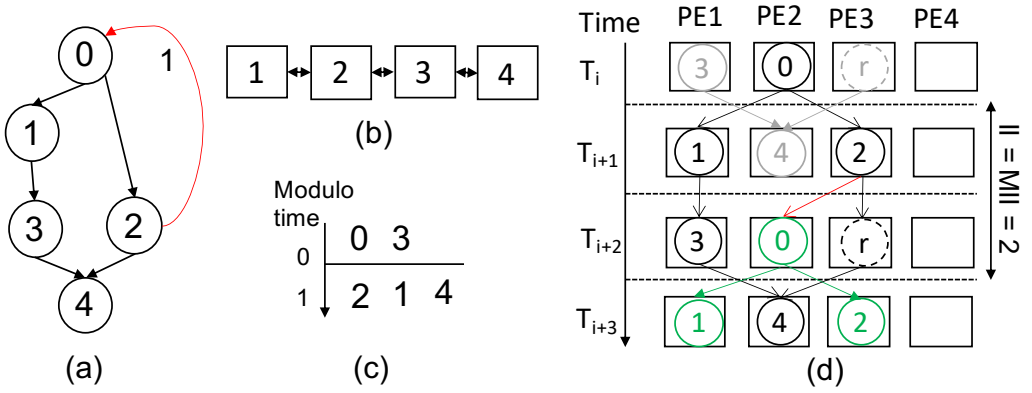


Figure 2.1: (A) DFG of an Application Loop. (B) a 1x4 CGRA Target Architecture. (C) an IMS Schedule of Nodes of DFG. The X-axis Is the Modulo Time. (D) a Mapping of the Scheduled Nodes on the Time-extended CGRA (TEC).

CGRA compilers, in general, first create the Data Flow Graph (DFG) $D = (V, E)$ of a compute-intensive loop, where V refers to the nodes of the loop and E refers to the edges (data dependencies between nodes) in the graph. The constructed DFG is then software pipelined using IMS[34], where each node is assigned a schedule time at which it should be executed.

Fig 2.1(a) shows the DFG of a loop, and Fig 2.1(b) shows the target CGRA architecture. The CGRA mapping process is usually divided into two phases, namely, (i) scheduling phase, where the nodes of the DFG are allotted a timeslot for execution, and (ii) mapping phase, in which the nodes of the DFG are allotted a PE at the scheduled timeslot where the node is executed honoring the data dependencies.

Most of the recent mapping algorithm uses Iterative Modulo Scheduling (IMS) [34] for scheduling the nodes. The schedule of the DFG nodes is shown in Fig 2.1(c), considering the resource and the recurrence cycle constraints. IMS schedules the operations at the theoretical minimum II. The theoretical minimum II (MII) is the maximum of resource constraint minimum II ($ResMII$) and recurrent constraint minimum II ($RecMII$), shown in Equation 2.1 and Equation 2.2, respectively.

$$ResMII = \frac{Total\_nodes\_DFG}{No\_of\_PEs} = \left\lceil \frac{5}{4} \right\rceil = 2 \tag{2.1}$$

$$RecMII = No\_of\_Recurrent\_cycles = 1 \tag{2.2}$$

$$MII = max(ResMII, RecMII) = max(2, 1) = 2. \tag{2.3}$$

After computing the MII from Equation 2.3, IMS schedules the nodes of the DFG for the MII. After scheduling, the nodes are then mapped onto the PEs of CGRA such that the dependent operands can be routed from the PE on which the source operation is mapped to the PE on which the destination operation is mapped through either registers, memory, or paths in the CGRA graph. A register can be used to route operands when the dependent operation is mapped to the same PE as the source operation. Memory can be used to route operands, but that requires inserting additional load and store instructions. A path is a sequence of edges and nodes in the CGRA graph that connects two PEs. In the simplest case, a path is just a single edge. For simplicity, the mapping shown in figure 2.1(d) uses only edges to route dependencies. In this mapping, node 0 of iteration i is mapped to $PE2$ at time T_i, nodes 1 and 2 are mapped to PEs, $PE1$ and $PE3$ respectively, at T_{i+1}. Similarly, nodes 3 and r of i^{th} iteration are mapped in $PE1$ and $PE3$ respectively at T_{i+2}. Node 4 of i^{th} iteration is mapped at $PE2$ at T_{i+3}. We can observe that

node 0 of iteration $(i+1)^{th}$ in time T_{i+2} (shown in green) is scheduled/pipelined even before the completion of the i^{th} iteration. In modulo scheduling, the interval in which successive instructions can begin execution is called the *Initiation Interval* (II) [34]. II is considered the performance metric for DFG mapping onto CGRA, as the total number of cycles required to execute a loop will be proportional to the II. Similarly, nodes 1 and 2 of the $(i+1)^{th}$ iteration is scheduled along with node 4 of i^{th} iteration at time T_{i+3}. Modulo schedule repeats itself every II cycle, in this case, II=2. The MII shown in Figure 2.1(d) is the theoretical minimum II that is possible for the given DFG and the CGRA resources.

Chapter 3

A LITERATURE SURVEY ON COMPILATION TECHNIQUES FOR CGRA

3.1 Background

CGRAs exhibit a wide variety of architecture differing in size, functionality, interconnect, etc. Figure 3.1 shows three different CGRAs based on their PE connectivity. Figure 3.1(a), (b) and (c) show mesh, torus and 1-hop CGRA interconnect variations. [26] and [35] explain in detail various interconnections in CGRAs in their research. In the literature, two of the most widely used PE configurations are, (i) homogeneous and (ii) heterogeneous. In the homogeneous CGRA, all the PEs are designed to execute all the operations, whereas in heterogeneous CGRA special PEs are used to execute expensive operations.

Due to the flexibility of the PEs and interconnect topologies, there have been numerous interconnect designs for CGRAs developed over the years. Likewise, CGRAs exhibit a wide variety of designs based on the register file (RF)architectures, explained in detail in [26, 27, 36]. The categories include Local RF dedicated to each PE, Global/Central RF shared by all PEs, Shared RF shared with four neighboring PEs [29], etc. The RFs are not limited to the above-mentioned categories, and most importantly, multiple RF configurations can be used in a CGRA. There have been several implementation of CGRA architectures such as MorphoSys [37], MATRIX [38], ADRES [26], RaPiD [39], HSRA [40], DRAA, RSPA, etc., for accelerating compute-intensive sections of applications, predominantly loops. There has been an increased interest in CGRA based data flow architectures in the bio-medical and machine learning domain. Samsung has been developing an in-house processor called

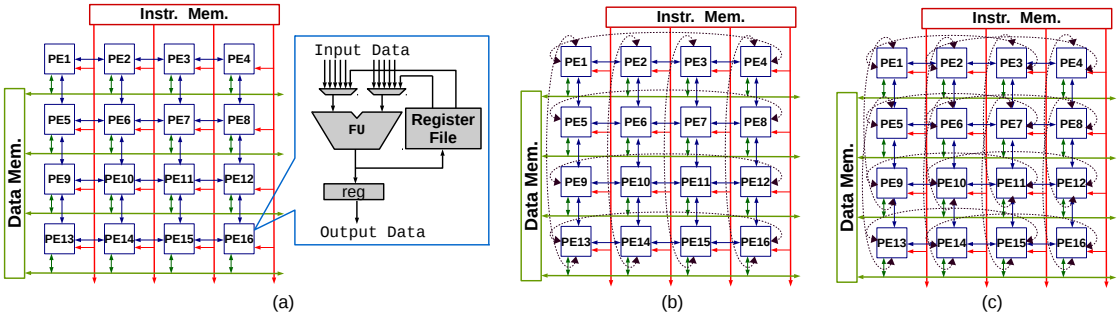


Figure 3.1: (a) A 4×4 CGRA with simple 2D-mesh interconnect. Each PE consists of an FU, muxes for data communication and Register Files to store intermediate data or constants. PEs are connected to the data memory and instruction memory. (b) a 4×4 CGRA with torus interconnect network. (c) A 4×4 CGRA with 1-hop interconnect network.

Ultra-low Power Samsung Reconfigurable Processors (ULP-SRP) [41], especially for biomedical applications like ECG monitoring, etc. Wave Computing Inc., [24] has been developing CGRA based architectures for data-flow applications for machine learning applications. Recent work from the Massachusetts Institute of Technology (MIT) and NVIDIA called Eyeriss [42] uses CGRA like architecture for energy-efficient Convolutional Neural Networks (CNN). These works are highly improving the prospects CGRA like data-flow architectures in mainstream computing. In this chapter, we will explore and categorize various general-purpose CGRA compilers, in detail.

Previous research works like [43, 32] have shown that CGRAs, having a simpler hardware compared to GPUs, can accelerate non-parallel loops in an energy efficient manner. In CGRA most of the complexity in terms of reconfiguration is shifted to the compiler, therefore the hardware is less complex. The efficiency and acceleration of application by CGRAs, greatly rely on the mapping of application kernels by the compiler onto the CGRA by better utilization the hardware resources available [44]. The CGRA compiler (i) identifies the compute-intensive loop, (ii) converts the kernel

into a data graph [1] , with V vertices/nodes, and E edges/dependences, (iii) Maps the data graph onto the CGRA architecture (identifying which PEs should execute which operation in which cycle for correct functionality) (iv) creates instructions for PE functionality and (v) loads the instructions onto the configuration memory for PEs for the correct functionality of the selected kernel. Some of the compilers developed for CGRAs are DRESC compiler framework [26], REMUS [45] architecture and compiler framework etc.

This survey analyzes in detail the various compiler techniques for accelerating compute-intensive sections on CGRA. We also try to classify the compiler techniques to understand the nature of each technique and the type and construct of loops that they can accelerate. The classification provided in this chapter is just a research guideline, and the categories can overlap for some techniques. For example, some spatiotemporal mapping techniques can utilize registers for routing. In such cases, we have categorized them in the section where the novelty of the approach seems to fit. Each section starts with an overview of the problem definition, followed by a brief explanation of each technique.

3.2 Classification of CGRA compiler techniques

The classification of CGRA compiler techniques is shown in Table 3.1. After identifying the kernel to be accelerated, the compiler converts the kernel into a data graph [2] and tries to map them onto the CGRA architecture. The mapping techniques can further be sub-classified into spatial mapping techniques, like [44, 46, 47, 48] and spatiotemporal mapping techniques. We further classify spatiotemporal mapping

[1] A data graph can be Data Flow Graph (DFG), Data Dependence Graph (DDG), Data Acyclic Graph (DAG), Control Data-Flow Graph (CDFG) etc., depending on the compiler technique used.

[2] The data graph can be either a Data-flow graph (DFG), Data-Acylic Graph (DAG), Data-dependence Graph (DDG), etc., depending on the implementation. We generally mention this group as data graphs.

Classification	Subcategories		Approaches
Mapping Algorithms	Spatial Mapping		[44, 46, 47, 48]
	Spatiotemporal Mapping	Incremental	[27, 28, 29, 49, 50]
		Exploratory	[26, 51, 52] [53, 54, 55] [56, 57]
		Graph-based	[25, 31] [30, 58, 59]
Resource Utilization	Data-Memory Aware		[47, 48, 60] [61, 62, 63]
	Register Aware		[31, 28, 64, 36]
Generalization	Nested Loops		[65, 66, 67, 68] [69, 70, 71] [44, 72]
	Branch Aware		[73, 74, 75] [76, 43, 77] [78, 79]
Multi-threading			[80, 81, 82]

Table 3.1: Classification of Compiler Techniques for CGRAs.

techniques into three categories namely, incremental techniques [27, 28, 83, 29, 84, 49, 34, 50], exploratory techniques [26, 51, 85, 86, 87, 56, 88, 52, 57] and graph based techniques like [25, 30].

The second category, Resource Utilization, delves into the details of techniques that intelligently use available CGRA architectural resources to better accelerate the kernel. Based on the methods we came across, we categorize this section into (i) data-memory aware techniques [60, 61, 62, 89] that try to better utilize the data-memory connected to the CGRA architecture for better performance, and (ii) register-aware techniques [64, 31, 90, 36] that utilize the register file (RF) architecture in the PEs or have an external register file to route the data variables used for the loop execution. There are different RF architectures explored in the CGRA community. Techniques like [23, 27, 36] show various RF architectures for CGRA.

The third category, Generalization, includes compiler techniques that try to solve

the problem of accelerating application kernels having a more generalized problem like that of nested loops and kernels with if-then-else conditional statements. We sub-categorize Generalization into techniques for nested loops like [65, 66, 68, 72] and branch-aware (loops with if-then-else) techniques like [76, 91, 92, 43, 77, 73, 78, 74]. The final category, Multi-threading, explains the techniques proposed in [80, 81, 82] to solve the problem of accelerating multi-threaded kernels efficiently on CGRA.

3.3 Mapping Algorithms

The goal of mapping techniques is to find a valid mapping of the data graph of the loop onto the PEs in space and time. Firstly, the loops in the target application are identified by the compiler, which is then converted into a graph like a Data Flow Graph $D(V_i, E_i)$, where vertices or nodes (V_i) are the loop operations, and edges (E_i) are the data dependencies [93]. Secondly, a resource graph is constructed $C(V_c, E_c)$, where V_c is the PE in the CGRA and E_c refers to the routing paths between the PEs. This resource graph may be time extended (in the case of spatiotemporal mapping) and not time extended (in the case of spatial mapping). Finally, finding a valid mapping between graph $D(V_i, E_i)$ to $C(V_c, E_c)$ is the goal of the techniques discussed in this section. Generally, existing mapping algorithms can be categorized into spatial or spatiotemporal mapping techniques.

3.3.1 Spatial Mapping

Spatial Mapping techniques try to map the data graph onto the CGRA in space, meaning the mapping for PEs is fixed throughout the kernel execution. If a node is mapped to a PE, then that PE cannot be used by other nodes during the execution of the loop. An example loop mapping is shown in Figure 3.2(a)-(d) . A simple loop kernel to be accelerated and the 3×2 CGRA architecture is shown in Figure 3.2(a)

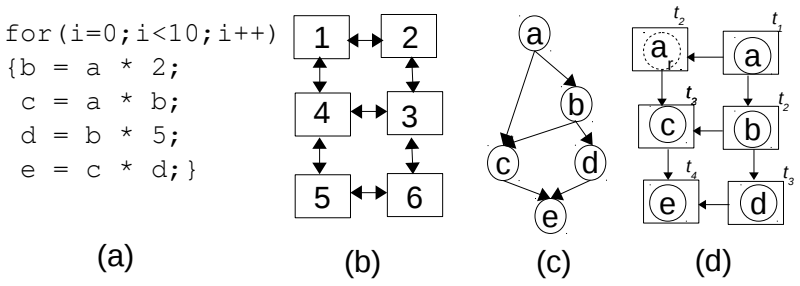


Figure 3.2: (A) an Example Kernel Code (B) 3×2 CGRA Architecture. (C) DFG of the Loop (D) Spatial Mapping of DFG onto 3×2 Architecture as Shown in (B) Considering the Data Dependencies from the Original DFG. A Routing Node a_r (Dotted Edges) Is Added to Communicate the Value of a to c.

and (b), respectively. The compiler first converts the loop kernel into a data graph as shown in Figure 3.2(c), followed by a valid spatial mapping by the compiler with II=4, as shown in Figure 3.2(d). At cycle time t_1 operation a is executed followed by a_r and b operations at time t_2. Nodes c and d are executed in time t_3. Finally, node e is executed at time t_4. A routing node a_r is introduced to communicate the value of a to c via PE resources.

A drawing method algorithm Split & Push technique, based kernel mapping was proposed in [44] called SPKM. The general idea of this method is to find a valid spatial mapping by the recursive use of the Split&Push graph technique to map the nodes of the Data Acyclic Graph (DAG) onto the CGRA. First, all the nodes of the DAG are mapped to one PE, and the nodes are systematically moved to different PEs by using a cut to separate the vertices into two distinct groups [44]. This step is repeated until each PE gets a unique node of the DAG.

In [46], the kernel is first converted into data graph called the "kernel tree" [46]. The algorithm analyzes the nodes that can be scheduled together on a PE. These operations are clustered together to create a cluster graph. The clustered graph is then scheduled and routed (laid-out) onto the CGRA architecture using an Integer

Linear Programming (ILP) [46].

A $2D$ spatial mapping algorithm considering memory interfaces was proposed in [47]. The heuristic starts with a PE-operation tree (similar to a Data Flow Graph) and then clusters the PE operations so that the clusters can be placed in each row of the target $DRAA$ [48] architecture of CGRA. During this placement the memory operations placed in the row in less than or equal to the total memory buses allocated for the row. For example, if there are only two memory ports in a row of CGRA, a cluster can have a maximum of two memory operations. The cluster data transfer is done via mapping them onto CGRA routing resources. A Brute-Force technique is used to determine the vertical placements of individual node placements in the row. The data transfers between clusters are placed closer to make the data transfer between the rows easy. [48] builds on [47], with additional grouping of memory operations that reads the same memory address, albeit at different iterations, called the *"alignable" nodes*[48]. Due to this grouping, two additional constraints are introduced to get a valid mapping. 1) The memory sharing alignable nodes should be placed in the same row and 2) the difference of the pipeline stages of two alignable nodes should be equal to the difference of iteration of their memory operations [48].

Analysis and Discussion: In spatial mapping, the configuration of the PEs does not change throughout the execution. The key challenges in spatial mapping algorithms are, 1) since there is no time-extended mapping (using the temporal dimension of PEs for mapping) all the nodes are mapped onto the CGRA architecture, but only a few nodes are executed at a time 2) number of PEs in the CGRA should be equal to or greater than the operations in the data graph. If the number of nodes of the data graph is greater than the available resources, this mapping technique cannot find a valid mapping.

3.3.2 Spatiotemporal Mapping

The techniques in this section overcome the resource availability problem of spatial mapping by time extending the CGRA resources and mapping the nodes of the data graph at a different time maintaining the data dependencies of the data graph, thus mapping a data graph in space, and time.

The objective of spatiotemporal mapping techniques is to map the kernel data graph to Time extended CGRA (TEC). To exploit the operation level (instruction-level) parallelism of the data graph and to maximize resource utilization these techniques overlap the consecutive iteration of the loop called *Software pipelining. Initiation Interval (II)* is the time between consecutive iterations of the loop at the time of mapping. For most of the techniques in this section, the performance metrics are *(II)*. Figure 3.3(a) shows a loop kernel to be mapped onto a 2×2 CGRA as shown in Figure 3.3(b). Figure 3.3(c) shows a DFG of the loop kernel. Figure 3.3(d) shows the Time-Extended CGRA and the mapping of the DFG honoring the data dependencies with II=2, which means that for every two cycles a new loop iteration can begin. The spatiotemporal mapping techniques are broadly classified into Incremental, Exploratory, and Graph-based methods based on the mapping style and heuristics used to achieve valid mapping at lower II. Predominantly, the techniques employ two major steps, (i) Scheduling and (ii) Place and Routing (P&R). In Scheduling. the compiler tries to schedule the data graph nodes onto the TEC tags the timing for each node. In the P&R stage, the compiler maps and routes the nodes onto the PEs. The spatiotemporal mapping techniques are further classified into three sub-categories, namely, Incremental, Exploratory, and Graph-based techniques.

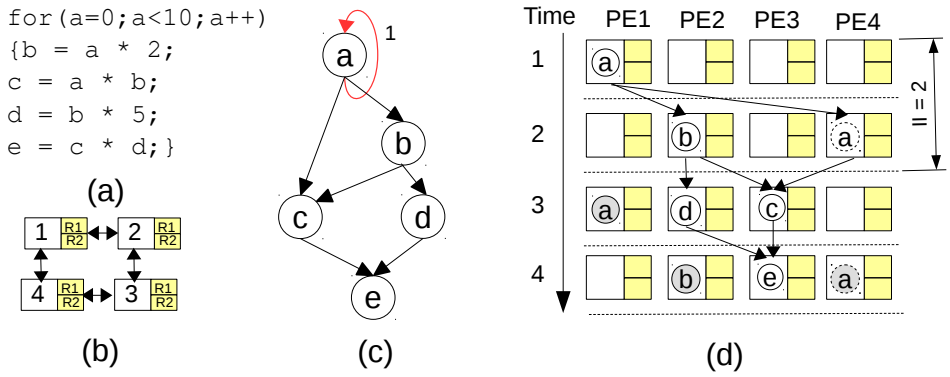


Figure 3.3: (A) an Example Kernel Selected for Execution on CGRA (B) 2×2 CGRA Architecture. (C) DFG of the Loop with One Recurrent Edge on Node a with Weight 1, (D) a Valid Spatiotemporal Mapping Considering the Data Dependencies. Next Iteration Can Begin in Time 3 Shown by Darker Nodes with II=2.

Incremental Techniques:

The techniques in this category try to find a valid mapping by either node-centric or edge-centric approaches. The heuristics start with the tightest II possible and increase ($Increment$) the II until a valid mapping is found.

The modulo graph embedding technique was proposed in [27]. This technique is a node-centric approach i.e., the nodes are placed in $3D$ schedule space which is $number\_of\_PEs \times II$. The placement of nodes is based on the height of the order of dependence. Subsequently, the placed nodes are then routed based on the *affinity values*. *Affinity value* is the measure of the distance of the common consumer in the DFG. To get a valid mapping, the *skewed* scheduling space approach is proposed where a narrow but tall scheduling space is created. To improve the II and reduce the compile-time, a more sophisticated approach using *Edge-Centric* approach, introduced by [28], was called "Edge-Centric Modulo Scheduling" (EMS). Firstly, the initial DFG is reduced by collapsing a few nodes (especially nodes with one source or one destination). High fan-out edges are ignored and node clusters are created.

During scheduling edges are given priorities, for example, recurrent edges [3] are give more priority, followed by simple edges, and high-fan out edges are given low priority. Then an integrated placement and routing function to get a valid mapping of the kernel onto the CGRA architecture. In addition to traditional EMS, a technique introduced by [94] uses a flattening-based approach to map DFGs onto CGRA of stream graphs-based applications like StreamIt.

Recurrence cycles in the data graph pose a greater challenge in mapping. The quality of mapping is affected greatly by the incremental techniques that we have seen so far. To address this issue, a modulo mapping technique considering recurrence cycles was proposed in [29]. The technique starts by clustering the nodes of a recurrence cycle into a single entity, thereby transforming the DFG into a data acyclic graph. All the incoming nodes of this recurrence cluster are mapped before mapping the recurrence cluster. In this way, the cluster that has all the incoming nodes to the recurrent cluster is scheduled first. For P&R of the DAG onto the CGRA PEs, this technique uses EMS [28] algorithm.

[49] proposed an efficient software-based Runtime Binary Translation Virtual Machine on LLVM-JIT compiler, which is a complete LLVM approach for mapping kernel DFG. The application C code is converted into an intermediate representation (IR) using clang. The framework has three main sections, (i) *L1JIT*, (ii) Monitor and, (iii) *L2JIT*. L1JIT identifies the innermost of the applications and inserts calls to the monitor function. The monitor function is a counter-based profiler, which checks if the number of loops is less than a given threshold value, 50 in the experiments. The loops beyond the threshold are executed in the host processor. After the monitor stage, the L2JIT block is called. In the L2JIT block, the DFG of the kernel is

[3] Recurrence cycles or recurrent edges are formed when a set of operations in the DFG is dependent on the result of the previous loop iteration.

extracted, and CGRA configurations are created using the EMS algorithm.

An interconnect-aware mapping algorithm was proposed in [50]. This algorithm starts with extracting the Control-Data Flow Graph (CDFG) of the loop. Then the available operation in the current cycle in A_{avail} list and list of PEs are stored in a list called $PElist$, are created. The $PElist$ and A_{avail} are then ordered using operations cost C_op and connectivity cost of PE C_{PE}, cost functions. The scheduling algorithm maps from the lowest C_{PE} followed by the routing algorithm, which checks the availability of connection (route) between candidate operation to the candidate PE in the current cycle. The operations in the $PEList$ are scheduled and mapped onto the PEs. The cycle is incremented, only if the list is exhausted [50].

Exploratory Techniques:

These techniques try to achieve a valid DFG onto a $3D$ CGRA schedule space using complex algorithms (like Simulated annealing, genetic algorithm, etc.) to explore the solution space and obtain a near-optimal mapping of the data graph onto the PEs.

DRESC compiler framework was proposed in [26] for their target ADRES CGRA architecture. The technique starts with a Data Dependence Graph (DDG) and a Modulo Resource Routing Graph (MRRG), which is identical to Time Extended CGRA (TEC). The algorithm starts with an initial modulo scheduled mapping with resource overuse. A cost function is proposed and the algorithm iteratively reduces the resource overuse until the overuse is eliminated. Then a *Simulated Annealing* based algorithm is used by exploring the solution space. For the first few notes, the temperature function is high, meaning all the placement of the nodes is possible. As the mapping progresses, there are fewer resources available for mapping, which reduces the temperature function. Lower temperature makes the placement of the nodes difficult. Technically, at the end of the simulated annealing cycle, a valid

optimal mapping should be achieved but, the compilation time (time to find a valid mapping) increases drastically with the increase in the number of nodes in the DDG.

A modulo scheduling method was proposed in [51]. The heuristic initially uses modulo scheduling to map the application graph (loop graph) onto the architecture graph of CGRA. The architecture graph supports all the irregular interconnects in the architectures. The heuristic then uses the Simulated Annealing technique to relocate the vertices, aiming to reduce resource overuse. After relocating vertices, the "Dijkstra's Algorithm" (shortest path) [51] is used for rerouting. After relocation and rerouting, the sum of resource overuse is calculated. The algorithm decides to make the changes or revert the nodes and edges to the original state based on the overall cost.

A Routing-Aware technique was presented by [88]. The technique starts with the *CDFG* and uses *List Scheduling* to get the initial schedule by sorting from sink to source. The vertices with the highest priorities are placed first. A graph of unmapped vertices and edges is constructed. It uses Dijkstra's algorithm (shortest path) to map this graph onto the CGRA. To explore the solution space, the technique uses Quantum-inspired Evolutionary Algorithm *(QEA)* [88]. QEA uses *Seed*, which is the initial mapping by list schedule, and *Fitness* function, which is performance or inverse of total latency. The solution is then given as a seed for the next generation.

An integrated kernel mapping and scheduling method was proposed in [52]. The main goal of this method is to hide the delay of operations that has various latency. Firstly, the DFG is expanded by inserting routing nodes to communicate and route the data necessary for the consumer node. The expanded DFG is then mapped onto the CGRA using an A* algorithm [95]. Redundant routing operations are then deleted. Kernels that cannot be fit into the CGRA architecture are partitioned to get smaller sub-kernels that are then mapped onto the CGRA. The sub-kernel creation is done

post-DFG step and a recursive partitioning algorithm is employed. To calculate the cost of the schedule, a Slack Violation Table (SVT) is formed to keep track of the timing violations, and a Modulo Resource Table (MRT) is formed to check resource overuse [52]. A placement cost is calculated by adding MRT and SVT values. A simulated annealing method is used to get a valid mapping. On encountering a failed mapping, a lower performance mapping is tried.

A slack-based simulated annealing approach proposed in [53]. Firstly, the DFG is expanded to incorporate all the routing nodes. Then, the operations are modulo scheduled, and simulated annealing is used to find a mapping. Then, based on the slack table the cost function for the cycle in a Time extended CGRA is taken. If any cycle's cost exceeds 100%, then without increasing the II best mapping is chosen. If a mapping is not available, II is increased and mapping of the nodes is repeated. EGRA [96] is used as the target architecture. The slack calculation for each operation (mem, route, mult, ALUs) and percentage of a 1.37ns clock period based on 90nm technology (1.37ns is the highest delay of `mult`) [53] were used for the mapping. These values were specific to the architecture used and may not be applied to arbitrary CGRA.

Techniques proposed in [54, 55], predominantly used for streaming-based/pipelined spatiotemporal mapping, the CGRA is divided into columns and performs different operations every cycle. For example, for a loop to be executed on CGRA with Load, execute1, execute2, and store operations, Column1 of the CGRA Loads the data of the first iteration in cycle 1, and in cycle 2 column 2 perform execute1, whereas column 2 of loads data for the second iteration. First, a CDFG is generated for a given application using SUIF2 [97] parser. Next, a loop unrolling is performed for each column, the PEs in the column being a constraint. Scheduling and routing of operations are done for one column because consecutive iterations can be performed

in the next columns. A Quantum-inspired evolutionary algorithm (QEA) is used for mapping CDFG onto the CGRA target architecture. Resource constraints of a number of PEs and registers are given as inputs to the QEA algorithm. This approach employs HLS techniques for loop-level parallelism. In addition, this technique relies on the registers, Global and local, for routing data instead of having routing nodes. The compilation of the technique is not mentioned in the paper and to the best of our knowledge, it should be on the higher side compared to incremental methods because of the exploratory nature of QEA. In addition to the previous approach, [54] an optimal mapping step and an overall compiler design flow were added in [55]. A list scheduling step was also added to QEA to make it faster than the traditional QEA and the ILP based mapping. The application C code is split into two parts, (a) the code segment for the processor (in this case RISC) and (b) a code segment for CGRA. The code segment for CGRA is first converted into a CDFG by the compiler. The compiler transforms the CDFG by adding dummy routing nodes for data forwarding. Unwanted routing operations are then the candidates/routing nodes of the same PE can be mapped on to the same PE.

Most of the compiler approaches use the application code or kernel code to produce a graph to be mapped on CGRA. Conversely, [56] uses an already compiled binary code of the application. The application binary code, compiled to run on any microprocessor, is first disassembled, and the kernels are extracted. The kernels are then converted into a control data-flow graph (CDFG) using the SoCDAL tool. This technique extracts the innermost kernel of any nested loop and kernels with static/known trip count. The scheduling and P&R and performed by high-level synthesis methods used in [88]. Post HLS, kernels with unknown trip counts or unsupported instructions and kernels with low-performance gains are discarded by the compiler.

An ILP based mapping algorithm was proposed in [57]. It is a highly flexible framework that uses Gurobi solver [98] to solve the mapping problem. First, the loops to be mapped, are converted into a Modulo Routing Resource Graph (MRRG). Next, there are nine various constraints included for the ILP for the correct functionality of CGRA. The ILP solver then tries to find a mapping, and it is not incremental, meaning if the ILP could not find a valid mapping it exists. Compared to the simulated annealing solver [57] was able to find a valid mapping for more loops.

Graph Based Techniques:

Graph-Based Techniques uses graph theory solutions to find a mapping of the data graph onto the Time-Extended CGRA (TEC). EPIMap, an epimorphic mapping solution, was proposed in [25]. Firstly, the "Out-degree" constraint (i.e., number of sibling nodes from a single parent node should be $\leq$ number of neighboring PEs any given PE is connected) and if this is violated is EPIMap reroutes by inserting a routing node or recomputes the parent operation. Secondly, EPIMap checks if a given DFG is balanced. If an arc exists from nodes $i \rightarrow j$ and the nodes are scheduled at time t_i and t_j respectively, and if $(t_i - t_j) > 1$ then the DFG is not balanced[25]. In this case, an extra routing node is added to balance the DFG. Finally, the EPIMap algorithm checks if the nodes at each height of DFG are less than the total number of PEs is a non-time extended CGRA. If the DFG does not satisfy this condition, the EPIMap finds nodes that can be moved, to the other level. Now a Minimum II (MII) is calculated followed by the creation of Modulo DFG ($MDFG$)[25]. During MDFG construction, all the above constraints should be satisfied, otherwise EPIMap increases II until such a graph is achieved. EPIMap finds a "Maximum Common Subgraph" (MCS) of $MDFG$ and TEC for placement of the nodes. The isomorphic condition check between the two graphs results in the valid mapping [25], and if

isomorphism does not hold, the MDFG must be changed until a valid mapping is achieved. Similarly, a clique-based approach was proposed by [31] in which the problem formulation for finding a mapping of a node onto the Data-Flow Graph is converted to finding a constrained maximal clique [31] in the product graph of DFG and TEC. The weight assigned to a node is the sum of out-edges' weight originating from the node. A mapping can be achieved by finding the maximal clique that has a total weight less than the registers available [31].

[30] technique employs the concept of minor of a graph to find a valid mapping between the data graph and the Time extended CGRA. Firstly, the Minimum II (MII) is obtained, which is a maximum of Resource II and Recurrence II. The algorithm starts with mapping and checks whether the mapping M with SRG (Schedule and Route Graph) is a minor of GII, which is Modulo Resource Routing Graph ($MRRG$) extended up to II cycles. A function is used to check whether the current mapping is a minor of the current GII. The algorithm generates all possible mappings of the DFG with the GII. So if a mapping exists, GraphMinor will algorithm will be able to find it. If the current mapping is not a minor of GII then the II is increased and the mapping steps are repeated. This process is performed until a valid mapping is obtained. Along with [99, 26] this technique uses path-sharing. i.e., sharing of same edges with same sources.

Unlike other graph-based mapping algorithm methods, the configuration context reduction method [58], uses sub-graph isomorphism and sub-graph reconfiguration algorithms, to merge similar configurations or eliminate unnecessary ones. The technique starts with a DFG of the application and converting them into different sub-graphs, called atomic-DFG, with no data dependency by the sub-graph isomorphism. The independent atomic-DFGs are categorized, for example, atomic-DFGs in the same category have the same DFG structure. The sub-graph recombination selects

congruent DFGs (kernels) from different categories and *templates* are extracted. The number of templates means the number of configurations times required for CGRA PEs and the number of congruent kernels means the loop count for the execution. The mapping problem is of the DFGs onto the CGRA can be explained as constructing the kernels by atomic-DFG. The REMUS [45], architecture and compiler framework, was used for experiments.

A MapReduce-based mapping technique was introduced by [59] which uses MapReduce, Geometric Programming problem [100], steps of segmenting (Mapper) and merging (Reducer) to the CGRA DFG. The loop computing is segmented by unrolling, and the problem of finding optimal unroll factor is formulated using a convex optimization problem. The performance is maximized for the memory bandwidth, hardware specifications, and resource constraints. A tree-structured Reducer model is used to merge the Mapper's output and produce the final mapping. The tree structure is used because it has the least critical path. Like the previous method, REMUS [45], architecture and compiler framework, was used for experimentation on six different variants of matrix multiplication.

Analysis and Discussion: The compilation time of spatiotemporal mapping techniques is one of the key factors to be considered. Some incremental methods are still considered to yield lower compilation time compared to the exploratory methods due to their simplicity. The cost metrics are different for different methods, and the complexity of cost metrics calculation can increase the compilation time to a greater extent, especially compilation time for exploratory methods are considered to be on the higher side since these techniques try to find an optimal mapping with sophisticated algorithms. Even though temporal techniques are highly reconfigurable, meaning, the configuration of PE changes during the execution of the loop, resource under-use is prevalent compared to spatial mapping. The advantages of

the spatiotemporal technique over spatial mapping technique are, 1) spatiotemporal mapping has PEs available in the time dimension to map a node, whereas in spatial, the first node alone has all the PEs for selecting during whereas the second node has one less PE and so on, 2) complex dependency edges can be easily routed across the cycles and 3) techniques for lowering the II and algorithms to find the best mapping can yield a near-optimal mapping, but in most cases, the quality of such mapping is inversely proportional to the compilation time of such algorithms.

3.4 Resource Utilization

The compiler methods discussed in this category focus mainly on utilizing the data memory available in the CGRA and register files(RF) available in CGRA to improve the mapping. They also identify the data communication overhead from the main memory and efficiently use the register files in the CGRA architecture to minimize the overhead. There are different types of register file architectures used in CGRAs like local RF (RF available in each PE), shared RF (RF shared between a few PEs), and Global RF (RF shared by all PEs). This category is one of the least explored in CGRA research, mainly the RF techniques, because the compiler is more restricted and depends greatly on the underlying RF architecture.

3.4.1 Data-Memory Aware Techniques

Managing data on CGRA memory, while executing loops is one of the important aspects that need to be considered for two reasons 1) The local memory of CGRA has to store all the data required for a particular cycle of the loop, and by design, there is no hit/miss for these memories and 2) having a large local memory or bigger cache is not preferred due to high complexity, power consumption and data transfer rate. Moreover, it is impractical and costly to have a large number of reading/write

ports. The existing techniques in this area try to explain some of these features and propose methods to handle these problems.

As described in the spatial mapping Section 3.3.1, the work by [47, 48], considers memory operations during clustering. The criteria are that the number of memory operations should be less than or equal to the number of reads/write ports per line in the CGRA architecture. The nodes accessing the same addresses at different iterations, known as *alignable* nodes, are mapped to the same row for efficient data transfer from local memory.

A Data memory bandwidth Aware mapping technique is presented by [60]. This technique primarily checks for Data Reuse Opportunity (DRO) on the DDG to create a Reuse Graph. The compiler then maps the reuse graph on the CGRA architecture using three cost constraints. During mapping, the compiler calculates the memory access delay, routing delay, and interconnect delay. By efficiently performing memory operations in parallel and routing the reused data and by minimizing interconnect delays, [60] achieves better performance. Expanding on the concept of data reuse and memory bandwidth use [61] proposed a technique to efficiently bank the data memory and update the mapping to better utilize the bank distribution. A DFG of a loop with memory operations is shown in Figure 3.4(a), and the CGRA architecture with double-buffered data memory is shown in Figure 3.4(b). As shown in Figure 3.4(c), naive mapping underutilized the bank distribution, whereas [61] utilizes the bank distribution and maps nodes that access the same memory parts to the PEs accessing the same bank.

A memory access optimization technique is presented in [62]. The DFG of the application loop is transformed using load reduction where trailing memory operations are removed, and the leading memory operation provides the data. The authors claim that this can effectively reduce the load operations but, only RAR and RAW

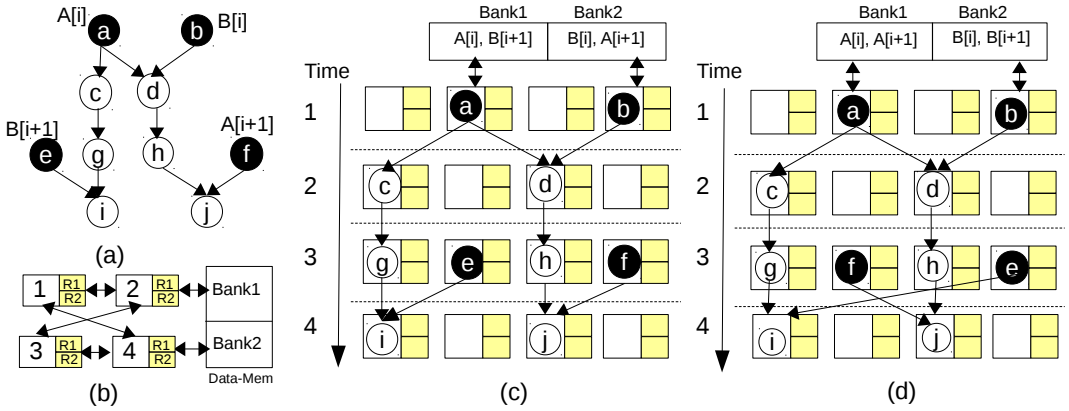


Figure 3.4: (A) DFG of a Loop with Nodes a, b, e, f Memory Nodes (Load, Store Operations) Denoted by Darker Shade. (B) 2×2 CGRA Architecture with Double-bank Local Memory, (C) Mapping by Ems Causing Bank Conflict, (D) Mapping by High Throughput Mapping Technique Resolving the Bank Conflict.

dependencies are considered for load reduction. Reuse edges are introduced in the *DFG* from the leading memory nodes to the trailing memory node and are annotated with reuse distance. Array Clustering is performed next, where the total number of arrays in the DFG is extracted, and priorities are calculated [62]. Then the arrays are mapped in decreasing order of priority with lower cluster cost [62]. Finally, the optimized DFG is modulo scheduled onto CGRA considering the reuse edges in the DFG.

A more intuitive way to solve more than one-cycle forward dependence edges in the DFG is by using routing operation to route the dependence. But the performance of using routing operation can deteriorate when the dependence becomes too large, for example, a dependence with more than a few cycles can add more routing nodes and can increase the II. To solve this problem, MEMMap [63] was proposed, which uses CGRA's local data memory to route the operations. For a dependence, $src \rightarrow dest$ the *src* operation is stored into the memory in the next cycle of execution and retrieve/loaded back from the memory one-cycle before the *dest* operation if

the dependence cycles are more than 3. Otherwise, MEMMap uses routing nodes. Usually, load-store operations can be costly, i.e., it can take more cycles to execute based on the data locality and data availability in CGRA's data memory which is a point worth investigating.

3.4.2 Register-Aware Techniques

Register-files (RF) in CGRA can be utilized to improve the acceleration by routing intermediate values through them. The naive way of communicating the values is by adding routing nodes, i.e., using PEs to route the values. There are only a few compiler techniques that utilize the register files in CGRA, and it is an unexplored area compared to some of the earlier sections covered in this survey. Figure 3.5 shows REGIMap [31] mapping technique using register files in the PEs. The loop to be accelerated is shown in Figure 3.5(a), and the 2×2 CGRA is shown in Figure 3.5(b). The Data-Flow Graph of the loop is shown in Figure 3.5(c). The mapping of the DFG on the CGRA architecture using the REGIMap mapping algorithm is illustrated in Figure 3.5(d). REGIMap uses the register files inside the PEs to communicate the data, improvement, over the naive methods that use PEs to route the values. In the naive case (using PEs to route), the minimum II would be 3, but using REGIMap, the II=2.

EMS [28] technique, explained in Section 3.3.2 allots register files during scheduling. The main use of allotting register files in EMS is to avoid spilling and to enable routing of operations through them.

Register-Aware application mapping algorithm was presented by [31]. The mapping is done in three steps. First, a compatibility graph $P(V_P,E_P)$ between the DFG $D(V_D,E_D)$ and $TEC\ RII(V_R,E_R)$ is constructed. Second, the arcs are given weights (a measure of a number registers to be used). Finally, DFG is mapped onto CGRA by

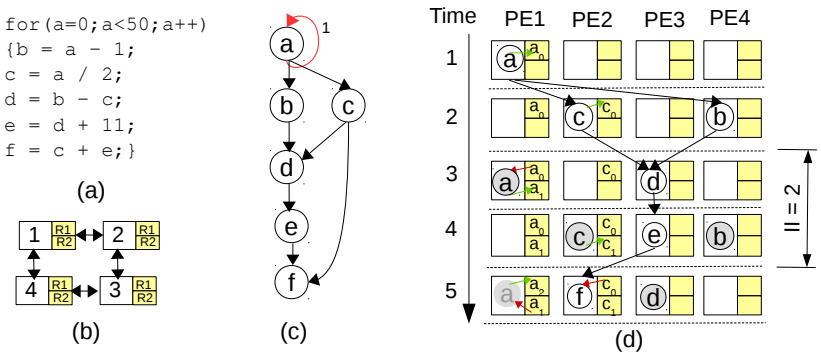


Figure 3.5: (A) Kernel Code (B) 2×2 CGRA Architecture. (C) Corresponding DFG of the Loop (D) Utilization of Registers for Routing With II=2

finding the maximum clique in the compatibility graph. Node weights are calculated, which is the sum of the weight of outward arcs. REGIMap solves the placement of D onto RII by finding the largest clique, whose summation of arc weights is less than the total registers available.

URECA, [36], is a recently proposed compiler technique, which advocated the use of unified register file architecture, which was previously proposed, in [101]. Ureca manages both the recurring variables (variables repeatedly read and rewritten in the loop iteration) and the non-recurring variables (read-only) in the same local unified RF that has both rotating RF(to manage recurring variables) and non-rotating RF (to manage read-only variables). The main contribution of URECA is the compiler that allocates only required registers and divides the RF into a rotating part and a non-rotating part using a value c (called "configuration boundary"). The read-only variables are later loaded into the non-rotating RF, ensuring the correct execution of the application. URECA is a versatile compiler solution and can be used with any mapping technique.

Analysis and Discussion: Register file utilization is an intelligent way to extract good performance in spatiotemporal mapping algorithms. To the best of our

knowledge, there are no spatial mapping algorithms that utilize the register files available in the PEs for routing. A design space exploration is necessary to find the size, connectivity, and ports required for the RF for better performance. The work presented in [90] tries to answer these questions by experimental methods. The RF routing techniques like REGIMap, map the DFGs with a constraint that the source and destination nodes should be scheduled more than one cycle apart, and the nodes should be mapped to the same PE. Mapping the source and destination nodes to the same PE, allows the value to be written into the RF by the source node and read by the destination node at different cycles. Due to this restriction, only a few dependencies are routed via RF, and other dependencies are routed via PEs.

The authors present that the 1) Global RF should be 12-16 registers 2) Global RF should have maximum ports and be connected to as many PEs as possible and 3) for Local RF, one read port, and one write port are sufficient for most of the application but may vary with the application domains. A recent technique, Resource-Aware Mapping (RAMP), was proposed in [32] to better utilize the CGRA resources, to improve the mapping quality. The heuristic proposed systematically analyzes the data dependence graph of the kernel application for better routing of the node of the graph and changes the graph before the mapping stage. Having a heuristic for resource utilization before the mapping stage not only guarantees a mapping but gives a better quality mapping, compared to the state-of-the-art techniques. The routing options explored by the compiler are i) Routing via PEs, ii) Routing via distributed register files, iii) Routing via Memory, iv) using memory to load read-only operations, and v) Re-computation [32]. At any given II, RAMP checks for these routing options and increases the II only if none of the options are viable.

3.5 Generalization

The compiler techniques previously discussed can accelerate simple loops in an application or the innermost loop of the nest. This section analyzes the compiler techniques that can map the most frequently used loop constructs like nested loops and loops with if-then-else (conditional) structures.

3.5.1 Nested Loops

[65] suggests that accelerating the innermost loop of the nest has a major drawback because of the communication overhead imposed between the host processor and the accelerator. Executing the outer loop+inner loop in the CGRA has a memory constraint, as all the variables of the loop should be transferred to the CGRA memory. An increase in the depth of the nested loop also poses many disadvantages.

[65] tries to solve the nested loop problem by both hardware and software methods. The authors provide an algorithm for sequential code execution on CGRA and hardware modification like outer loop trip and sequential code execution counters and some extra states for sequencing the outer loop. There are three architectures proposed for executing the nested loops 1) Baseline - Inner loop pipelining + SW outer loop (traditional) 2) Intermediate - Inner loop pipelining + HW Outerloop (by the above hardware modifications) 3) Nested Loop Pipelining - Intermediate + outer loop pipelining. The compiler forms "Epilog-Independent Configuration" (EIC) and "Prolog-Independent Configuration" (PIC) regions [65]. By forming these independent regions, multiple epilog-prolog pairs are obtained for seamless execution by moving EIC and PIC configurations suitably. This rescheduling technique has a few shortcomings like 1) there should be no data dependencies between EIC and PIC to merge epilog-prolog, and 2) this method can be applied for two-level nested loops

only.

Flattening of loop kernels are presented in [66, 67] in which nested loops are flattened to make a single loop and is executed in the CGRA. Loop flattening increases the computation inside the loops. To reduce the overhead due to flattening, special operations such as 1) nested iterators 2) extended accumulator 3) periodic store are proposed. These special operators were proposed for design space exploration of CGRA to improve data path reuse and utilization. Imperfectly nested loops can be handled by guarding the statements in the outer loop by predication. If there are sibling inner loops, predication-based guarding of outer loop statements is not practical. Another way to handle imperfect nested loops is by loop fission, where the outer loop statements are extracted into a separate loop when possible. This method can also be applied to independent sibling loops. Loop fission is not possible when there are dependencies between the inner loop and the outer loop. Execution resorts to guarding outer loop statements by predication when such a transformation is not possible.

Polyhedral model-based mapping technique was proposed in [68] to map two innermost loops of a perfect nested loop structure. The iteration bounds of the two innermost loops in a rectangular space are affine transformed into an iteration space defined by a parallelogram. The new iteration domain in form of a parallelogram will be tiled, and the resulting tiles will be executed on the processing elements of the CGRA arranged in a 2-D array. The polyhedral model is restricted to 2-deep loops, and the loops should be perfectly nested [68]. Imperfect loops are converted into a perfectly nested loop, and then polyhedral technique [68] is employed. Improving on [68], [71] proposed an affine transformation-based approach to mapping perfectly nested loops (max depth of 2) and reducing communication and reconfiguration cost. This was done by incorporating communication minimal transformation [102] to

reduce communication costs, and in a later stage, the compiler finds the trade-off between reconfiguration and communication costs.

For mapping multi-level nested loops, a spatiotemporal mapping approach using SPKM [44] is used in PolyMap, [103]. Firstly, the two-innermost loops of arbitrarily nested loops are spatially mapped on the kernel by SPKM. The outer nests are then mapped by time extension. Parallelism in the spatial mapping is by using loop tiling methodology used in [68]. PolyMap also uses a Genetic Algorithm for search space optimization for mapping the loop.

DNestMap [72], maps loops that are deeply nested. Employing a spatiotemporal computing format, a beneficial code segment called (Mapping Units, MU) [72] is selected from the Control Data Flow Graph (CDFG). The CGRA is spatially partitioned to execute different nests of the loops. Subsequently, this partition is temporally time extended, and the MU is mapped. Even though this may increase the II of an MU, the authors claim that considering context switching overhead in conventional CGRA (accelerating only the innermost loop), this technique provides acceleration at lower energy.

Analysis and Discussion: The approaches discussed above are valid only for two-level deep nested loops, and the iteration of the loops should be known a priori (loops with unknown trip counts cannot be accelerated). Irregular memory accesses of the loop nests can degrade the performance on CGRA. For example, in a technique that accelerates only one innermost loop, when the outer loop runs for more iterations and the inner loop runs for just for few iterations, the memory transfer is huge, and performance degradation is imminent. As mentioned earlier, memory bandwidth, sequential codes in the outer loops, and inter-loop data dependencies are some of the unexplored issues. DNestMap [72] tries to tackle this problem by having a spatiotemporal execution model but, the CGRA resources may be limited to the

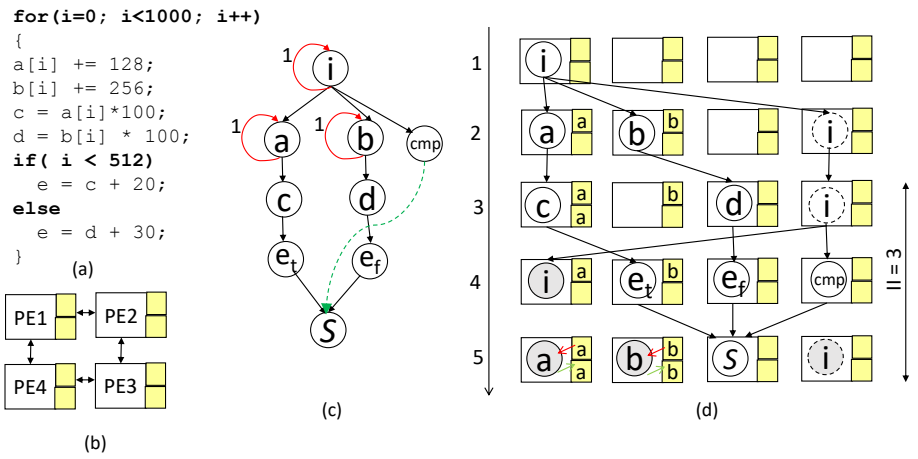


Figure 3.6: (A) a Simple Loop with If-then-else Conditional (B) a 2×2 CGRA Target Architecture with 2 Registers in Each PE. (C) Partial Predication Adds Three Operation for e Inside If-then-else Statement, e_t for If-path, e_f for Else-path and S, a Select Operation to Select Between If and Else Path Based on the `Cmp` Result (D) a Valid Mapping Obtain With II=3

depth of the nests in the application, meaning this technique is not general enough to accelerate arbitrarily nested loops.

3.5.2 Branch Aware – Loops with Conditionals

Along with nested loop structures, loops containing if-then-else are common in applications. The underlying idea of the branches is that there is only one path (true path or false path) that is executed in an iteration based on a condition. It is very difficult to know the condition at compile-time, and it cannot be known until the condition is resolved. Existing approaches in this section try to maps the loops of the branches onto the CGRA architecture. A simple if-then-else code and a partial prediction scheme DFG are shown in Figures 3.6(a)-(b). The S node is a select node that selects either if-path or else-path values based on the branch outcome. The DFG mapping is shown in Figure 3.6(c) with an $II = 3$ using Register-Aware mapping, REGIMap. The three most commonly used techniques to map loops with conditional statements are explained below.

Partial predication executes operations from both paths [77, 43] of conditional branches and selects the correct-path [4] result at runtime based on the branch condition value (predicate value) [75]. The S node represents the *select* operation that selects between the final result of the if-path or else-path. [76] proposed an ISA modification, a conditional move statement *cmov*, which was added to incorporate partial predication technique into their compiler. In full predication, the operations of both the path updating the same variable are mapped onto the same PE but at different times. Unlike partial predication, the correct path is executed based on the select signal (predicate). The full prediction scheme was discussed in [73, 74, 43].

The dual-issue scheme for CGRA was proposed in [76], in which two instructions are issued simultaneously to the same PE, one from if-path and one from else-path. These two instructions are packed into a single node during DFG formation, and context creation is called packed nodes. One of the operations in the packed nodes is selected in the runtime based on the predicate value. In the case of an unbalanced path [5] , nops are added to balance the paths. The Branch Aware mapping [43] developed compiler techniques for the dual-issue scheme.

Improving on the Dual-issue and partial predication methods, a"The Path Selection Based Acceleration of Conditionals in CGRAs" (PSB) proposed in [77]. Based on the branch outcome, only the correct path instruction is issued by the Instruction Fetch Unit (IFU) to the PEs. The various stages of PSB are shown in Figure 3.7. Figure 3.7(a) shows the loop containing a simple if-then-else to be accelerated on a 2×2 CGRA architecture as shown in Figure 3.7(b). PSB compiler $fuses$ the both the conditional path operations in to $fused$ nodes, denoted by e as shown in Figure 3.7(c) [77]. Based on the *cmp* result, which is communicated to the Instruction Fetch Unit

[4]The if-path or the else-path instruction selected at runtime will be referred to as correct-path.

[5]Unbalanced paths are conditional statements where the instruction inside if- and else-paths are not equal.

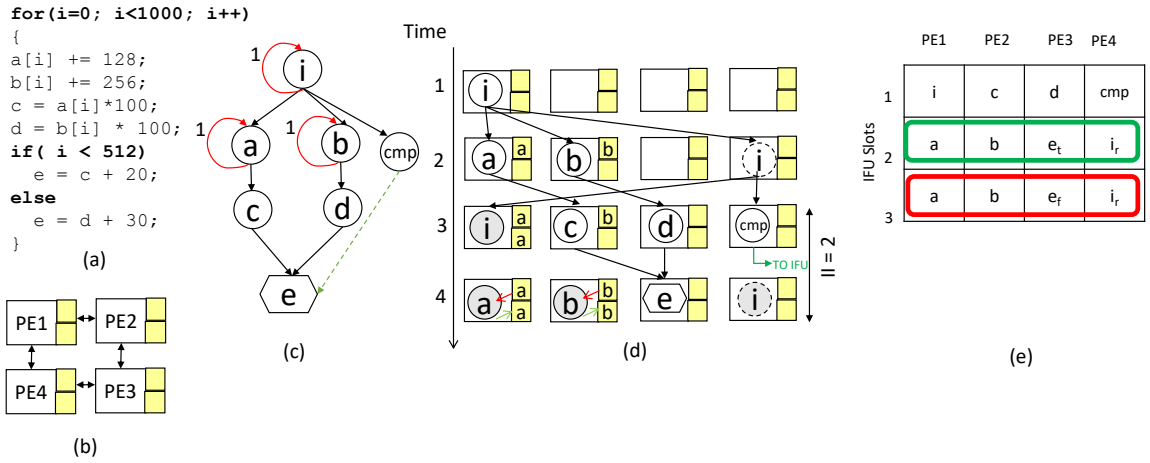


Figure 3.7: (A) a Simple Loop with If-then-else Conditional (B) a 2×2 CGRA Target Architecture with 2 Registers in Each PE. (C) Psb Fuses the If-path, Else-path and Select Operation from Partial Predication to Form a Single *e* Operation. (D) a Valid Mapping Obtain with II=2, Where `Cmp` Outcome Is Communicated to the Ifu. (E) to Facilitate the Issue of Only the Correct Path, the Instruction Is Laid out in the Instruction Memory. If the `Cmp` Is True Ifu Slot 2 Instructions Are Issued and Executed Whereas If `Cmp` Is False Ifu Slot 2 Is Skipped and Ifu Slot 3 Is Issued and Executed.

(IFU), at runtime, only one correct operation is issued by the IFU. So either *c* or *d* is executed at runtime. Figure 3.7(d) shows the mapping of the DFG onto the CGRA architecture using the REGIMap mapping algorithm. This execution is made possible by arranging the instructions conveniently in the instruction memory. PSB requires balanced conditional paths for correct execution meaning, instructions inside the true-path and false-path should be equal, and if not, nops are added to make them balanced. Figure 3.7(e) shows the instruction memory layout, where slot 2 is the if-path, and slot 3 is the false-path. Since there is only one instruction in the conditions for the example, if the *cmp* is true, slot 2 instructions are issued, and slot 3 is skipped, and if the *cmp* is false, slot 2 is skipped and slot 3 is issued [77].

All the above methods add nops in case of *unbalanced* path of the conditionals [78]. The evaluator-executor method was proposed in [78] to overcome the unbalanced paths. In this compiler technique, the loops are first converted into multiple smaller

loops, in general, two loops. The first loop is called the evaluator, which executes the branch condition, and the second loop is called the executor, which executes the branch. Now a Program Dependence Graph *PDG* is formed. This PDG is scheduled and mapped using EMS [28] without using rotating registers.

A Trigger-based scheme was introduced by [79] which is a co-design approach for accelerating nested if-then-else. The software technique converts the CDFG of the loop into a "Triggered Data Flow Graph (TDFG)" [79] with triggers as nodes with Single Static Assignment (SSA) transformation. Data dependencies are transformed to predicate dependencies using a heuristic algorithm proposed in this paper. Additional Trigger resolution hardware is added for every PE to translate the trigger signal, and a priority encoder [79] is added to select the appropriate trigger for the instructions. The trigger instructions are programmer-coded, and the trigger resolution hardware chooses the instructions based on the predicate updates from the PE. It is shown in [79] that this technique achieves the best performance (II) compared to BRMap [43], partial predication, and SFP algorithms but has modest PE utilization compared to partial predication and BRMap. The scalability of the approach is not discussed.

To accelerate loops with arbitrarily nested conditionals, partial predication imposes high overhead in terms of increasing the number of nodes in DDG for each operation in an if-path or an else-path, as shown in Figure3.6(c).

Analysis and Discussion: The existing co-design approaches in this section try to improve the performance but have hardware and software overheads. There is a trade-off between the hardware and software overheads for the co-design approaches. The compiler requires more time analyzing the code for merging candidates, which increases the compilation time. Executing the fused operations requires hardware support. Partial Predication is better for accelerating smaller DFGs on larger CGRAs,

where there are enough resources for executing both paths. For larger DFGs partial predication increases the number of nodes drastically, and approaches like LASER can be effective. Reducing the II by co-design approaches is very effective in terms of energy reduction. Even though there is a slight power overhead due to additional hardware components for supporting branches, there is a reduction in the number of cycles the loops execute by CGRA (lower II).

3.6 Multi-threading

In this section, we have discussed techniques proposed for accelerating multi-threaded applications onto CGRA. For example, if a compiler finds a valid mapping for thread $T1$ which takes p PEs, and if another thread $T2$ requires q PEs for mapping and if $p + q \leq total\_number\_of\_PEs$, the compiler can map both the threads onto the CGRA for simultaneous execution and can maximize the performance [80].

Enabling Multi-threading on CGRA was proposed in [80]. This method starts by dividing the CGRA PEs into *pages* containing a fixed number of PEs. First, the DFG of an application, with no page restriction, is mapped with better utilization of PEs on CGRA. When a new thread is invoked, the existing kernel is transformed to fit into lesser pages and allocate the remaining free pages to the new thread. After the completion of the new thread, if the old thread is still executing, it is expanded to its original form to better utilize PEs again. During the dynamic page transformation, the inter-page and inter-node dependencies are maintained [80]. The other constraints addressed in this work are better utilization of register resources available in each PE and data flow constraint to meet the data dependency between the nodes and the pages while dynamically transforming the schedule.

The 'Single-Graph Multiple Flows' technique was introduced by [81]. This work converts the multi-threaded application into a CDFG and balances the if and else

paths. Then these kernels are then mapped onto the CGRA like architecture called Coarse-Grained Reconfigurable Fabric (CGRF) that is modified to pipeline the instructions of various threads of the kernels and introducing dynamic scheduling of the instructions to achieve Simultaneous Multi-Threading (SMT). Apart from the compiler improvements, the hardware of the CGRF is also improved to enable the correct execution of multi-threaded kernels. For the correctness of the computed results and to prevent deadlock, the technique uses throttling and parallelism-based thread epoch (TIDs) [81]. To hide the latency of the non-pipelined instructions, dedicated additional function units (compute units) [81] are used for pipelining the non-pipelined instructions. It has also been shown that the modified CGRF architecture consumes lesser energy compared to the general-purpose GPUs [81]. A later technique by introduced [82] improved upon [81] for better inter-thread communication of intermediate values during execution of kernel on CGRF.

Chapter 4

CRIMSON: A RANDOMIZED ITERATIVE MODULO SCHEDULING APPROACH

4.1 Background

The CGRA mapping problem, being NP-complete, is performed in a two-step process namely, scheduling and mapping. The scheduling algorithm allocates timeslots to the nodes of the DFG, and the mapping algorithm maps the scheduled nodes onto the PEs of the CGRA. On a mapping failure, the II is increased and a new schedule is obtained for the increased II. Most previous mapping techniques use the Iterative Modulo Scheduling algorithm (IMS) to find a schedule for a given II. Since IMS generates a resource-constrained ASAP (as-soon-as-possible) scheduling, even with increased II, it tends to generate a similar schedule that is not mappable. Therefore, IMS does not explore the schedule space effectively. To address these issues, in this chapter we propose CRIMSON, Compute-intensive loop acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping technique that generates random modulo schedules by exploring the schedule space, thereby creating different modulo schedules at a given and increased II.

4.2 Motivating Example

Let us consider the DFG of loop to be mapped on a 2×2 CGRA, shown in Figure 4.1(a) and (b), respectively. Previous state-of-the-art techniques like RAMP, get a schedule from IMS [34] before mapping the nodes. IMS starts by computing the resource constrained minimum II ($ResMII$) and recurrence constrained minimum II

($RecMII$) from the DFG and the architecture description. For the given example in Figure 4.1, total nodes = 9 and total resources available = 4. The minimum II (MII) is the maximum of $RecMII$ and $ResMII$. Therefore for the above example, $MII = ResMII = \lceil 9/4 \rceil = 3$. After computing the MII, IMS sets the priorities for each node. Priority is a number assigned to each node, which is utilized during scheduling. Based on the height of the node, from the given DFG, the deepest node is given the least priority using depth-first search strategy. For the loop DFG given in Figure 4.1(a), node e gets priority 0, nodes d and i get priority 1, nodes b, c, g, h get priority 2 and finally a and f get priority 3. The nodes with higher priority number are scheduled first with earliest start time. The modulo scheduling starts with II=MII for scheduling the nodes. The CGRA is time-extended, II times and a modulo resource table (MRT) is maintained to check for resource overuse for each timeslot. While trying to schedule each node, resource conflicts are checked. If there is a resource conflict a higher schedule time is tried. For the example DFG, the II=MII=3. Nodes a and f are scheduled at modulo time 0 (0%3). Nodes b, c, g, and h are scheduled at modulo time 1 (1%3) without any resource constraint because there are 4 resources(PEs) at each modulo time. Nodes d and i are scheduled at modulo time 2 (2%3). Finally, e is scheduled at modulo time 0 (3%3). The IMS schedule of nodes (shown in column 1 Figure 4.1(c)) at $II = 3$ is shown in Figure 4.1(c) column 2.

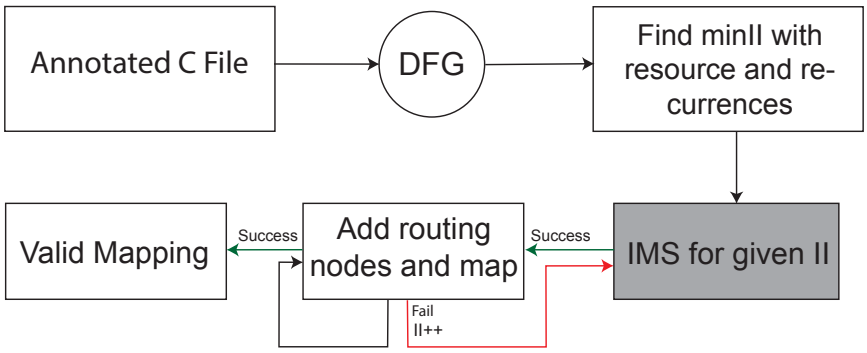


Figure 4.2: Overview of Scheduling and Mapping Workflow of Previous Techniques.

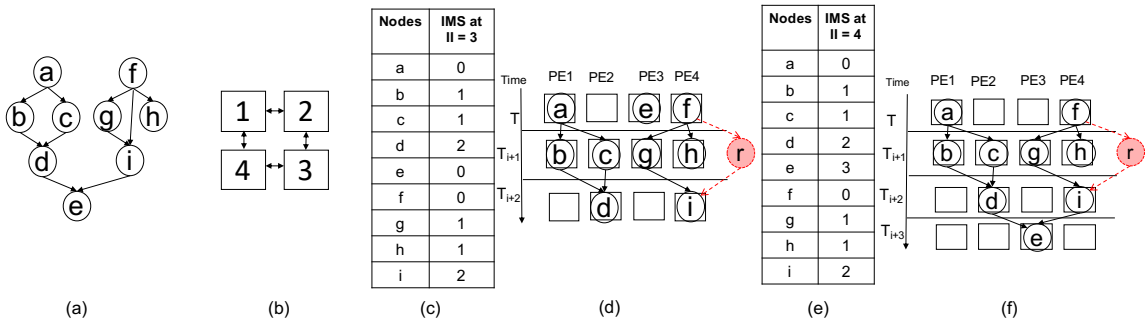


Figure 4.1: (A) DFG of an Application Loop. (B) a 2x2 CGRA Target Architecture. (C) Column 1 Shows the Nodes in the DFG and Column 2 Shows an IMS Schedule for the Nodes at II=MII=3. (D) the Mapping Algorithm Tries to Map the Nodes Scheduled, but Fails Due to Additional Routing Nodes "r" Required to Route Nodes f and i. Failure to Find a Valid Mapping, the II Is Increased to 4 and IMS Is Called Again to Schedule the Nodes Based on the Workflow given in Figure 4.2. (E) IMS Schedule for an Increased II (II=4). (F) Even at an Increased II, the Mapping Algorithm Cannot Find a Valid Mapping Due to Resource Constraint at t_{I+1} Which Is Not Resolved at II=4 and Will Not Be Resolved on Any Further Increase in II.

With this prescribed schedule, mapping algorithms start to map the nodes, but eventually find that a routing node needs to be added to route operation f and i. Due to the unavailability of PEs in that timeslot a routing node cannot be added, as shown in Figure 4.1(d). At this juncture, the mapping algorithm increases the II in an effort to find a schedule that is mappable. On increasing the II from 3 to 4, the IMS algorithm is invoked again to get a schedule. Since the priority calculation of IMS is DFG-based, all the nodes get the same priority. Now, IMS algorithm starts

to schedule nodes based on the priorities for each node. Nodes a and f are scheduled at modulo time 0 (0%4). Nodes b, c, g, and h are scheduled at modulo time 1 (1%4). Nodes d and i are scheduled at modulo time 2 (2%4) and e is scheduled at modulo time 3 (3%4). The IMS schedule for II=4 is shown in Figure 4.1(e) column 2. Again, on failure to map, the mapping algorithm increases the II to 5. IMS repeats the process of assigning priorities to the nodes and as seen in II=4, the priorities do not change. Nodes a and f are scheduled at modulo time 0 (0%5). Nodes b, c, g, and h are schedule at modulo time 1 (1%5). Nodes d and i are scheduled at modulo time 2 (2%5) and finally e is scheduled at modulo time 3 (3%5). On comparing the schedules obtained for II=3, II=4, and II=5, it can be seen that only node e has a different schedule time (from II=3 to II=4) and rest of the nodes have the same schedule. Hence, with IMS, it can be seen that an increase in the II does not correspond to a change in modulo schedule time of the nodes.

The algorithm keeps trying to find a valid mapping at higher II even when there is a mapping failure at a given modulo schedule. This process keeps on repeating endlessly. In the workflow of the previous techniques, as shown in Figure 4.2, after finding the MinII and obtaining an IMS schedule, the mapping of the nodes begin assuming that the schedule is mappable. There are no mechanism to statically and systematically find the feasibility of the obtained schedule, which results in an infinite loop between the scheduling and the mapping stages.

4.3 Algorithm

To alleviate the challenges posed by IMS and the previous mapping algorithms, CRIMSON randomizes the schedule time of each node of the DFG by choosing a time between RC_ASAP and RC_ALAP. Additionally, CRIMSON proposes a change to the previous mapping algorithm workflow Figure 4.2 by performing a feasibility

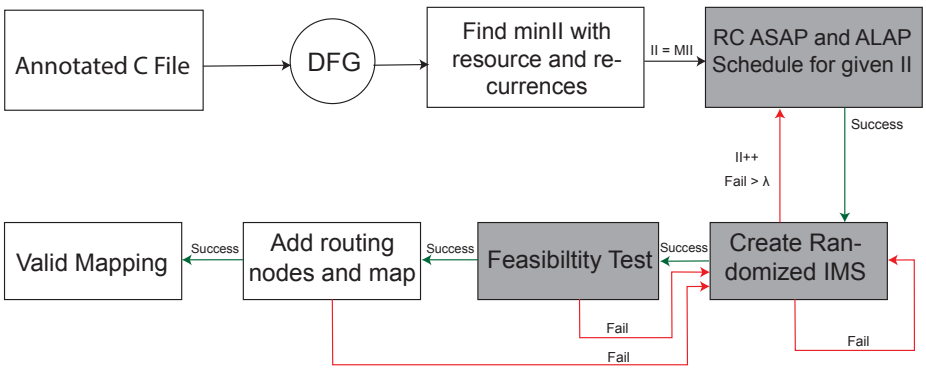


Figure 4.3: An Overview of CRIMSON Workflow, with Addition of Rc_asap and Rc_alap Computation, Randomized Scheduling Algorithm, and a Feasibility Test (Shaded Blocks in the Image Are the Proposed Methods).

test before the actual mapping.

Figure 4.3 shows the modification to the traditional IMS-based workflow shown in Figure 4.2. CRIMSON modifies the IMS-based mapping workflow by adding RC_ASAP and RC_ALAP computation steps before finding a random schedule. The "Create Randomized Schedule" block uses Algorithm 1 and Algorithm 2 to find a random modulo schedule time. On a failure to find a schedule, "Create Randomized IMS" block is invoked λ times before increasing the II. When a random modulo schedule is obtained, the feasibility test statically analyzes if the obtained random schedule honors the resource constraints when routing nodes are added. If a schedule is found to be infeasible due to possible resource overuse, a different modulo schedule is tried for the same II. If the random schedule obtained is valid and feasible, then the mapping algorithm is called to add routing nodes and map the scheduled DFG onto the CGRA architecture.

4.3.1 Computing Resource-Constrained ASAP and Resource-Constrained ALAP

Algorithm 1 shows the CRIMSON's randomized iterative modulo scheduling algorithm. Lines 1-2 finds the RC_ASAP from the Strongly Connected Components

(SCCs) [1] of the DFG. The RC_ASAP is computed in Line 3 of Algorithm 1 as a top-down, depth-first search approach, from the nodes that do not have any incoming edges in the current iteration. After computation of RC_ASAP, RC_ALAP is computed, starting from the nodes that do not have any outgoing edges in the current iteration and in a bottom-up (reverse), depth-first search manner, in Line 4 of Algorithm 1.

4.3.2 Randomized Scheduling Algorithm

After computing RC_ASAP and RC_ALAP, Algorithm 1 Line 5 populates the unscheduled array whereas line 6 sets a boolean Scheduled operation to false for all the nodes, which is used in Algorithm 2. For all the unscheduled sorted nodes in the array, a random modulo timeslot is picked by honoring the resource constraints maintained by MRT, in Line 10 of the Algorithm 1.

[1] Getting the list of SCCs ensures that the nodes in recurrence-cycles are scheduled first using $Sort\_SCC()$ function in Line 5.

Algorithm 1: *Rand_Iterative_Mod_Schedule* (Input DFG D, CGRA CA, Input II)

```
1: D' ← D
2: SCCs ← Find_List_of_Sccs(D')
3: Find_RC_ASAP(II, Sccs, CA)
4: Find_RC_ALAP(II, Sccs, CA)
5: unscheduled ← Sort_Sccs(Sccs)
6: Set_Scheduled_op_false(unscheduled)
7: iter ← 0
8: while unscheduled_size > 0 & iter < threshold do
9:    operation ← unscheduled[0]
10:   TimeSlot ← Find_Random_ModuloTime(operation, CA)
11:   if (schedule(nodes, TimeSlot)) then
12:      scheduled ← nodes
13:   else
14:      return failure
15:   end if
16:   unscheduled ← Subtract(unscheduled, scheduled)
17:   iter++
18: end while
19: if iter == threshold & unscheduled_size > 0 then
20:   return failure
21: end if
22: return success
```

The *schedule*() function in Line 11 of the Algorithm 1, schedules the node at chosen random timeslot . This *schedule* function sets the schedule time of the current operation and consecutively displaces the nodes that have resource conflicts. Previously scheduled nodes having a dependence conflicts with the current operation are also displaced after updating the RC_ASAP and RC_ALAP based on the current schedule operation. The displaced nodes are added to queue of unscheduled nodes. Similar to the *BudgetRatio* in IMS [34], the *iter* is a high value. On a failure to find

a schedule, either due to unscheduled nodes lines 13-14 or if the *iter* value is greater than a threshold (lines 17-18), the Algorithm 1 is invoked again. This is repeated λ times before increasing the II, in an attempt to find a valid schedule. This λ value is not reset for a particular II and used to control the failure due to unmappable schedule or a failure in the mapping step.

Algorithm 2: *Find_Random_ModuloTime* (Operation *op*, CGRA CA)

```
1:  op_ASAP ← get_RC_ASAP(op)
2:  op_ALAP ← get_RC_ALAP(op)
3:  sched_slot ← ∅
4:  timeslots ← get_all_timeslots(op_ASAP, op_ALAP)
5:  Randomize(timeslots)
6:  while sched_slot == ∅ & timeslots_size > 0 do
7:     currTime ← timeslots[0]
8:     if ResourceConflict(op, currTime, CA) then
9:        timeslots ← Subtract(currTime, timeslots)
10:       continue
11:    else
12:       sched_time ← currTime
13:    end if
14: end while
15: if sched_slot == ∅ then
16:    if !Scheduled[op] || op_ASAP > Prev_Sched_Time[op] then
17:       sched_slot ← op_ASAP
18:    else
19:       sched_slot ← Prev_Sched_Time[op] + 1
20:    end if
21: end if
```

Algorithm 2 is called by CRIMSON's randomized iterative modulo schedule (*Rand_Iterative_Mod_Schedule*) Algorithm 1 line 10, to find a random timeslot between RC_ASAP and RC_ALAP. The RC_ASAP and RC_ALAP for a given operation

is retrieved in lines 1-2 of Algorithm 2. Then, an array of timeslots is constructed using the *op_ASAP* and *op_ALAP*, line 4 of Algorithm 2. The array holds all the timeslots from *op_ASAP* with an increasing value of 1 until *op_ALAP*. If *op_ASAP* is equal to *op_ALAP* then the array size is one with either ASAP or the ALAP time. Each timeslot from the randomized array is checked for the resource constraint using MRT. The first valid timeslot is returned as the modulo schedule time for the operation. Due to the resource conflict if a valid timeslot is not present, there are two things to handle, (a) a timeslot for the operation should be chosen and (b) an already scheduled operation from that timeslot should be displaced. Concern (a) in handled in lines 13-17 of Algorithm 2 where if the nodes has not been scheduled previously, *op_ASAP* is chosen as the schedule, else the previous schedule time of the operation is found and the modulo schedule time is computed using line 17. Concern (b) is addressed in the *schedule*() function in Algorithm 1 line 11, explained earlier. The methods addressing these concerns are similar to IMS implementation.

4.3.3 Novel Feasibility Test

Given a valid schedule, it may not be possible to map it because of two main reasons: i) limited connectivity among the PE nodes, and ii) the need to map the extra routing nodes that will be created as a result of scheduling. In a valid schedule dependent operations may be scheduled in non-contiguous timeslots. When this is the case, the operands need to be routed from the PE on which the source operand is mapped, to the PE on which the destination operation is mapped. The operands can be routed using a string of consecutive CGRA interconnections and PEs. These PEs are referred to as routing PEs, and the operation that is mapped on these PEs (just forward the operand from input to output) is called a routing operation. Because of the addition of these routing nodes, the generated schedule may not be mappable.

Previous techniques assume that the schedule is mappable and spend a lot of time searching for a mapping when none is available. In order to avoid wasting time in exploring unmappable schedules, CRIMSON adds a conservative feasibility test to prune schedules that can be proven to be unmappable.

The feasibility test examines the random schedule produced, and for each routing resource that will be added in the future, it estimates the resource usage, considering path-sharing [30]. The feasibility test checks if the total number of unique nodes including the routing nodes per timeslot is less than or equal to the number of PEs in that timeslot. $schedule\_nodes_i + routing\_nodes_i <= PEs_i$, where i is the modulo timeslot. This feasibility check is performed for all the II timeslots. The mapping algorithm is invoked only for schedules that are feasible, unlike the previous approaches such as RAMP [32], where the mapping algorithm is invoked even for infeasible schedules. Since the time complexity of such mapping algorithms is high (time complexity of RAMP is $\mathcal{O}(N^8)$, where $N = n*m$, and 'n' is the total nodes in the loop DFG, and 'm' is the size of the CGRA), invoking them for infeasible schedules is counter productive. The feasibility test reduces the overhead incurred by the mapping algorithm by pruning the infeasible schedules.

4.3.4 Determining the λ value

With every failure in the feasibility test a new schedule is obtained for a given II. The number of times a schedule is obtained for a given II is controlled by the λ value. The scheduling space that can be explored for a given II is calculated by the product of the total nodes in the DFG, the size of the CGRA, and the II, given in Equation 4.1. A brute force exploration of the schedule space is time consuming. Lower λ values may increase the II prematurely, by superficial exploration of schedule space, whereas higher λ values increase the compilation time, due to elaborate exploration of the

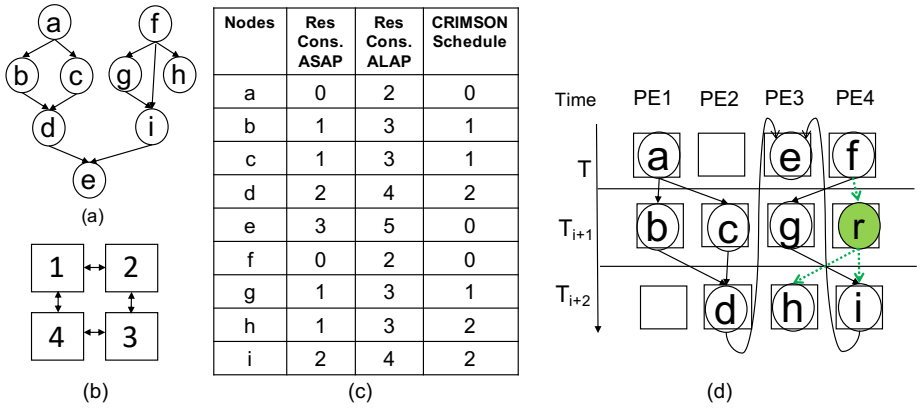


Figure 4.4: (A) the DFG of the Motivation Example. (B) a 2x2 CGRA Architecture. (C) for Each Node of the DFG, Resource Constrained Asap (Column 2) and Resource Constrained Alap (Column 3) Is First Calculated. Then a Random Schedule Time Between Rc_asap and Rc_alap Is Chosen for Each Node. A Valid Randomized Modulo Schedule Is Shown in Column 4. (D) with CRIMSON Schedule a Valid Mapping Is Achieved by the Mapping Algorithm At II=3.

schedule space. Due to the randomness in the scheduling algorithm, a feasible schedule may be obtained faster by chance even for a higher λ value. The λ value is computed using,

$$\lambda = exploration\_factor \times n \times m \times II \tag{4.1}$$

where, 'n' is the total number of nodes in the loop DFG, 'm' is the size of the CGRA and, *exploration_factor* is the percentage of the schedule space that is to be explored. The *exploration_factor* is a user defined parameter. II is also one of the parameters that determines the λ value in Equation 4.1, which means that a new λ is computed for each II. When the II is increased, the scheduling space is also increased therefore the scope of exploration gets broadened. A detailed discussion on the effects of exploration_factor on the scheduling time and II is given in Section 4.4.3.

4.3.5 Running Example

Figure 4.4 shows the working of CRIMSON's randomized iterative modulo schedule algorithm for the DFG and CGRA architecture shown in Figure 4.4(a)-(b) [2] . Instead of assigning a priority based on height like IMS, each node in DFG is assigned two times namely, Resource Constrained As Soon As Possible (RC_ASAP) and Resource Constrained As Late As Possible (RC_ALAP), which constitutes a good lower and upper bound for scheduling [28]. Similar to IMS, CRIMSON maintains an MRT to check for resource overuse during RC_ASAP and RC_ALAP assignment. The RC_ASAP is calculated from the nodes that does not have any incoming edges in the current iteration. These nodes are allotted RC_ASAP time as 0, which means, that the earliest start time of these nodes is at time 0. Based on the outgoing nodes from these start nodes and the delay of each operation, the RC_ASAP of consecutive nodes are computed in a depth-first manner (similar to IMS priority calculation). For the DFG in analysis, nodes a and f are assigned the RC_ASAP time as 0. Nodes b, c, g, and h are assigned RC_ASAP time as 1. Nodes d and i are assigned RC_ASAP time 2 and node e is assigned RC_ASAP time 3. The RC_ASAP times of each node is shown in Fig4.4(c) column 2. Next, starting from the last nodes of the DFG, i.e., nodes without any outgoing nodes in the current iteration, the nodes are assigned RC_ALAP in a reverse depth-first search manner, using $RC\_ALAP = RC\_ASAP + II - 1$. This ensures that $RC\_ALAP >= RC\_ASAP$. For the given DFG, e is assigned RC_ALAP time 5, node h is assigned 3. Nodes d and i are assigned RC_ALAP time 4. Nodes b, c and g are assigned RC_ALAP time 3. Finally a and f are assigned RC_ALAP time 2. The RC_ALAP times of each node is shown in Fig4.4(c) column 3.

After computing the RC_ASAP and RC_ALAP, CRIMSON chooses a random

[2]The DFG and the architecture is the same as the motivation example Figure4.1(a)-(b)

time between RC_ASAP and RC_ALAP, to schedule the nodes. Like IMS, CRIMSON maintains a Modulo Resource Table (MRT) to check for resource overuse in each II modulo timeslot. After checking for resource constraints the modulo schedule time is chosen for each node. This randomization of modulo schedule time creates flexibility of movement for the nodes, which explores different modulo schedule spaces, thereby increasing the chances of finding a valid mapping by the mapping algorithm. A randomized modulo schedule for the example DFG is shown in Fig4.4(c) column 4, and a valid mapping for the scheduled nodes is shown in Figure 4.4(d) at II=3. The loop that was previously unmappable due to the restrictive scheduling of IMS Figure 4.1, is now mappable at II=3 due to randomization in assigning modulo schedule time.

If we take a closer look at the RC_ASAP and RC_ALAP times shown in Fig4.4(c) column 2 and 3, we can observe that there is a chance that the RC_ASAP may be the modulo schedule chosen for all the nodes, since assigning a modulo schedule time for the nodes from RC_ASAP and RC_ALAP is randomized. As seen in Figure 4.1(d)&(e), this schedule is not mappable. Unless there is a change to the workflow, there is a chance that finding a schedule that is unmappable and increasing the II to get a schedule process is repeated. To take care of this issue, CRIMSON proposes changes to the previous IMS-based workflow by statistically computing the feasibility of the scheduled nodes, prior to the mapping of the nodes. This makes sure that if a schedule is not mappable, a different random schedule is tried again for the same II. The number of times the mapping is tried for a given II is controlled by a threshold factor λ. With induced randomization in mapping and changes to the workflow, CRIMSON is able to achieve mapping of the application loops that were previously unmappable by IMS-based mapping techniques.

Suites	Loops	#nodes	#mem. nodes	#edges
MiBench	bitcount	22	4	28
	susan	31	8	35
	sha	31	10	39
	jpeg1	43	10	48
	jpeg2	28	6	33
Rodinia	kmeans1	15	6	17
	kmeans2	16	6	17
	kmeans3	17	4	20
	kmeans4	16	4	19
	kmeans5	12	2	13
	lud1	21	4	24
	lud2	20	4	24
	b+tree	13	2	13
	streamcluster	16	4	19
	nw	20	6	21
	BFS	28	10	32
	hotspot3D	76	20	96
	backprop	39	16	44
Parboil	spmv	25	8	27
	histo	18	4	20
	sad1	25	4	30
	sad2	19	4	20
	sad3	12	4	12
	stencil	69	16	94

Table 4.1: Benchmark Characteristics.

4.4 Results

Benchmarks: We profiled top three of the widely used benchmark suites namely, MiBench [104], Rodinia [105], and Parboil [106]. The top performance-critical, non-vectorizable loops [3] were chosen for the experiments. Loops that could not be compiled or the loops that were memory bound were not considered. Experiments were designed to consider only innermost loops so that a direct comparison with IMS can be made. These benchmarks depict a wide variety of applications from security,

[3]Maximum up to 5 loops per benchmark, with each contributing >7% of the execution time of the application when executed with standard inputs that are shipped with the benchmark suites.

telecomm etc. to parallel, high-perfomance computing (HPC) loops like spmv (sparse matrix-vector product). These loops on average across all the benchmark loops, corresponds to $\geq$50% of the total application execution time.

Compilation: For selecting the loops from the application and converting the loops to the corresponding DFG, we used CCF [107] - CGRA Compilation Framework (LLVM 4.0 [108] based). On top of the existing framework, to effectively compile the loops with control-dependencies (If-Then-Else structures), we implemented partial predication [76] as an LLVM pass, to convert the control-dependencies into data dependencies. Partial Predication [76] can efficiently handle loops with nested if-else structures. The loop characteristics are shown in Table 4.1 including the number of nodes in the DFG (only computing nodes are included and constants that can be passed in the immediate field of the ISA are excluded) and number of memory (load/store) nodes. The CCF framework [107] produces DFG of the loop with separate address generation and actual load/store functionality. Furthermore, during the addition of routing resources after scheduling, we have implemented path-sharing technique proposed in GraphMinor [30]. Path-sharing can reduce the redundant routing nodes added. We implemented CRIMSON as a pass in the CCF framework including the λ value computation and the feasibility test. We also implemented the IMS-based state-of-the-art RAMP [32] and GraphMinor [30] as a pass in CCF. Since, RAMP has demonstrated equal or better results when compared to GraphMinor, we compare CRIMSON against RAMP on of the IMS-based techniques. We compiled the applications of the benchmark suites using optimization level 3 to avoid including loops that can be vectorized by compiler optimizations. We considered 2D torus mesh CGRA of sizes 4×4, 5×5, 6×6, 7×7, and 8×8.

Suites	Loops	4x4			5x5		
		MII	RAMP	CRIM.	MII	RAMP	CRIM.
MiBench	bitcount	3	3	3	3	3	3
	susan	2	3	4	2	2	2
	sha	3	3	4	2	**X**	3
	jpeg1	3	**X**	6	2	**X**	4
	jpeg2	2	**X**	5	2	**X**	3
Rodinia	kmeans1	2	2	2	2	2	2
	kmeans2	2	2	2	2	2	2
	kmeans3	2	2	2	2	2	2
	kmeans4	2	2	2	2	2	2
	kmeans5	2	2	2	2	2	2
	lud1	2	2	2	2	2	2
	lud2	2	2	2	2	2	2
	b+tree	2	2	2	2	2	2
	streamcluster	2	2	2	2	2	2
	nw	3	3	3	2	3	2
	BFS	2	2	3	2	2	3
	hotspot3D	5	**X**	10	4	**X**	7
	backprop	5	**X**	7	4	4	4
Parboil	spmv	3	3	3	2	2	2
	histo	2	2	2	2	2	2
	sad1	2	2	2	2	2	2
	sad2	2	2	2	2	2	2
	sad3	2	2	2	2	2	2
	stencil	4	**X**	6	3	4	5

Table 4.2: Performance (II) Comparison Between IMS-based RAMP and CRIMSON (CRIM.) for Sizes 4×4 and 5×5. "**X**" Denotes That There Was No Mapping Obtained from RAMP. MII Denotes the Theoretical Minimum II.

4.4.1 Performance Evaluation

From Tables 4.2, 4.3 & 4.4, we can infer that for loops, *jpeg*1, *jpeg*2, *hotspot*3*D*, *backprop*, and *stencil*, IMS-based state-of-the-art heuristic RAMP, was not able to find a valid mapping for a 4×4 CGRA (denoted by "**X**" in Tables 4.2, 4.3 & 4.4). From the motivating example Figure 4.1, IMS produces almost the same modulo schedule time for most of the nodes for any increase in II. CRIMSON, on the other hand, facilitates the exploration of different modulo scheduling times for nodes of

Suites	Loops	6x6			7x7		
		MII	RAMP	CRIM.	MII	RAMP	CRIM.
MiBench	bitcount	3	3	3	3	3	3
	susan	2	2	2	2	2	2
	sha	2	3	2	2	2	3
	jpeg1	2	2	2	2	2	2
	jpeg2	2	**X**	2	2	2	2
Rodinia	kmeans1	2	2	2	2	2	2
	kmeans2	2	2	2	2	2	2
	kmeans3	2	2	2	2	2	2
	kmeans4	2	2	2	2	2	2
	kmeans5	2	2	2	2	2	2
	lud1	2	2	2	2	2	2
	lud2	2	2	2	2	2	2
	b+tree	2	2	2	2	2	2
	streamcluster	2	2	2	2	2	2
	nw	2	2	2	2	2	2
	BFS	2	2	3	2	2	2
	hotspot3D	4	**X**	7	3	**X**	6
	backprop	3	3	3	3	3	3
Parboil	spmv	2	2	2	2	2	2
	histo	2	2	2	2	2	2
	sad1	2	2	2	2	2	2
	sad2	2	2	2	2	2	2
	sad3	2	2	2	2	2	2
	stencil	3	3	3	3	3	4

Table 4.3: Performance (II) Comparison Between IMS-based RAMP and CRIMSON (CRIM.) for Sizes 6×6 and 7×7. "**X**" Denotes That There Was No Mapping Obtained from RAMP. MII Denotes the Theoretical Minimum II.

the DFG, resulting in a valid mapping. It is observed that even at a lower CGRA size 4×4, CRIMSON was able to map these particular loops. From Tables 4.2, 4.3 & 4.4, when running on RAMP, loops that were not mappable on a 4×4 CGRA, were eventually mapped when allocated enough resources. For example, *stencil* which was unmappable by RAMP on a 4×4 CGRA was mapped on a 5×5 CGRA due to allocation of additional resources. Therefore it can be said that the motivating example can also be mapped when allocated enough resources. From the motivating example,

Suites	Loops	8x8		
		MII	RAMP	CRIM.
MiBench	bitcount	3	3	3
	susan	2	2	2
	sha	2	2	4
	jpeg1	2	2	2
	jpeg2	2	2	2
Rodinia	kmeans1	2	2	2
	kmeans2	2	2	2
	kmeans3	2	2	2
	kmeans4	2	2	2
	kmeans5	2	2	2
	lud1	2	2	2
	lud2	2	2	2
	b+tree	2	2	2
	streamcluster	2	2	2
	nw	2	2	2
	BFS	2	2	3
	hotspot3D	3	**X**	4
	backprop	3	3	4
Parboil	spmv	2	2	2
	histo	2	2	2
	sad1	2	2	2
	sad2	2	2	2
	sad3	2	2	2
	stencil	2	2	2

Table 4.4: Performance (II) Comparison Between IMS-based RAMP and CRIMSON (CRIM.) for 8×8 CGRA. "**X**" Denotes That There Was No Mapping Obtained from RAMP. MII Denotes the Theoretical Minimum II.

if Figure 4.1(b) CGRA architecture was a 3×3 CGRA, then the IMS-based mapping algorithm would have used the extra resources provided to route the operation r. But this conclusion was not applicable to all the loops, meaning, loops such as *hotspot3D* and *jpeg2* were unable to find a valid mapping even when additional resources were allocated. RAMP was not able to achieve a mapping even at 8×8 CGRA for *hotspot3D* whereas RAMP was not able to achieve a mapping till 6×6 for *jpeg2*. While RAMP is able to map most of the loops at a higher CGRA size, CRIMSON

with effective randomized modulo scheduling was able to map all the loops at size 4×4. Additionally, for *sad*1 and *sad*3 loops, for which GraphMinor was not able to find a mapping, CRIMSON was able to achieve a mapping at 4×4 CGRA size.

From Tables 4.2, 4.3 & 4.4 we can observe that for loops mapped using RAMP, the II obtained from CRIMSON was comparable to RAMP across five different CGRA sizes ranging from 4×4 to 8×8. We can see an occasional spike in the II in CRIMSON for *susan* at 4×4 and *stencil* on 5×5, which is due to premature II increase by CRIMSON based on the λ value. To emphasize, λ is the maximum number of randomized schedules that are explored at the same II. A new schedule may be requested (i) on a failure to find a randomized schedule, (ii) on a failure of the feasibility test or, (iii) a failure to map. The λ value is not reset for a given II. After exhausting the λ limit, the II is increased and a new RC_ASAP and RC_ALAP is computed along with a new λ value. The λ value is computed by Equation 4.1 for each II. The λ value is determined by the user defined *exploration_factor*, which is the percentage of schedule space to that should be explored. If the *exploration_factor* is set too low, less modulo schedules are explored per II, thereby making it difficult to obtain a valid mapping and increasing the II prematurely. If the *exploration_factor* is set too high the time to obtained a valid schedule/mapping increases, which negatively affects the compilation time of CRIMSON. Tables 4.2, 4.3 & 4.4 comprehensively conveys that CRIMSON has a nearly identical performance compared to RAMP for all the loops across different CGRA architectures that RAMP was able to map and CRIMSON is better than RAMP by mapping the five loops that were not mappable by RAMP and seven loops that were not mappable by GraphMinor on a 4×4 CGRA. The II obtained from CRIMSON is not always equal to or better than state-of-the-art RAMP and is dependent on the λ value.

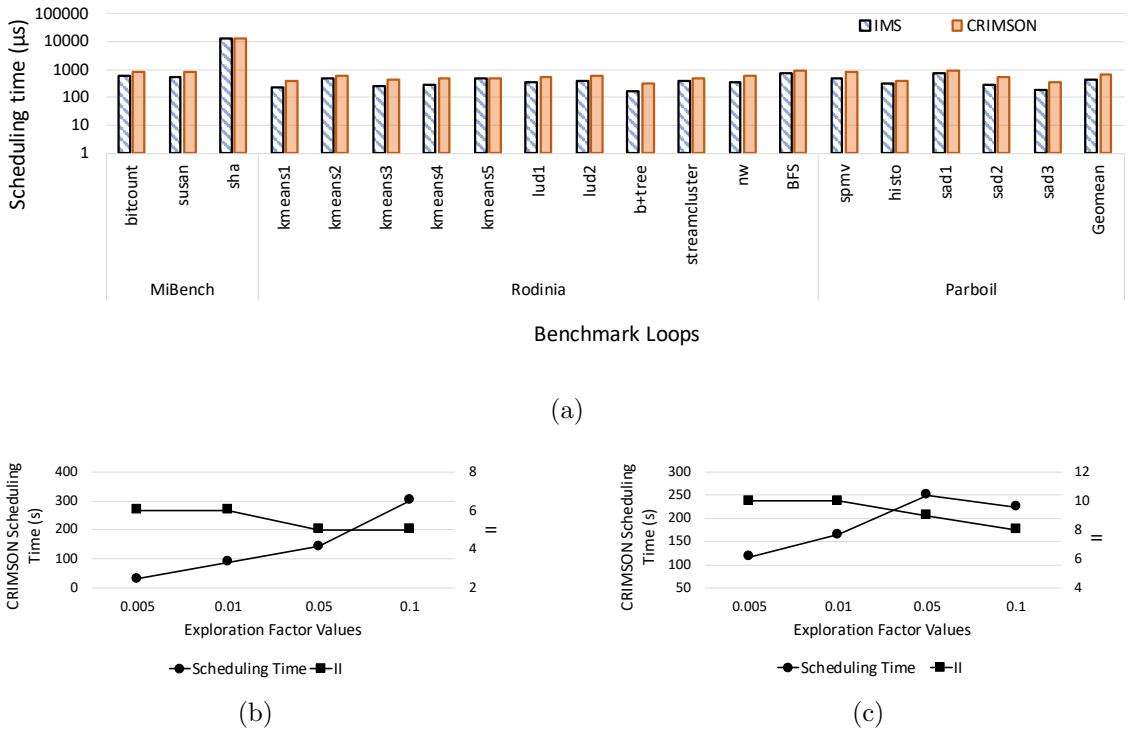


Figure 4.5: (A) Scheduling Time Comparison of CRIMSON with IMS. (B) Scheduling Time Vs. II Trade-off Trend for Stencil. (C) Scheduling Time Vs. II Trade-off Trend for Hotspot3d.

4.4.2 Scheduling time analysis between CRIMSON and IMS.

The scheduling time for IMS [34] and CRIMSON are shown in Figure 4.5a, which is reported based on the execution of both the algorithms on Intel-i7 running at 2.8GHz with 16GB memory. As shown in Figure 4.5a, the x-axis is the scheduling time i.e, time to obtain a valid schedule that is mappable, in μs (microseconds) and the y-axis corresponds to the benchmark loops. The 19 benchmarks shown in Figure 4.5a are those in which a mappable schedule was obtained by IMS. From Figure 4.5a, we can see that the scheduling time of CRIMSON is slightly higher than that of IMS. This is due to the additional computation of RC_ASAP and RC_ALAP, and the feasibility test (Figure 4.3). For the loops shown, the *exploration_factor* was kept at 0.005.

4.4.3 Trade-off analysis between scheduling time and II at different λ values.

From Equation 4.1, we can see that the λ value depends on the *exploration_factor*. This factor is defined as the percentage of modulo schedule space to be explored when there is an infeasible schedule or a mapping failure. The exploration_factor was changed from 0.5% (0.005) to 10% (0.1) and the corresponding scheduling time and II were recorded. The scheduling time numbers are recorded from executing CRIMSON on Intel-i7 running at 2.8GHz and 16GB memory and the compilation was performed for a 4×4 CGRA. A 4×4 CGRA was chosen because the II obtained by CRIMSON was much greater than the MII and the effect of λ can be shown clearly. In Figure 4.5b and Figure 4.5c, the left y-axis (primary axis) denotes the CRIMSON scheduling time, in seconds, and the right y-axis (secondary axis) denotes the II obtained. The x-axis denotes the different *exploration_factors*. From Equation 4.1 it is to be noted that as the exploration_factor increases, the λ value increases. From Figure 4.5b and Figure 4.5c, it is evident that as exploration_factor increases the CRIMSON scheduling time increases, due to elaborate exploration of the schedule space at a given II. For lower value of the exploration_factor, superficial exploration of modulo schedule space prematurely increases the II but at lower scheduling time. We can also note from Figure 4.5c at 0.1 that the above statement is not always true. At 0.1 the II decreases with the decrease in the scheduling time because a feasible and a mappable schedule was obtained earlier in the modulo schedule space exploration due to the innate randomness of the CRIMSON scheduling algorithm.

4.5 Chapter Summary

This chapter presented some of the major challenges encountered in the state-of-the-art mapping techniques with respect to scheduling and mapping of compute-

intensive loops onto the CGRA. The previous mapping techniques use IMS scheduling that rarely showed a change in the modulo schedules for increased II, which obstructed the mapping algorithm to map the application loops onto the CGRA architecture. Additionally, previous mapping techniques assumed that the obtained IMS schedule is mappable and started to map the scheduled nodes. On a failure to map, due to the limited connectivity of the PEs or addition of routing nodes, the mapping algorithms increase the II and call IMS again to get a schedule that almost never changes. To mitigate these challenges, this paper introduced CRIMSON, that comprehensively modeled RC-ASAP and RC-ALAP, picking a random modulo schedule time between these upper and lower boundaries. CRIMSON generated different schedules, thereby exploring different schedule spaces, on each invocation for a given or increased II. CRIMSON also introduced a novel feasibility test that pruned schedules that are unmappable. On evaluating the top 24 performance-critical loops from MiBench, Rodinia and Parboil, CRIMSON was able to map 5 application loops that were unmappable by RAMP and 7 application loops that were unmappable by GraphMinor. The II achieved by CRIMSON was comparable to the II achieved by RAMP for the application loops that were mappable by RAMP.

Chapter 5

PATHSEEKER: A FAST MAPPING ALGORITHM FOR CGRA

5.1 Background

In order to achieve the high performance and highly power-efficient operation of CGRAs good compilers are needed, which will be able to obtain a good quality mapping of performance-critical loops from applications. CGRA compilers can be classified into two categories: (1) Parallel-loop compilers, (2) Modulo Scheduling-based compilers. The parallel-loop compilers like the ones for[3, 22] employ various compiler optimizations to exploit the inherent spatial and temporal parallelization strategies to map parallel loops of an application onto the PEs of the accelerator [22]. However, not all the compute-intensive loops of an application may be parallel, and those can be accelerated through modulo scheduling-based compilers. Modulo-scheduling based compilers accelerate the data flow graph of the loop body through the pipelining present in the CGRAs using software pipelining [99, 25, 31, 30, 109, 32, 27, 28]. This paper focuses on the modulo scheduling-based compiler techniques that can support a wide variety of application loops.

One of the biggest limitations of the existing modulo scheduling-based state-of-the-art CGRA mapping techniques is that, when trying to map loops onto the CGRA if a mapping attempt fails, these techniques either discard the current mapping and restart anew or backtrack to the previously mapped node. Techniques that restart do not learn anything from the failure, and just blindly explore the mapping space. Even the backtracking based approaches may not be effective, as they recursively unmap the last mapped node, while the last node may not be the one that is making the

mapping infeasible. As a result, existing modulo scheduling-based state-of-the-art CGRA mapping techniques are unable to map some performance-critical loops even after 27 hours! This not only exacerbates the compilation time, but given reasonable limits on compilation time, it also negatively impacts the quality of the mapping achieved by these techniques.

To address these concerns, in this chapter, we present a novel mapping algorithm - PathSeeker. First, instead of backtracking or restarting the mapping like the previous mapping methods, PathSeeker analyzes the predecessor and successor nodes to find the reason behind the failed mapping. Second, PathSeeker explores local transformations for the predecessor and successor of the failed node to achieve a valid mapping. Finally, when local transformations do not yield a valid mapping, different PE positions of the other nodes in the time-slot of the failed node, the predecessor, and successor are iteratively explored, to find a valid mapping. We compare the mapping quality generated by PathSeeker to that of GraphMinor [30] and RAMP [32], which are state-of-the-art mapping algorithms in backtracking and restart, respectively. Experimental results on 35 application loops from the top three benchmark suites, MiBench [104], Rodinia [105], and Parboil [106] show that (i) PathSeeker can map all the 35 application loops on 4×4 CGRA, whereas GraphMinor and RAMP were not able to map 20 and 5 loops, respectively, (ii) PathSeeker achieves a better quality of mapping at lower compilation time with 550x and 10x compilation time speedup over GraphMinor and RAMP respectively, (iii) PathSeeker scales well across different sizes of CGRA.

A mapping failure can occur in mapping the scheduled onto the CGRA due to the limited connectivity among the PEs of the CGRA, and because of the need to map new routing nodes. Routing nodes occur when dependent operations are scheduled in non-contiguous timeslots. Figure‘5.1, shows the common mapping failure due to

limited connectivity on CGRA. In the context of response to a mapping failure, the existing mapping techniques can be classified into two categories, i) restart, and ii) backtrack. Genetic algorithms, simulated annealing [26, 51], minimum common subgraph (MCS) [25] or maximal clique [32, 31] based techniques can be classified as restart. Minimum common subgraph and maximal clique techniques discard the mapping on failure and simply search for another mapping. Simulated Annealing techniques try random time and PE placements.

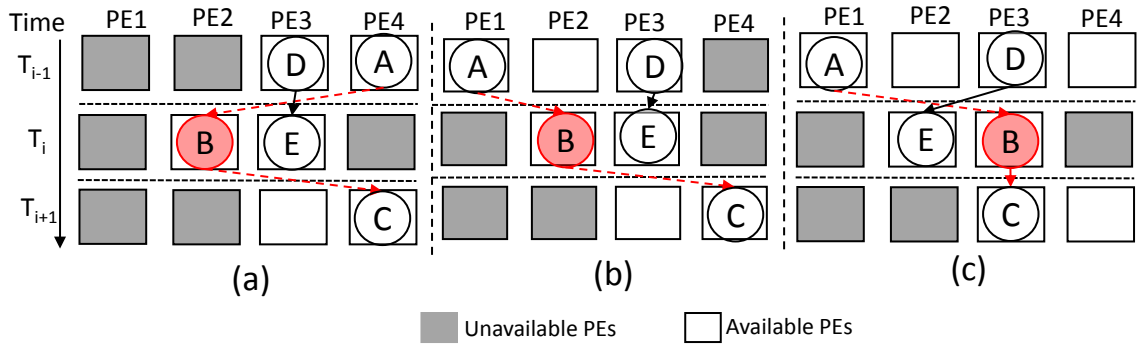


Figure 5.1: (a) b Cannot Receive Values From A or Pass Values to C, Resulting in a Failed Mapping. (B) B Is Unable to Pass Values to C, and (C) B Is Unable to Receive From A.

GraphMinor [30], RAMS [29], and BMS [110] perform backtracking on a mapping failure. RAMS and BMS form clusters from the DFG and map the clusters one-by-one. On a failure to map a node of the cluster, all the mapping of the current cluster is discarded and the algorithms backtrack to previously mapped cluster. However, GraphMinor maps the DFG by prioritizing nodes based on the critical path, one node at a time. On a failure, GraphMinor, backtracks to previously mapped node in the mapped order. Essentially, GraphMinor un-maps the last mapped node, and tries again by mapping that node to a different place. If that does not work, it continues to un-map the nodes in the reverse order in which they were mapped and keeps trying. However, the last node mapped may not be the problematic node. Even if that last

node were re-mapped, it might not enable a valid mapping.

5.2 Motivating Example

In this section we will map a simple DFG on a simple CGRA using the existing state-of-the-art techniques to illustrate how valid mapping opportunities may exist near mapping failures that can be achieved by some local rearrangement of the mapping. From the set of restart techniques, we picked Simulated Annealing [26] and RAMP [32], and from the set of backtracking techniques, we picked the GraphMinor [30] for this illustration.

Simulated Annealing and RAMP: The DFG to be mapped is shown in Fig. 5.2(a) and a 1×4 CGRA is shown Fig. 5.2(b). Simulated Annealing has an integrated scheduling and mapping algorithm employing randomization in the selection of time and PE, which takes a long time to find a valid mapping [30, 110]. On a failure to find a schedule and a PE, another random timeslot and PE are checked. Even when the mapping is just a few steps away there is no systematic way to find this out, so another random time and PE are chosen. For the example shown in Fig. 5.2(c), node 2 cannot be mapped due to the unavailability of connected PEs. By remapping node 4 from $PE4$ to $PE3$, a valid mapping for node 2 can be achieved. But the Simulated Annealing algorithm tries random placements again and restarts from the beginning. RAMP on the other hand tries to find a maximal clique (size of the clique should be equal to the number of nodes in the DFG), and does not identify or address the reason for failures. After a few restart attempts of finding different routing options for the DFG in Fig. 5.2(a) RAMP will be able to find a maximal clique. The maximal clique algorithm is time consuming, $\mathcal{O}(N^8)$ [32, 111], where N is a product of nodes in DFG and CGRA size. Restarting the algorithm on every failure will possibly lead to longer compilation time.

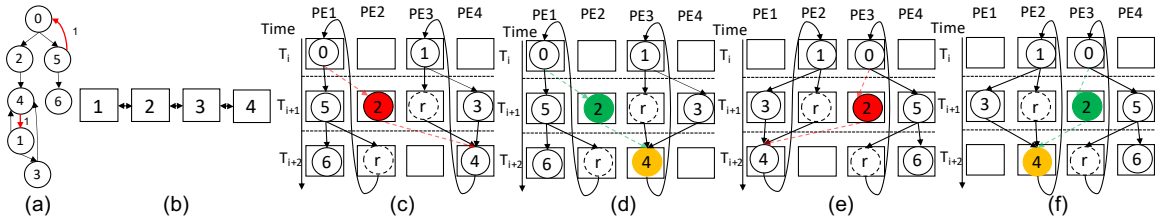


Figure 5.2: (A) DFG of an Application Loop. (B) a 1×4 CGRA Target Architecture. (C) Failure to Map Node 2 by Simulated Annealing, (D) PathSeeker Identifies the Problem and Remaps the Successor 4 and Finds a Valid Mapping for Node 2 (E) Failure to Map Node 2 by Graphminor, (F) PathSeeker Remaps Node 4 to $pe2$ to Find a Valid Mapping for 2.

GraphMinor: GraphMinor employs a backtracking approach, where on a mapping failure, the algorithm backtracks to previously mapped nodes and tries other placements. In GraphMinor, the order in which the nodes are mapped plays an important role in determining the compilation time. GraphMinor sorts the nodes of the DFG in the order of the critical-paths and cycles. Fig. 5.2(e) shows a mapping failure of node 2 due to unavailability of connected PEs (connected resource for PE1 and PE3 is PE2 that is occupied by r). GraphMinor backtracks to previously mapped node 6, which does not affect the mapping of 2. After backtracking through all the mapped nodes, GraphMinor reaches 4 wherein the actual problem lies. GraphMinor fails to identify that the problem was one step away – remapping node 4 would have fixed the mapping. Since the GraphMinor algorithm exhaustively explores the mapping search space, the compilation time increases exponentially with increase in DFG size and the CGRA size.

As illustrated in the motivating example, even a simple loop takes a lot of compilation to achieve a valid mapping, due to backtracking and restart mechanisms. Even if we take into consideration that these techniques produces a valid mapping, chances are it might not be a optimal mapping due to absence of failure-awareness. But these techniques do not produce a mapping for many of the critical loops with an allocated

100,000 seconds. Given the NP-completeness nature of the mapping we will never know if these techniques will produce a mapping within a finite time. The objective of PathSeeker is to achieve a good quality mapping within a limited amount of time for all the application loops considered across various sizes of CGRA.

Algorithm 3: *Schedule_And_Map*(DFG D, CGRA G)

```
1:  (RecMII, ResMII) ← Get_MII(D, G)
2:  MII ← Max(RecMII, ResMII)
3:  II ← MII
4:  while II ≤ Max_II do
5:     Schedule(II)
6:     G_II ← MRRG(G, II)
7:     D ← Update_Routing_Info(D)
8:     list ← CreateAdjacencyList(D)
9:     α ← search_parameter * D_nodes * G_size * II
10:    while mappingAttempts ≤ α do
11:       mappingAttempts = mappingAttempts + 1
12:       v ← Get_End_Nodes(D)
13:       while all_nodes_visited(D) ≠ true do
14:          if PathSeeker(list, v.pop()) then
15:             return success
16:          end if
17:       end while
18:    end while
19:    II = II + 1
20: end while
```

5.3 PathSeeker

An overview of the driver function, *Schedule_And_Map*() is shown in Algorithm 3. First, lines 1-3 compute the minimum II by taking in the DFG and CGRA size as inputs. The *Schedule*() function in line 5 schedules the nodes of the DFG at a

given II. Instead of having an as-soon-as-possible (ASAP) approach of IMS [34], the *Schedule*() function employs a randomized iterative modulo scheduling, which computes the resource constrained ASAP and ALAP for each node. This gives more mobility for the nodes to be scheduled. Next, a Modulo Resource Routing Graph (MRRG) and an adjacency list for Breadth First Search traversal is constructed in lines 6 and 8, respectively. MRRG has been extensively used in previous techniques like [26, 30, 32] etc. A simplified MRRG proposed in [30, 27] is used in PathSeeker. An MRRG is a directed graph, $G(V, E)$, extended II times, where V is the vertices and E is the edges of the DFG. Each node, $v \in V(G_{II})$ in MRRG is a tuple of (t, PE), where t is the timeslot, obtained from the scheduling algorithm, and PE is the PE resource at which the node v can be mapped. For $e = (x, y) \in E(G_{II})$, is an edge from x at (t, m) and y at $(t + 1, n)$, then the edge is a connection between the two CGRA resource m and n. Generally, CGRA PEs m and n are said to be connected if node x at (t, m) is connected to node y at $(t + 1, n)$ where $t \geq 0$. After the MRRG creation, an adjacency list is created for reverse-BFS graph traversal. For each node in the DFG, the adjacency list consists of the incoming nodes in the DFG. The graph traversal starts from the DFG's end nodes, i.e., nodes without any outgoing edges in the current cycle. Subsequently, the *PathSeeker* algorithm is invoked in line 14 to map the DFG.

For a given schedule, it may not be possible to find a valid mapping for a node because of the following two reasons (i) current placement of predecessors and/or successors may not have a connecting PE and (ii) if the predecessor and/or successor has a connecting PE that may be occupied by other nodes. Based on these observations, in order to find a valid mapping PathSeeker employs three stages of recovery, (i) try different placement of predecessor and/or successors, (ii) try different placements (remapping) of the nodes of the failed node's timeslot and try different placements to

find a valid mapping, and (iii) remap the nodes in predecessor and successor timeslots and try different placements for the predecessor and successor nodes until a valid mapping is achieved. On failure to achieve a mapping after stage three, PathSeeker tries a new random schedule again for a given II. Since PathSeeker maps the children nodes first, it is easier for the parent nodes to find PEs connected to their children. Line 9 computes α value that controls the number of times the PathSeeker algorithm should be restarted on a failure from all the three recovery stages. The II is incremented only if the mapping attempts reaches the α value. Until then different possible mappings are tried for a given II.

Algorithm 4: *PathSeeker*(List *AList*, Node v)

```
1: queue ← ∅
2: visited[v] = true
3: queue.push(v)
4: while queue ≠ ∅ do
5:    v = queue.front()
6:    queue.pop(v)
7:    if is_already_mapped(v) then
8:       continue
9:    end if
10:   P ← Get_Mapped_Pred(v)
11:   S ← Get_Mapped_Succ(v)
12:   Γ ← Get_Connected_PEs(v, P, S)
13:   if Γ.size() = 1 then
14:      PE ← Γ(0)
15:   else if Γ.size() > 1 then
16:      PE ← Γ(Rand(Γ.size())
17:   else
18:      if Localized_Search(v, P, S) ≠ true then
19:         if Recovery_Level_One(v, P, S) ≠ true then
20:            if Recovery_Level_Two(v, P, S) ≠ true then
21:               return failure
22:            end if
23:         end if
24:      end if
25:   end if
26:   SetMappablePositions(v, Γ)
27:   SetCurrentPosition(v, PE)
28:   for i in AList[v] do
29:      if visited[i] ≠ 0 then
30:         visited[i] = true
31:         queue.push(i)
32:      end if
33:   end for
34: end while
35: return success
```

5.3.1 Mapping Algorithm

The key contribution of PathSeeker is in its exploration and analysis the mapping failure in a systematic manner and the corrective course of actions it employs, pertaining to the node that was unmappable. Like the previous techniques, PathSeeker employs the two-step approach of scheduling the nodes of the DFG onto the time-extended CGRA followed by mapping (place and route) of the scheduled nodes onto the PEs of the CGRA. To find a valid mapping for the failure scenarios shown in Fig. 5.1(c)-(d), PathSeeker employs a three stage recovery process, (i) try different placements for successor and/or predecessor of the failed node, (ii) try different placements (remapping) for nodes in the failed node's timeslot and re-mapping the failed node, and (iii) remapping the nodes in successor and/or predecessor's timeslots and try different placements for the successor and predecessor nodes until a valid mapping is achieved. On a failure to achieve a mapping after these three stages, PathSeeker restarts the mapping process for a given II. To ensure an optimum mapping space exploration, the II is incremented only if the mapping attempts reaches α, as explained in the following subsection. Unlike existing techniques that use IMS [34] for scheduling the nodes, PathSeeker uses CRIMSON [112] to obtain a randomized iterative modulo schedule of the DFG onto the time-extended CGRA.

5.3.2 Failure-Aware Mapping & Novel Timeslot Level Remapping

The PathSeeker algorithm is shown in Algorithm 4. The lines 10 and 11, $Get\_Mapped\_Pred()$ and $Get\_Mapped\_Succ()$ routines, return only the predecessors and successors of the current node that are already mapped. $Get\_Connected\_PEs()$ function, in line 12, returns all the possible free PEs that are connected to the mapped predecessor and successors from the Modulo Resource Routing Graph (MRRG), used

in GraphMinor [30]. PathSeeker, starts the mapping in a reverse breadth-first search graph traversal (using an adjacency list $AList$) to aid the mapping of predecessors easily. This design decision was taken by analyzing the loops considered for the experiments. Since the nodes are already scheduled to a timeslot before mapping, taking a reverse breadth first search (BFS) approach will aid the mapping of predecessor node with lesser mapping failures. On the contrary, when we analyzed the breadth first search with predecessors mapped first followed by the successor nodes, due to the random placement of the predecessors, there was a high chance that the predecessor nodes are placed in non-connected PEs, which resulted in a successor node mapping failure. Based on the size of the Γ from line 12, the placement for the node v is chosen. For nodes where all the positions are possible a random position is obtained from line 16. After choosing a random position, all the possible placements i.e., Γ, and the selected PE position are stored for recovery purposes by $SetMappablePositions()$ and $SetCurrentPosition()$ routines, respectively in lines 26 and 27. An empty Γ from line 12, means that there were no possible placements available for the node v. At this juncture, PathSeeker employs a three-tier recovery approach to find a valid placement for node v. In line 18, the $Localized\_Search()$ routine is invoked. On a failure of this routine, in line 19 $Recovery\_Level\_One()$ routine is called. On a Level One failure, $Recovery\_Level\_Two()$ routine is employed, in line 20. The Level One and Level Two routines use complex timeslot level remapping procedures to find a valid mapping for node v. In an event of all three recovery failures, the PathSeeker algorithm is restarted for the given II.

Algorithm 5: *Localized_Search*(Node v, Predecessor P, Successor S)

```
1: timeslot ← Get_Modulo_Schedule_Time(v)
2: mapped_pred_succ ← P.size() + S.Size()
3: succ_map_set ← ∅
4: pred_map_set ← ∅
5: v_pe ← Get_Free_PEs(timeslot)
6: for i in v_pe do
7:    for j in S do
8:       for k in P do
9:          p_pe ← GetMappablePositions(k)
10:         for kk in p_pe do
11:            if connectedPEs(i, kk, jj) then
12:               Γ.insert(i)
13:               succ_map_set.insert(k)
14:               pred_map_set.insert(j)
15:               store_connected_pes(i, kk, jj)
16:            end if
17:         end for
18:      end for
19:   end for
20: end for
21: if Γ.size() = 0 then
22:    return false
23: end if
24: Update Γ and PE values
25: return success
```

Localized Search: As a first step in the recovery process to find a valid mapping, PathSeeker invokes the *Localized_Search*() algorithm shown in Algorithm 5. On a failure to map a node the algorithm searches through the possible positions of the predecessors and the successors to find a valid placement for the failed node. Lines 6-20 search through both predecessors' and successors' possible positions when there

are mapped predecessors and successors for the failed node. This localized search routine does not modify any other nodes that are already mapped onto the CGRA. The *GetMappablePositions*() function in lines 9 retrieves the possible PE positions. On finding a successful connected PE, the PE position for node v (failure node) is store into Γ array. A valid mapping is obtained for node v only if all the predecessors and successors have a connected PE to v. Line 24 updates the predecessor nodes and successor nodes, which is used to check if all the predecessors and successors were able to successfully find a connected PE. On failure of this localized search (when Γ's size is 0), PathSeeker invokes the *Recovery_Level_One*() routine, from Algorithm 4, line 19. Algorithm 5 lines 8 and 10 are modified when the failed node does not have a predecessor mapped.

For the example error model showed in Figure 5.1(b) and Figure 5.1(c), the *Localized_Search*() algorithm collects all the mappable positions of the predecessors (A) and the successors node (C) of the failed node (B), and iterates through all the mappable position to until a valid mapping for failed node B is achieved. For Figure 5.1(b), since the B is not able to pass the computed value to node C, the alternate positions for node C are tried and eventually C is moved to PE3 for a valid mapping. For Figure 5.1(c), since B is able to receive the value from the predecessor A, node A is moved to PE2 or PE4 for a valid mapping of B.

This algorithm checks all the possible PE positions for all the successors and predecessor of the failed node until a mapping is found. Since the initial mapping is reverse-BFS, checking for successors followed by predecessors helps in finding the mapping faster. The *Localized_Search*() algorithm is particularly effective in finding a valid mapping faster because rather than backtracking to the previous mapped node that may or may not be directly related to the failed node, *Localized_Search*() tries find a valid mapping from the failed node's perspective by moving the predecessor

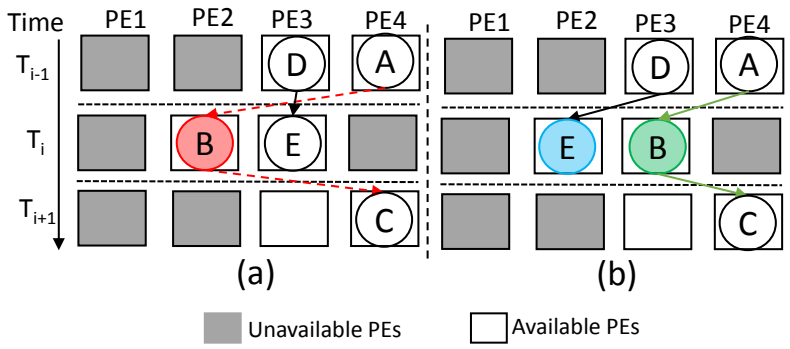


Figure 5.3: a) B Fails Because the It Is Not Able to Receive Value from A or Pass Value to C. (B) PathSeeker Identifies the Failure and Swaps B and E in the Timeslot to Get a Valid Mapping.

and/or successor nodes of the failed node. On a failure to find a successful mapping for the failed node, the successors' and predecessors' positions are reset before calling the Level One recovery routine.

Recovery Level One: On a failure of localized search, the recovery Level One routine is invoked. This routine employs the novel timeslot level remapping. The remapping starts by collecting all the nodes mapped to current timeslot as that of the failed node. Next, the remapping algorithm iterates over the mappable positions of each node and remaps them. On remapping each node, the valid position for the failed node is checked. This local rearrangement of the already mapped nodes to the timeslot is the novelty of PathSeeker, and it helps to change the course of the mapping. On a successful mapping of the failed node, the current remapping of the nodes is finalized and their positions are updated.

The Level One recovery mechanism is illustrated in Figure 5.3. In Figure 5.3(a), B is unable to receive or pass the values from A or C respectively, and localized moving of C may not yield a valid mapping. In this scenario, Level One recovery routine collects the nodes mapped B's timeslot, i.e., and tries to move E until a valid mapping is achieved. By swapping the PE positions B and E Level One routine find a

valid mapping. This rearrangement also honors all the data and control dependencies for the nodes that are shuffled.

Recovery Level Two: On a failure of Level One recovery, Level Two recovery routine is invoked. This routine is similar to Level One recovery, except Level Two recovery remaps nodes that are mapped to the successor and predecessor timeslots, T_{i+1} and T_{i-1}, and checks for a valid mapping for the updated positions of successor and predecessor nodes. On a failure from this recovery routine, the mapping can be restarted for a given II. The remapping algorithm collects all the nodes mapped to the current timeslot (T, timeslot of the failed node), the successor timeslot (T_{i-1} of the failed node), and the predecessor timeslot (T_{i+1} of the failed node). Iterating through the mappable nodes for each of the nodes in each timeslot, a valid mapping position for the failed node is checked. On success, i.e., if a valid mapping for the failed node is found, then the current position and the mappable positions for each of the nodes are updated based on the new positions via $SetMappablePositions$ and $SetCurrentPosition$. This novel approach of predecessor and successor timeslot level remapping aids in finding a mapping for the failed nodes, if there is a mapping available by shifting around the already mapped nodes to a new position honoring the data dependencies and connectivity. As far as we have explored, PathSeeker's timeslot level recovery mechanism is a novelty when comparing it to other popular mapping algorithms employed for CGRAs. On a failure from the Level Two recovery, the mapping can be restarted for a given II. This exploration is controlled by α computed by; $\alpha = \lceil search\_parameter \times n \times PEs \times II \rceil$ where, n is the number of nodes in DFG, PEs is the total number of PEs in CGRA, and $search\_parameter$ is the percentage of the mapping search space to be explored. Since a new α value is computed for a given II, the exploration of search space is dependent on the loop characteristics. The II is increased only when the number of restarts exceeds α.

The time complexity of the graph traversal is $\mathcal{O}(V + E)$ since PathSeeker uses an adjacency list approach for reverse-breadth first search, where V and E are nodes and edges of the DFG. The initial mapping of the nodes are randomized selections from the list of PEs (line 16 of Algorithm 4). The time complexity of which is $\mathcal{O}(1)$. On a failure to map a node, the time complexity of the Level Two recovery mechanism is $\mathcal{O}(N^3)$, where N is the product of nodes mapped to each timeslot and number of mappable positions of the nodes.

5.3.3 Running Example

Fig. 5.2(d)&(f) shows the working of the PathSeeker technique on a failure to map node 2. Fig. 5.2(c) shows failure to map node 2 by Simulated Annealing and Fig. 5.2(e) shows the failure to map node 2 encountered by GraphMinor. For the failure in Fig. 5.2(c), PathSeeker's *Localized_Search* function is invoked first which gets the predecessors and successors of failed node 2, i.e., node 0 and node 4, respectively. PathSeeker iterates through all the possible positions to find a valid mapping of the successor, and consecutively the predecessor. There is just one possible position for node 4, i.e., $PE3$, which meets all the dependencies. A valid mapping by PathSeeker for this failure case is shown in Fig. 5.2(d). For the failure case shown in Fig. 5.2(e), PathSeeker follows the same procedure. The possible positions for node 4 are $PE1$ and $PE2$. Since $PE1$ is the current position, PathSeeker tries $PE2$ which results in a successful placement. A valid mapping by PathSeeker for this failure case is shown in Fig. 5.2(f). It can be observed from Fig. 5.2(d) and (f) that PathSeeker's *Localized_Search* does not modify the placements of other mapped nodes and instead only explores within the existing mapping. In a hypothetical case where there is no possible mapping available for node 2, Level One recovery routine will be called to remap the nodes in timeslot T_{i+1}. On a failure to find a valid mapping from Level

Loops	G-Minor (s)	RAMP (s)	PathSeeker (s)
bfs	**NA**	100.64	2.76
backprop1	2.066	3.4	6.6
backprop2	**NA**	50.1	10.25
b+tree1	2.56	3.53	2.87
b+tree2	**NA**	**NA**	2.61
kmeans1	1.54	3.23	4.41
kmeans2	297.9	2.53	4.45
lud1	**NA**	1.67	4.49
lud2	**NA**	2.43	3.02
streamcluster1	**NA**	8.45	11.67
streamcluster2	**NA**	11.83	16.25
nn1	6588.99	**NA**	2.42
nn2	1.26	1.05	2.69
particlefilter1	2.36	2.03	1.56
particlefilter2	**NA**	35.05	1.53
mri-gridding1	**NA**	3.01	1.58
mri-gridding2	**NA**	2.67	1.47

Table 5.1: PathSeeker Has a Better Compilation Compared to Graphminor and RAMP. **NA** Denotes the Loops for Which a Valid Mapping Was Not Obtained Within the 100,000 Seconds Threshold.

One recovery, Level Two recovery function will be called to remap the nodes in timeslot T_{i+2} and subsequently the nodes in timeslot T_i, which are the successor and predecessor timeslots of node 2. While previous techniques explore the design space on a node-by-node basis, PathSeeker's Level One and Level Two recovery remaps all the nodes in the timeslot of the failed node, its predecessors, and its successors until a valid mapping is achieved.

Loops	G-Minor (s)	RAMP (s)	PathSeeker (s)
myocyte1	**NA**	2.86	23.31
myocyte2	**NA**	66.84	2.95
srad1	1.55	1.414	1.31
srad2	3.88	3.44	1.54
bitcount	1.61	4.35	1.7
fft	1.0969	1.68	1.33
gsm	**NA**	2224.52	115.61
patricia1	**NA**	**NA**	0.9182
patricia2	**NA**	**NA**	134.31
sha	**NA**	32.89	5.93
basicmath	1.8591	**NA**	13.98
stringsearch1	**NA**	3.389	2.2
stringsearch2	**NA**	65.94	1.46
susan	**NA**	49.04	16.56
spmv	**NA**	32.48	2.66
histo	396.68	15.66	1.56
sad1	1857.57	8.28	33.8
sad2	12250.82	21.43	2.89

Table 5.2: Results Continued from Table 5.1

5.4 Results

We profiled applications from three widely used benchmark suites [1] MiBench [104], Rodinia [105], and Parboil [106]. These benchmarks depict a wide variety of application domains comprising of embedded system applications like automotive, industry, office, network, security, and telecommunication, heterogeneous applications like data mining, pattern recognition, image processing, graph algorithms, and high performance computing application like spare matrix-dense vector multiplication (spmv) . We have considered a total of 35 application loops from these benchmark suites consisting of top two performance-critical loop from each application. We profiled

[1] Top two performance-critical loops were chosen from each application, with each contributing $> 7\%$ of the execution time of the application when executed with standard inputs that were shipped with the benchmark suites.

the loops for compute-intensive application and application that were memory intensive (I/O bound) we not considered. In case of application with nested loops, only the innermost loop was considered for acceleration. Additionally, the 35 loops considered for evaluation are not vectorizable, i.e., the acceleration achieved is beyond vectorization (SIMD) like Intel's AVX, ARM's SVE etc. The extraction of loops and converting them to Data Flow Graph (DFG) were performed using CCF [107], an LLVM 4.0 [108] based CGRA compilation and simulation framework. Additionally, we have implemented partial predication [76], for compiling if-then-else and nested if-then-else structures efficiently. CCF framework [107] produces DFG of the loop with separate address generation and actual load/store functionality. We have also implemented path-sharing, proposed in GraphMinor [30], when adding routing nodes, which can effectively reduce the overhead. We have implemented RAMP [32], GraphMinor [30], and PathSeeker (proposed technique) mapping algorithms as a pass in CCF. We compiled the application loops with optimization level 3, to avoid those loops that are vectorizable by the compiler. We also scaled the three mapping algorithms across five CGRA sizes, namely 4×4, 5×5, 6×6, 7×7, and 8×8 and analyzed the scalability results.

5.4.1 Performance Evaluation

Fig.5.4a and Fig. 5.4b shows the performance comparison of PathSeeker with GraphMinor and RAMP. The values were recorded by executing PathSeeker, RAMP and GraphMinor on an Intel-i7 running at 2.8 GHz with 16 GB memory. A 4×4 CGRA was used for this experiment. The compilation time threshold was kept at 100,000 seconds. It can be infered from Fig.5.4a and Fig. 5.4b that PathSeeker, with its novel remapping scheme was able to map all the 35 loops considered, whereas GraphMinor and RAMP were not able to map 20 and 5 loops, respectively. The loops

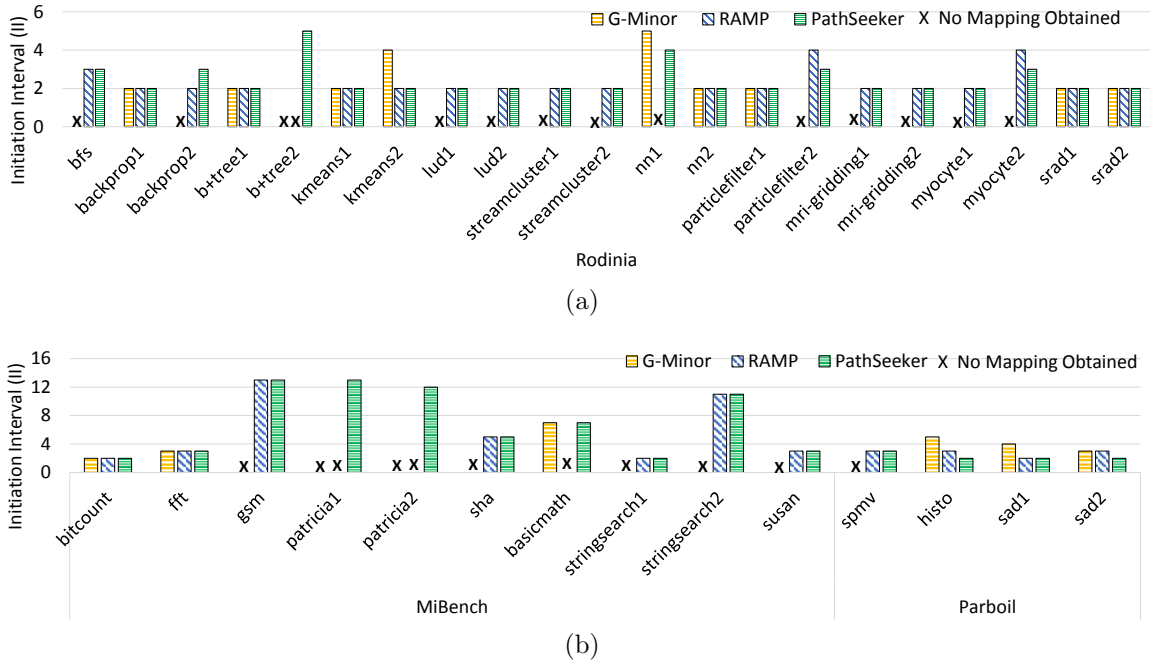


Figure 5.4: II Comparison of PathSeeker with Graphminor (G-minor) and RAMP. **"x"** In the Graph Denotes That There Was No Obtained for until the Threshold Time. (A) Benchmark Loops from Rodinia, (B) Benchmark Loops from Mibench and Parboil.

for which a valid mapping cannot be obtained within 100,000 seconds are denoted by "**X**" in the Fig. 5.4a and Fig. 5.4b.

The II obtained from GraphMinor and RAMP are not always optimal (lower II is better). This can particularly be noted in loops such as *kmeans*2, *nn*1, *histo* and *sad*1 where GraphMinor had higher II, and *particlefilter*2, *myocyte*2, *histo*, and *sad*2 for which RAMP had higher II. Considering the loops for which the GraphMinor has obtained a valid mapping within 100,000 seconds, PathSeeker showed a 28% increase in performance, i.e., lower II. Compared to RAMP, PathSeeker achieved a comparable performance in all the loops and had better performance in five loops mentioned above. Due to the novel routing strategy search to achieve a valid mapping, RAMP had a better II for *backprop*2. When there is a mapping failure from all three recovery schemes, a new random mapping is tried until the α value is exhausted, ensuring a

lower II. The II is increased only when the number of mapping tries exceeds α.

Table 5.1 and Table 5.2, shows the compilation time comparison of PathSeeker with GraphMinor and RAMP. From Table 5.1 and Table 5.2 it can be observed that exhaustive search space exploration and backtracking to the previously mapped nodes, and exploring different routing options by restarting the mapping, increases the compilation time drastically. For GraphMinor and RAMP, as the number of CGRA resources increases, the compilation time increases exponentially. The exponential compilation time increase for GraphMinor is due to the fact that, on a failure to find a valid mapping, the backtracking algorithm reverts back to the previously mapped nodes, un-maps the nodes and exhaustively remaps the previous node until a valid mapping is achieved. With increase in the CGRA size, the mapping search space also increases, thereby increasing the compilation time. In RAMP, given the algorithmic time complexity, an increase in CGRA size increases the compilation time drastically. PathSeeker's initial randomized mapping and novel three-tier failure recovery approach, has resulted in a significant reduction in the compilation time while achieving a valid mapping. PathSeeker was able to map all the loops on a 4×4 CGRA within a few tens of seconds. For the loops that were mappable within 100,000 seconds by the comparative techniques, PathSeeker achieved a 10x speedup over the average compilation times of RAMP, and 420x speedup over average compilation time of GraphMinor.

Due to the exhaustive search and arbitrary backtracking mechanism, we can observe that the II obtained from GraphMinor is not always optimal. The recovery mechanism and reshuffling of the nodes connected to the failed nodes makes PathSeeker an effective mapping tool to map compute-intensive loops on to CGRA. For example, from Fig. 5.4b *sad*1, we can see that the II achieved by PathSeeker is better than GraphMinor. Analyzing this loop in detail we find that there were 43 mapping

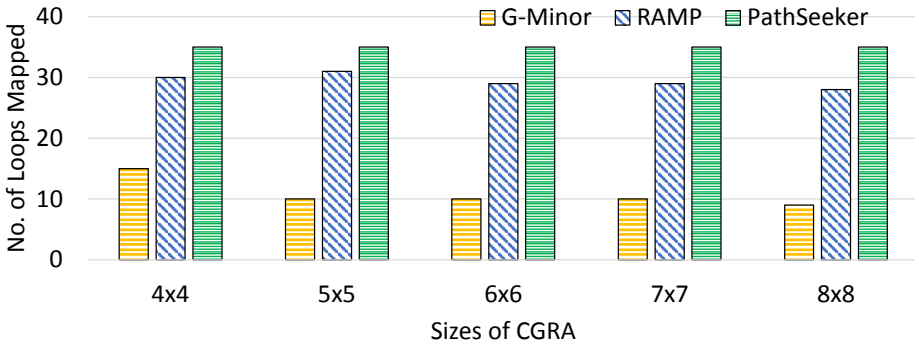


Figure 5.5: PathSeeker Is Able to Achieve a Valid Mapping for the All the 35 Loops Considered Across Various Sizes of CGRA.

failures encountered, and 19 of those failures were rectified in the localized remapping of the predecessor and successor nodes. Level One recovery was able to rectify 16 of the errors by shuffling the nodes in the failed node's timeslot and 11 of the errors were rectified in Level Two recovery routine which employs novel timeslot level remapping. There were 7 failures for which a valid mapping was not obtained from all the three recovery stages and the mapping algorithm was restarted. The restarting is done for the given II and not increased until the α value reached. The novel recovery mechanism along with randomized placements within the given II ensured lower II for *sad*1 compared to GraphMinor.

5.4.2 Scalability Analysis

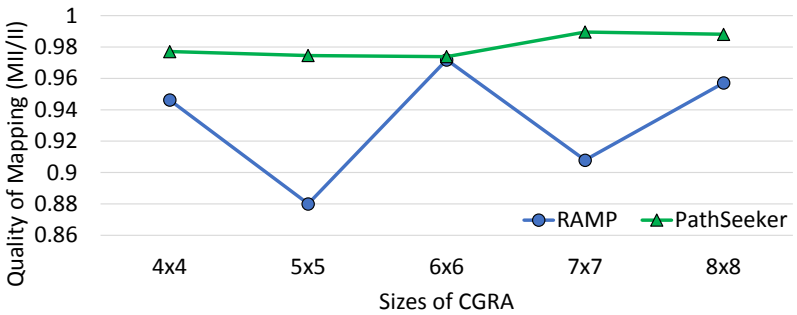


Figure 5.6: PathSeeker Achieves a Superior Mapping Quality (II Closer to MII) Compared to RAMP.

We performed the scalability experiment for CGRA sizes of 5×5, 6×6, 7×7, and 8×8. Figure 5.5, shows the scalability of PathSeeker with respect to GraphMinor and RAMP. We can observe that as the size of the CGRA increases the number loops mappable by GraphMinor and RAMP reduces. PathSeeker, on the other hand, is able to achieve a valid mapping for all the 35 loops considered. Due to the backtracking mechanism, GraphMinor was not able to find a mapping within the threshold of almost 75% of the loops. Figure 5.5 clearly shows that arbitrary backtracking to the previously mapped nodes on encountering a mapping failure is clearly not a scalable solution.

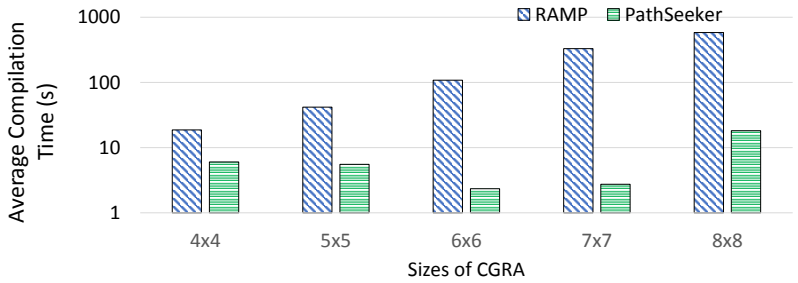


Figure 5.7: PathSeeker Achieves a Mapping for All the Loops Across Various Sizes of CGRA at a Lower Compilation Time.

The Minimum II (MII) is the minimum possible II that can be achieved for a given loop DFG and the CGRA architecture. The quality of mapping of a mapping algorithm is the ratio of MII/II, which indicates how close the obtained II is to the MII. Figure 5.6, shows the quality of mapping of RAMP and PathSeeker across all the five CGRA sizes. The mapping quality achieved by PathSeeker is better across all the CGRA sizes, compared to RAMP. On a mapping failure, RAMP discards the maximal clique obtained and restarts the mapping, usually at an increased II. This accounts for the drop in mapping quality and increased compilation time.

Fig. 5.7, shows the scaling of average compilation times of RAMP and PathSeeker, considering only the loops that were mappable by RAMP. The x-axis of Fig. 5.7 shows

the average compilation time across all the benchmark loops for which RAMP was able to achieve a valid mapping, in log scale, and the y-axis shows the various sizes of the CGRA. As shown in Fig. 5.7, the compilation time of RAMP increases exponentially, due to the restart mechanism on encountering a failure and its algorithmic complexity. In comparison, the compilation time of PathSeeker scales linearly, due to the initial randomized placement of the nodes and localized modifications of the mapping pertaining to the failed nodes.

5.5 Chapter Summary

This paper presented a novel CGRA mapping scheme, PathSeeker, that was able to map all the loops in a smaller CGRA size, with better II and lower compilation time. Existing techniques, such as GraphMinor and RAMP, resort to backtracking to a previously mapped node or restarting the mapping process, when encountering a mapping failure. This leads to a significant increase in the compilation time and poor II. PathSeeker's novelty lies in employing localized search strategies and time-slot level remapping to rectify a mapping failure. PathSeeker was able to map all the 5 top performance-critical loops across three widely used benchmark suite loops on a 4×4 CGRA, whereas GraphMinor and RAMP were not able to map 20 and 5 loops on the same CGRA size, respectively. On comparing the loops that were mappable by GraphMinor and RAMP, PathSeeker achieved a 28% lower II compared to GraphMinor and 3% lower II compared to RAMP on a 4×4 CGRA. PathSeeker was able to get a 420x and 10x compilation time improvement compared to GraphMinor and RAMP, respectively.

Chapter 6

LASER: EFFICIENT METHOD FOR MAPPING CONDITIONAL LOOPS

6.1 Background

Previous compiler techniques such as [31, 28, 25] accelerate only the innermost loop and fall short in accelerating rest of the loop nest which in turn has to be executed on a core. The communication overhead also multiplies if the trip count of outer-loop is higher. Existing techniques such as [65, 68] are restricted to handle only perfectly nested loops with 2-level. On the other hand, flattening based approach of [66] is promising but restricts the scalability because of its hardware-based solution with modified PE architecture. Major techniques to accelerate loops with conditionals are - (i) Full predication, (ii) Partial Predication, (iii) Dual-Issue and (iv) Path Selection Based Mapping (PSB). Full and partial predication schemes requires predicated register files and muxes to communicate the branch outcome. Full predication maps the nodes from both the if- and else- path on the same PE, but at different time, so that correct value is updated at the end of the execution [43]. Partial predication allows execution of nodes from both paths simultaneously but correct outcome needs to be selected through additional select node [76]. Dual issue schemes such as [43] fetches instructions for both paths but executes instructions of only correct path based on the branch condition, but requires additional mux in each PE to select the if-path or else-path instructions and is applicable to single-level only. Path selection based approach [77] selectively issues the instruction based on the branch outcome, but is applicable to only single if-then-else. For nested-conditionals PSB relies on partial predication. In this paper, we evaluate partial predication as it is the only technique

that can map loops with nested conditional at lower II.

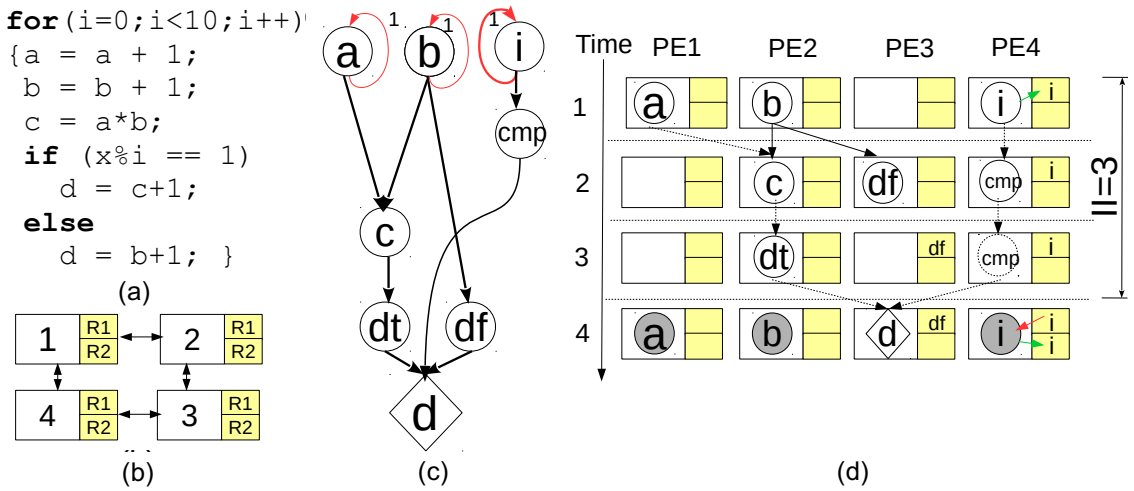


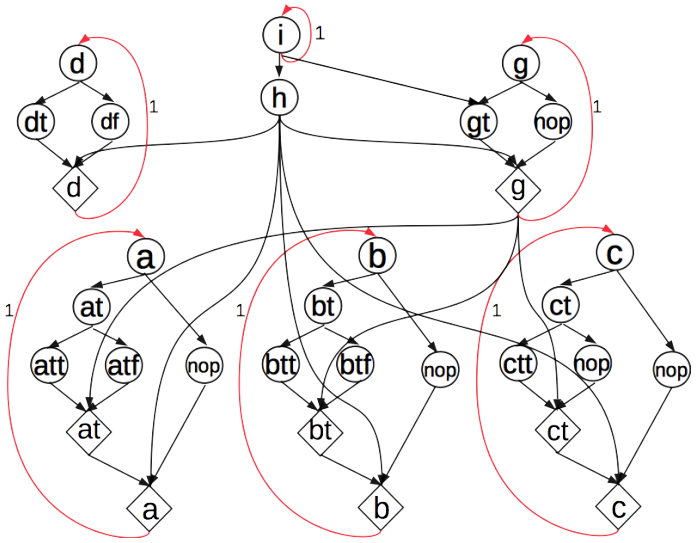
Figure 6.1: (A) a Simple Loop to Be Accelerated on CGRA (B) Flattened 2×2 CGRA Where Each PE Has 2 Registers (C) a Loop with an If-then-else (D) Data Flow Graph (DFG) of the Loop with Partial Predication (E) Mapping of DFG on 2×2 CGRA With II=3.

6.1.1 Partial Predication incurs high overhead

In partial predication, the nodes of DFG from both true and false paths can be mapped on different PEs and a select operation is required to choose the correct outcome based on the condition evaluated. Fig 6.1(a) shows a simple loop with conditional, while Fig 6.1(c) shows DFG using partial predication. Node *cmp* represents condition $x\%i==1$. Nodes d_t and d_f are true and false paths of d and a selection operation is added. Mapping of the DFG is shown in Fig 6.1(e) with II is 3. Due to the additional nodes required by partial predication, if a variable is computed inside the innermost nest of if-then-else, there is a corresponding node for operation inside each if-path and an else-path and so is a selection. Applying partial predication on a loop with nested conditional in Fig 6.2(a), we get DFG shown in Fig 6.2(b). Mapping DFG on 2×2 CGRA yields II of 11! Partial predication method increases the number of nodes in accelerating performance-critical loops with nested conditionals and the

```
1:for(i=0;i<10;i++){
2: if (x%i==1){d+=0;
3:  if (y%i==1){
4:  a+=0;
5:  b+=0;
6:  c+=0;}else{a=a+1;
7:              b=b+1;}}
8:  else d=d+1; }
```

(a)



(b)

Figure 6.2: (A) a Loop With Nested Conditional (B) DFG Using Partial Predication Results in 31 Nodes. Nodes h and g Represent Conditions $x\%i==1$ and $y\%i==1$.

nested loops from MiBench benchmark suite. Clearly, there is no technique that can accelerate nested loops and nested conditionals with less overhead.

6.2 LASER: Loop Acceleration By Selective Execution

The compiler transforms arbitrary nested (perfect or imperfect) loops into a single loop with nested conditional by **loop flattening**[66]. Fig 6.3 shows the transformation of a simple nested loop into a single-level loop with nested conditional. In some

special cases, nested loops cannot be converted into a single loop [1] . However, in general, loop flattening is needed to convert a nested loop to a loop with conditional statements. Executing branches on CGRA is challenging due to the lack of support from the CGRA's instruction fetch unit (IFU). The existing CGRA IFU issues instructions sequentially from the instruction memory and hence cannot jump memory addresses in case of conditional operations. In LASER, we enhance the CGRA IFU functionality to issue only the instructions of the correct path [2] at runtime. For the correct-path instructions to be issued by the IFU, LASER compiler lays out the program instruction in a specific way such that the IFU jumps to the exact memory location of instruction of the correct-path and issue them at runtime.

```
for (;cond1;) {
 /*statements*/
  for (;cond2;) {
   /*statements*/
  }
   /*statements*/
}
```

(a)

```
for (;cond3;) {
  if(cond4) {
   /*outer for-loop statements
       and iterator calculations*/
     }
  else {
   /* inner for-loop statements
       and iterator calculations*/
} }
```

(b)

Figure 6.3: (A) an Imperfectly Nested Loop with `Cond1` and `Cond2` Conditions (B) Flattening Converts (a) into Single-level Loop with Conditionals with New `Cond3` and `Cond4`

With this IFU support to issue correct-path instructions, if a variable c is updated in both true and false path, mapping c_t and c_f on different PEs without a *select* operation will lead to an incorrect execution. This is because the compiler generates

[1] If a loop contains sibling loops, flattening based approach may be impractical, so a loop fission approach [66] should be used. We did not come across any compute-intensive loops that have sibling loops, in our experiments.

[2] Either true-path or false-path based on the branch outcome at runtime.

instructions statically and since the correct-path executed is unknown at the static time, the PE that will hold the correct value of c at the end of the execution is also unknown. This discrepancy can lead to errors in the value of c at the end of program execution. To overcome this, LASER compiler fuses the true-path operation and false-path operation of the variable into a single node, $\langle c_t, c_f \rangle$. This single fused node is mapped to only one PE of the CGRA and only one instruction (either true-path or false-path) is issued at runtime by the IFU. After the execution of the instruction the PE on which the fused node was mapped, holds the correct value of c. Similarly, if a variable d is updated in only one path (only in true-path (d_t) and not updated in the false-path) the compiler creates a no-operation (nop) for the false path and performs the fusing. The fused node will now have $\langle d_t, nop \rangle$, which means that if the branch condition is true d_t is issued by IFU otherwise a nop is issued. LASER compiler transforms complicated loops, maps them on to the CGRA architecture and lays the instructions in the memory in a specific manner, such that the IFU can fetch the instructions from correct-path at runtime.

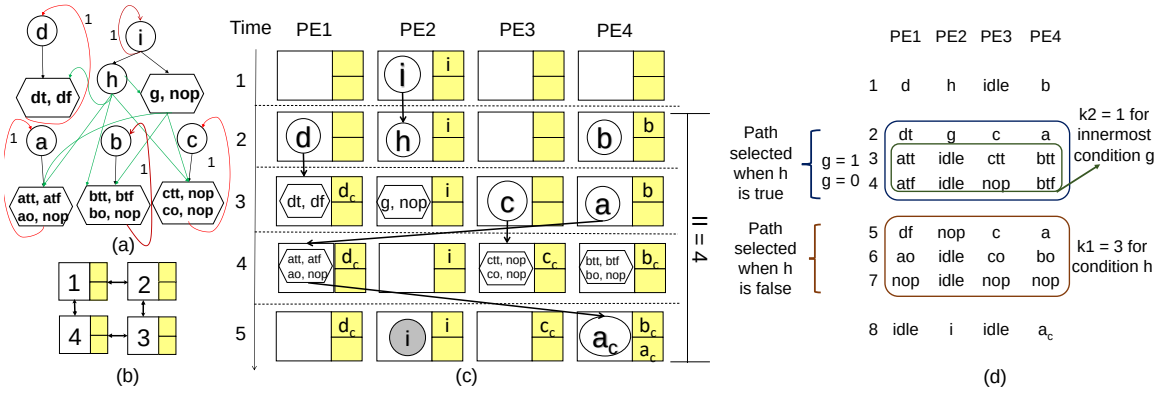


Figure 6.4: (A) DFG Obtained from LASER-compiler for Loop of Fig 6.2. Nodes from Multiple If-paths and Else-path to a Single Node. If Such Path Is Absent, Balancing No-ops Are Added and a Node Such as a_o Preserves the Old Value. (B) 2×2 CGRA Where Each PE Has 2 Registers. (C) Mapping with $II = 4$. (D) Instructions Are Selectively Issued During the Execution of the Kernel.

6.2.1 Compiler Method

By evaluating the condition of a nest a priori and then mapping the true and false path of the nest on to the same PE, LASER-compiler reduces the total number of nodes created. For example, in the program of Fig 6.2(a), the assignments to the variable a are inside a nested if-then-else (if-else inside another if-else). So, for a conditional nest of two, four different assignments for variable a are possible. Corresponding four nodes (or operations) are fused as a single node by LASER-compiler. At runtime, correct instruction out of four possible instructions can be provided to the PE to execute the operation from the nested conditional.

Our heuristic targets fusing nodes from different if-else paths pertaining to the conditional nest. Pairing is done with operations from the innermost if-then-else (i.e., one with highest conditional depth d). The unbalanced operations (i.e. one path has more operations than the other) are paired with a no-op. For example, in program of Fig 6.2(a), operations corresponding to variables a, b and c are fused first. Hence, $\langle a_{tt}, a_{tf} \rangle$ and $\langle b_{tt}, b_{tf} \rangle$ are fused nodes, as shown in Fig 6.4(a). Such pairing is one-to-one with operations from both the paths. In our example, innermost if-path has 3 operations compared to 2 operations inside respective else path. Hence, the unbalanced operation c_{tt} is fused with a no-op. Note that we do not need any selection among the operations from if-path and else-path so, corresponding select operations are eliminated during this DFG transformation. Once the operations of the innermost conditional are fused (i.e. $y\%i == 1$), operations from outer nests can be fused iteratively. So, operations of the conditionals with nest depth of $d - 1$ can be fused where d is the highest depth. Thus, we fuse all the operations associated with the condition $x\%i == 1$. The compiler iterates on the entire conditional nest and produces DFG with the fused nodes as shown in Fig 6.4. Mapping can be then

obtained with mapping techniques such as [31, 28]. Mapping the DFG with the fused nodes, obtained from LASER-compiler is like any other mapping with CGRAs. The fused nodes can be also routed to satisfy data-dependency and necessary values are stored in the register file [3] .

[3]In Fig 6.4(c) fused node $\langle\langle a_{tt}, a_{tf}\rangle, \langle a_o, nop\rangle\rangle$ is routed (named as a_c) and the correct value of a is also stored in a register of PE 4 for later usage.

Algorithm 6: FuseNodes (Input DFG D, Output DFG P)

1: $d \leftarrow getHighestConditionalDepth()$
2: **for** $i = d$ to 1 **do**
3: $n^i_{if} \leftarrow getLastNode(N^i_{if})$
4: $n^i_{else} \leftarrow getLastNode(N^i_{else})$
5: **while** $n^i_{if} \neq NULL$ or $n^i_{else} \neq NULL$ **do**
6: **if** $n^i_{if} \in N^i_{if}$ and $n^i_{else} \in N^i_{else}$ **then**
7: $fuse(n^i_{if}, n^i_{else})$
8: **else if** $n^i_{if} \in N^i_{if}$ and $n^i_{else} == NULL$ **then**
9: $fuse(n^i_{if}, nop)$
10: **else if** $n^i_{if} == NULL$ and $n^i_{else} \in N^i_{else}$ **then**
11: $fuse(nop, n^i_{else})$
12: **end if**
13: $n^i_{if} \leftarrow getLastRemainingNode(N^i_{if})$
14: $n^i_{else} \leftarrow getLastRemainingNode(N^i_{else})$
15: **end while**
16: **for** n^i_j such that $j = 0$ to $|N|$ **do**
17: **if** n^j_i is an eligible select operation $\in N^j_{other}, \ni input1(n^j_i), input2(n^j_i) = m_{fused} \in M_{fused}$ **then**
18: $Eliminate_{phi}(n^j_i)$
19: **end if**
20: **end for**
21: $RemoveRedundantArcs(E)$
22: $PrunePredicateArcs(E)$
23: **end for**

After obtaining the mapping for CGRA PEs, compiler generates instructions to support the execution of conditional nest. One such layout of instructions for CGRA PEs is shown in Fig 6.4(d). Instructions are grouped in particular manner so that hardware can easily issue the needed instructions based on the condition evaluated. Compiler associates k value with each of the conditional, which is simply number of the CGRA instructions associated. For example, first condition h ($x\%i == 1$) is evaluated on PE2 which is associated k_1=3 because maximum number of cycles required to execute the if-path or the else-path for h are three. If this condition is true, PEs should be given next three instructions from location 2– 4. In this case, PE2 is issued another conditional g ($y\%i == 1$). g is associated with k_2=1 as all fused nodes related to conditional g are mapped on PEs in a single cycle (time 4 in Fig 6.4(c)). So, only one instruction for each of CGRA PEs is enough to execute either if or else-path corresponding to g. If g is evaluated as false, k_2=1 instruction will be skipped at run-time. Once instructions from location 2–4 are issued, if-path corresponding to h gets over and next k_1=3 instructions are skipped, which corresponds to else-path of the outer conditional h. Then, instruction at location 8 can be executed allowing independent operations and kernel instructions executes from the location 1 again. Before the architecture can support the execution in such fashion, it is the compiler's job to associate corresponding k values with CGRA instructions and to configure the hardware correctly.

As shown in Algorithm 6, our heuristic first determines conditional with highest nest depth and pairs the nodes from both if and else paths. Pairing can proceed until there is an operation in if-path or else-path (line 5). If no such path exists or if the number of nodes in either of the paths is unbalanced, we need to fuse the nodes with no-ops (lines 8-11). Such assembling results in fused nodes after iterative pairing (lines 3-15). While forming the DFG, compiler preserves the data dependen-

cies throughout such fusing. After all operations in if-and-else paths are paired for a particular conditional (with any depth), eligible select operations are eliminated via a phi elimination. Then the redundant edges are eliminated and predicate arcs are pruned, which is shown at lines 16-22.

6.2.2 Architecture Improvements

LASER-compiler relies on Instruction Fetch Unit (IFU) support to jump to the correct instruction in the instruction memory and issue only those instructions based on the branch outcome. LASER-architecture is shown in Fig 6.5 which aids in selec-

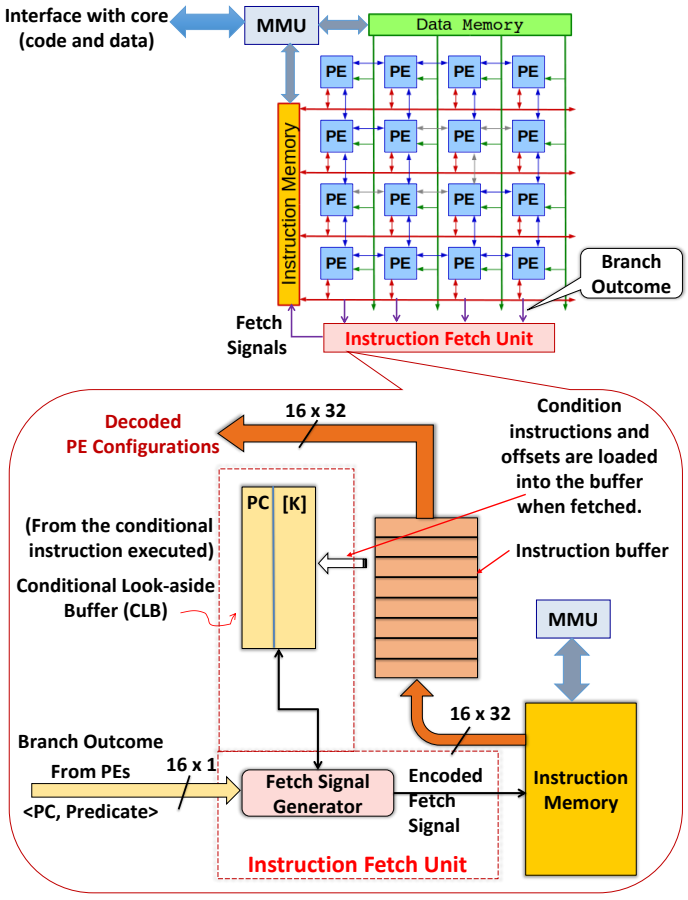


Figure 6.5: LASER-Architecture to Accelerate Complex Loops. PEs Do Not Have a Predicate Network. Branch Outcome Is Communicated to the Ifu to Issue Instructions Selectively Based on the Path Taken at Runtime.

tively issuing the instructions throughout the loop execution. The IFU keeps track of the all the conditions being evaluated in the loop. Once a PE encounters a conditional node and evaluates the outcome, it communicates that to the instruction fetch logic. Based on the information about the latest branch outcome, IFU can lookup in conditional look-aside buffer (CLB) to determine the number of instructions (k) associated with that condition. CLB keeps track of the information about PC of the conditional instruction and corresponding k value. So, if the condition evaluated is false, hardware can look-up for needed k value and IFU skips k instructions. To correctly determine the k_i value, the hardware maintains a state register which gets incremented when a new conditional is evaluated. During execution of the path for a conditional, corresponding cycle counter keeps incrementing by 1. Once the cycle counter reaches the value k_i, it means that all k_i instructions for the path of condition C_i is executed and now it should again execute the instructions from the path of the higher condition nest.

6.3 Results

We profiled MiBench and extracted 12 compute-intensive loops which are nested and/or have conditional nest. We implemented LASER-compiler in the DFG construction stage to correctly fuse the nodes of the true and false paths. LASER-compiler can be used with any mapping technique for mapping the nodes onto the CGRA. We compare LASER with partial predication scheme – only viable approach to map loops with nested conditionals. For evaluation, we used REGIMap [31] to map the DFG obtained from LASER and partial predication. PEs perform fixed-point operations with 1-cycle latency and have 4 local registers. The memory bus is shared among PEs in a row. For load and store operations, two instructions are executed, one generates the address and second loads/stores the data.

6.3.1 Performance Evaluation

Partial predication scheme requires three nodes to correctly execute an operation (true and false paths, and a selection) and increases total nodes to be mapped drastically. In Fig 6.6 the vertical axis denotes the number of nodes normalized to partial predication and the horizontal axis denotes various benchmarks. Due to fusing of nodes and elimination of select operation, LASER reduces the nodes by 43.43%. LASER achieves much better utilization with increase in depth of nested conditionals and with increase in number of operations inside the nests e.g., *susan_corner* has a depth of 24, resulting in the geomean reduction of 64%, but *gsm_2* shows very less reduction, as it has only 2 operations in a conditional.

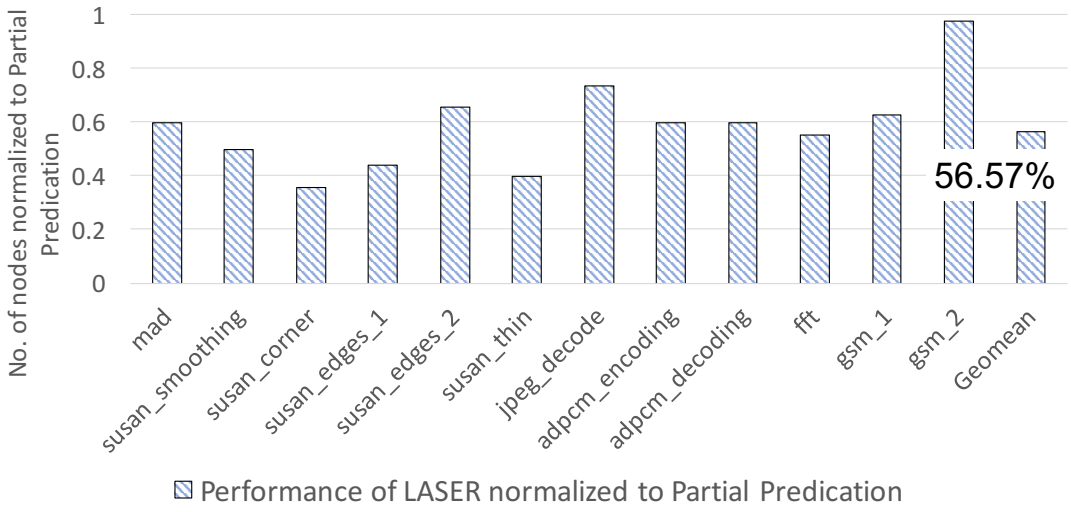


Figure 6.6: LASER Reduces Nodes by 43.43%

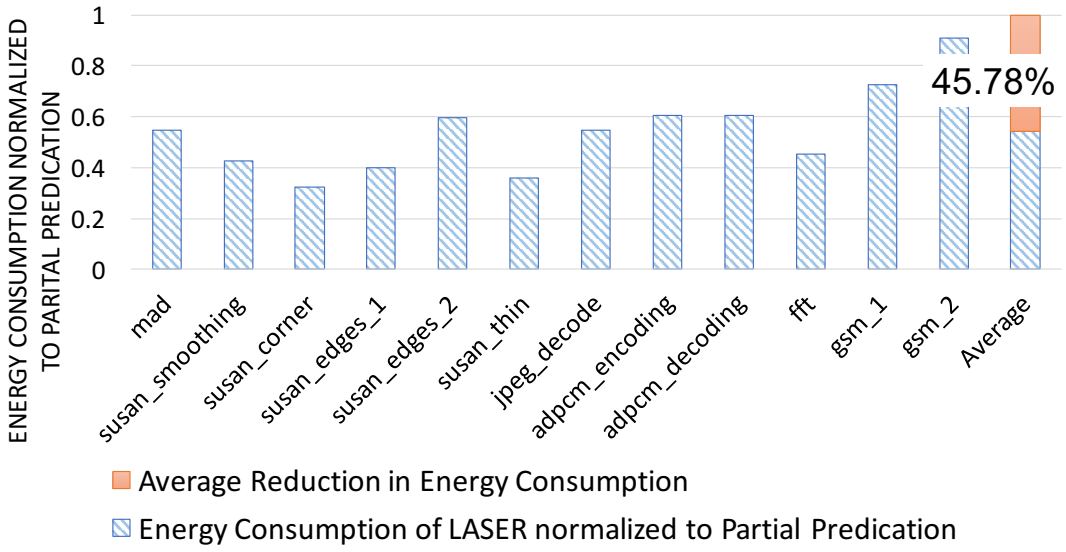


Figure 6.7: LASER Reduces Energy by 46%

We implemented the RTL model of LASER-architecture shown in Fig 6.5, and for comparison with partial predication a 4×4 CGRA with predicate network in each PE was implemented. Both the models were synthesized in 32nm using RTL compiler. The power is estimated by Cadence RTL power estimation tool. From the power numbers obtained, we estimated the energy consumed (given in [109]) by LASER and partial predication to accelerate the loops of MiBench benchmarks. Energy consumed (nJ) is given by $E = clock\_cycle \times critical\_path\_delay(ns) \times Power(W)$. Fig 6.7 shows that LASER consumes on an average 45.78% less energy compared to partial predication.

6.3.2 Scalability Analysis

Fig 6.8 shows the comparison of II achieved with partial predication and LASER for different CGRA sizes 4×4, 8×8 and 16×16. Compared to partial predication, LASER has a geomean performance improvement of 42.79% on 4×4 CGRA. As the size of CGRA increases to 8×8, the geomean II reduction for LASER was 38.05%,

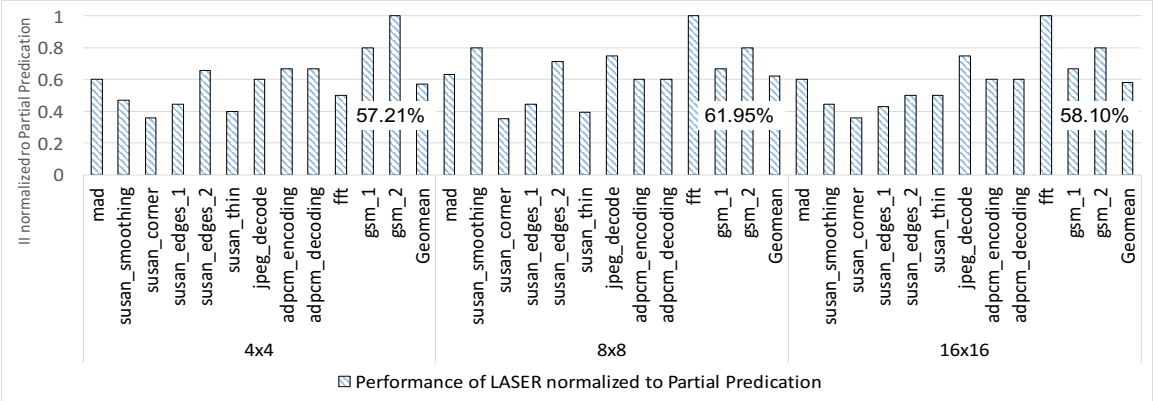


Figure 6.8: LASER Is a Scalable Solution With 40.91% Cumulative Geomean Reduction in II Compared to Partial Predication.

compared to partial predication. For 16×16 CGRA the geomean II reduction is 41.9%. LASER achieves consistent performance improvement with a cumulative geomean reduction of 40.91% across all three configurations of CGRA.

6.4 Chapter Summary

To accelerate general purpose applications with computation bottlenecks as nested loops and nested conditionals, CGRA should behave more like a general purpose modern processor with operationally enhanced IFU, to issue only the correct instruction. State-of-the-art compilers impose a high overhead to accelerate loops with only marginal performance improvement. We have presented LASER, a novel hardware-software approach where, the improved compiler fuses the nodes of various paths of the conditionals, and IFU issues selectively only correct instructions based on the branch outcome. LASER exceeds the state-of-the-art partial predication in accelerating complicated loops efficiently, with 43.43% node reduction and 40.91% better performance.

Chapter 7

CASE STUDY: SCALING UNION OF INTERSECTIONS METHOD

7.1 Background

The growth of the Internet and social media applications has paved the way for the development of highly sophisticated machine learning and statistical data analysis tools. Further scientific data collection strategies have grown exponentially over the years by innovation in the field of sensors and advanced data collection methods. Many fields such as genetics, mass spectrometry, and neuroscience [113, 114, 115, 116] now have the means of collecting big data through various devices and sensors [1]. In particular, advanced recording devices created as part of the BRAIN Initiative enable recording neural activities from hundreds to thousands of neurons for days at a time, generating TeraBytes and in some cases, PetaBytes of time series data [117, 118]. A challenge in such data sets is to infer the causal network that generated the time series data, and thus gain insight into scientific mechanisms of complex phenomena [117, 118, 115, 116]. Similarly, one may wish to understand the causal influences among companies from stock price time series [119].

Vector autoregressive (VAR) models are well suited for inference of Granger causal networks from such high-dimensional, multi-variate observational time series data. Introduced for the analysis of econometric time series, Granger causality is the amount of variance in one time series accounted for by the past of another time series [119]. Thus, from a statistical-machine learning perspective, the challenge of Granger causality is to accurately infer the existence (or not) and weight of directed edges between nodes in the network from noisy time series observations of the nodes. Although VAR

models provide a flexible framework and are probabilistically tractable [120], scaling VAR inference to massive data sets is a major challenge due to unfavorable scaling of the problem size with the number of nodes or features in the network.

The Union of Intersections (UoI) framework developed in [121] is a powerful statistical-machine learning framework which has natural algorithmic parallelism. Methods based on UoI improve the selection of features (model selection) and estimation of the contribution of the selected features (model estimation). The main mathematical innovations of UoI are 1) creating a family of potential model supports through an intersection operation for a range of regularization parameters in model selection, and 2) combining the above-computed supports with a union operation so as to increase prediction accuracy on held out data in model estimation. Theoretical and extensive numerical evaluation of a sparse linear regression algorithm based on UoI (UoI_{LASSO}) presented in [121] shows state of the art feature selection (low false positives and low false negatives) and feature estimation (low-bias, low-variance) compared with many regression algorithms (e.g., LASSO, SCAD and Ridge). This is done without formulating a non-convex optimization problem. Similarly, the statistical performance evaluation for the UoI implementation for VAR models, UoI_{VAR}, presented in [122], shows less bias and superior selection accuracy when compared to LASSO and non-convex optimization method such as the minimax convex penalty (MCP). Note that non-convex optimizations (as utilized in e.g., SCAD and MCP) are extremely challenging for implementation in the multi-nodal distributed computing paradigm [123]. In contrast, methods relying on convex optimization (e.g., UoI_{LASSO} and UoI_{VAR}) can utilize the Alternating Direction Methods of Multipliers (ADMM) [124] for solving the constrained convex optimization in a distributed manner in a multi-nodal computing environment. Thus, while our prior works establish the state-of-the-art statistical properties of UoI_{LASSO} and UoI_{VAR}, several challenges remain

in the application of these methods to large data sets.

In this section, we develop a scalable implementation of UoI_{VAR} to infer Granger causal networks from high dimensional time series data sets. UoI_{VAR} builds upon UoI_{LASSO}: thus, in order to better understand the scalability of UoI_{VAR}, we start by studying the scalability of LASSO-ADMM in UoI_{LASSO}, and then proceed to UoI_{VAR}. We reveal the computation, communication, input/output bottlenecks for UoI_{LASSO} and UoI_{VAR}, and develop solutions to mitigate them. To accommodate the randomness required for bootstrap sampling used in UoI methods, we introduce a Random Data Distribution strategy to efficiently manage data read and distribution time from large data sets. We introduce distributed Kronecker product and vectorization strategies for UoI_{VAR}. Above and beyond parallelization of optimization through ADMM, we analyze both algorithms for natural parallelism and evaluate our multi-node implementation. With high-dimensional synthetic data sets (1000 nodes or features) we demonstrate the weak and strong scaling of each algorithm. Due to the unfavorable scaling of the problem size with the number of nodes in the network ($\approx p^3$ for p nodes), it is rare to encounter VAR models with more than 50 nodes or features. In the statistical literature on high-dimensional VAR modeling, numerical experiments considered adequate to represent typical data applications are around 30 nodes, and larger-scale data applications are on the order of a few hundred nodes: [125] used simulated data of 30 nodes and [126] used monthly home-price appreciation (HPA) data set with 352 nodes. Finally, we analyze a real world neurophysiology data set of 192 neurons and a stock market data set (S&P 500 index in 2013-2014) with 470 companies. Our scaling to 1000 nodes (1M parameters) reflects an $\approx$ 3-fold increase in network scale ($\approx$ 9 fold more parameters), while doing so in the context of a superior inference algorithm.

7.2 Methods

7.2.1 Formal Statistical Description

Let us consider n samples of input data $((Y_1, X_1),\ ...,(Y_n, X_n))$ with univariate response variable Y and p-dimensional predictor variable X. The linear regression model for this input data is generated as:

$$Y = X\beta + \epsilon \tag{7.1}$$

where $Y = (Y_1, ..., Y_n), X$ is a $n \times p$ design matrix; $\epsilon = (\epsilon_1, ..., \epsilon_n)$ are random noise terms with $\epsilon \sim N(0, \sigma^2 \mathbf{I}_n)$. Let $S = \{i : \beta_i \neq 0\}$ be the non-zero coefficient set of β.

The LASSO regression algorithm with penalization parameter $\lambda > 0$ minimizes the following constrained convex optimization problem with respect to β:

$$\hat{\beta} = argmin_\beta ||Y - X\beta||^2 + \lambda||\beta||_1 \tag{7.2}$$

Here, the first term on the right-hand side penalizes the error of the predictions, while the second term penalizes the L_1 norm of the parameter vector β, setting some values of β to zero.

7.2.2 Model Selection and Model Estimation

For every bootstrap sample T^k the LASSO estimates $({}^j\hat{\beta}^k)$ are computed (here, using the Alternating Direction Method of Multipliers (ADMM)), see equation 7.6) across different regularization parameter values, λ_j. For each bootstrap sample, the support (S_j^k) are the non-zero values of the estimates calculated by LASSO-ADMM. It is known [121] that the LASSO estimator is prone to false positives for a decrease in penalizing parameter (λ): i.e., it includes more parameters than are in the model.

To mitigate this, in UoI_{LASSO} the support associated with a given λ, S_j is taken as the intersection of the supports across bootstrap samples:

$$S_j = \bigcap_{k=1}^{B_1} S_j^k \tag{7.3}$$

This is done for each value of (λ), creating a family of potential model supports $\mathbf{S} = [S_1, S_2, ..., S_q]$.

A number B_2 of bootstrap samples are used to compute the model estimates. For each potential support from the model selection step (Algorithm 7 line 18), the unbiased Ordinary Least Squares (OLS) estimator is used to estimate the associated model from each of the B_2 bootstrap samples. The algorithm then computes a *Union* of supports by averaging the OLS estimates that optimize predictions, which reduces variance and performs a union operation on the supports optimizing predictions. The variable set post-union (averaging) can be represented as (approximately):

$$S_{UoI} = \bigcup_{l=1}^{B_2} S_{jl} = \bigcup_{l=1}^{B_2} \bigcap_{k=1}^{B_1} S_{jl}^k \tag{7.4}$$

7.2.3 Distributed Constrained Convex Optimization by Alternating Direction Method of Multiplier

The core calculations in both UoI_{LASSO} and UoI_{VAR} involve solving a constrained convex optimization problem (L_1 regularized linear regression). Here, we use the Alternating Direction Method Multiplier (ADMM)[124] to minimize the loss function (Equation 7.2). LASSO-ADMM solves the dual problem in form of equation 7.5:

$$\begin{aligned} \text{minimize } & f(x) + g(z) \\ \text{subject to } & x - z = 0 \\ \text{where, } & f(x) = (1/2)||Y - X\beta||_2^2; \\ & g(z) = \lambda||\beta||_1 \end{aligned} \tag{7.5}$$

where x $\in \mathbb{R}^n$, z $\in \mathbb{R}^m$, and f and g are convex. The LASSO-ADMM algorithm consists of an x minimization, z minimization followed by a dual variable update. The separation of minimization over x and z allows for the separate decomposition of f and g. Here, x and z can be updated in sequential or alternating computations which gives the name alternating direction. In the distributed ADMM algorithm, each compute core is responsible for computation of its own objective (x) and constraint (z) variables and its quadratic term ($f(x)$) is updated so that all the cores converge to a common value of estimates. To ensure a good scalability, the ordinary least squares (OLS) is implemented using LASSO-ADMM algorithm for model estimation by setting regularization parameter λ to 0, thereby making g in equation 7.5 equal to 0.

Algorithm 7: UoI_{LASSO} $(InputData(X, y) \in \mathbb{R}^{n\times(p+1)}, \lambda \in \mathbb{R}^q,\ B_1,\ B_2)$

Model Selection
for $k = 1$ to B_1 **do**
  Generate bootstrap sample $T^k = (X_T^k, Y_T^k)$
  for $\lambda_j \in \lambda$ **do**
    Compute LASSO estimate ${}^j\hat{\beta}^k$ from T^k
    Compute support $S_j^k = \{i\}$ s.t ${}^j\hat{\beta}_i^k \neq 0$
  end for
end for
for $j = 1$ to q **do**
  Compute Bootstrap-LASSO support
  for λ_j : $S_j = \bigcap_{k=1}^{B_1} S_j^k$ (as in equation 7.3)
end for
Model Estimation
for $k = 1$ to B_2 **do**
  Generate bootstrap samples for training and evaluation:
  training $T^k = (X_T^k, Y_T^k)$
  evaluation $E^k = (X_E^k, Y_E^k)$
  for $j = 1$ to q **do**
    Compute OLS estimate $\hat{\beta}^k_{S_j}$ from T^k
    Compute loss on E^k : $L(\hat{\beta}^k_{S_j}, E^k)$
  end for
  Compute best model for each bootstrap sample:
  $\hat{\beta}_S^k = {}_{\hat{\beta}^k_{S_j}}\ L(\hat{\beta}^k_{S_j}, E^k)$
end for
Compute averaged model estimates $\hat{\beta}^* = \frac{1}{B_2}\sum_{k=1}^{B_2}\hat{\beta}_S^k$ (as in equation 7.4)
Return: $\hat{\beta}^*$

7.2.4 UoI_{LASSO} Algorithm

A high-level overview of the UoI_{LASSO} algorithm, shown in Algorithm 7, consists of two Map-Solve-Reduce steps (Figure 1). The algorithm takes multiple random

bootstrap subsamples of the input data (*Map*) and distributes it across different computing cores. Next, LASSO and OLS (*Solve*) use the distributed data and solve the convex optimization. The resultant estimates are then combined by intersection and union operations (*Reduce*). The *Reduce* step in model selection performs a feature compression by intersection operation of supports across bootstraps. The *Reduce* step in model estimation performs a feature expansion by averaging (union operation) the OLS estimates across different model supports.

Algorithm 8: UoI_{VAR} ($InputData(X_1, \ldots, X_N)^T \in \mathbb{R}^{N \times p)}$, $\lambda \in \mathbb{R}^q$, B_1, B_2)

Model Selection
for $k = 1$ to B_1 **do**
  Generate bootstrap sample $T^k = (X^k_{T1}, \ldots, X^k_{TN})$
  Construct $(\mathbf{Y}^k_T, \mathbf{X}^k_T)$ (as in equations 7.7 - 7.8)
  Construct $Y^k_T = \text{vec}\mathbf{Y}^k_T$ and $X^k_T = (\mathbf{I} \otimes \mathbf{X}^k_T)$
  for $\lambda_j \in \lambda$ **do**
    Compute LASSO estimate ${}^j\hat{\beta}^k$ from (X^k_T, Y^k_T)
    Compute support $S^k_j = \{i\}$ s.t ${}^j\hat{\beta}^k_j \neq 0$
  end for
end for
for $j = 1$ to q **do**
  for $\lambda_j : S_j = \bigcap_{k=1}^{B_1} S^k_j$ (as in equation 7.3, Compute Bootstrap-LASSO support)
end for
Model Estimation
for $k = 1$ to B_2 **do**
  training $T^k = (X^k_{T1}, \ldots, X^k_{TN})$ (Generate bootstrap samples for training and evaluation)
  evaluation $E^k = (X^k_{E1}, \ldots, X^k_{EN})$
  Construct $(\mathbf{Y}^k_T, \mathbf{X}^k_T)$ (as in equations 7.7 - 7.8)
  Construct $(\mathbf{Y}^k_E, \mathbf{X}^k_E)$ (as in equations 7.7 - 7.8)
  Construct $Y^k_T = \text{vec}\mathbf{Y}^k_T$ and $X^k_T = (\mathbf{I} \otimes \mathbf{X}^k_T)$
  Construct $Y^k_E = \text{vec}\mathbf{Y}^k_E$ and $X^k_E = (\mathbf{I} \otimes \mathbf{X}^k_E)$
  for $j = 1$ to q **do**
    $\hat{\beta^k_{S_j}}$ from T^k
    Compute loss on $E^k : L(\hat{\beta^k_{S_j}}, E^k)$(OLS estimate)
  end for
  Compute best model for each bootstrap sample:
  $\hat{\beta^k_S} = {}_{\hat{\beta^k_{S_j}}} \; L(\hat{\beta^k_{S_j}}, E^k)$
end for
Compute averaged model estimates
$\hat{\beta}^* = \frac{1}{B_2} \sum_{k=1}^{B_2} \hat{\beta^k_S}$ (as in equation 7.4)
Partition $\hat{\beta}^*$ and rearrange into $(\hat{A}_1, \ldots, \hat{A}_d)$ and $\hat{\mu}$
Return: $(\hat{A}_1, \ldots, \hat{A}_d)$ and $\hat{\mu}$

7.2.5 UoI_{VAR} Algorithm

The UoI_{LASSO} implementation can be adapted to sparse estimation of vector autoregressive model parameters from high-dimensional time series data. In this case the input data is a vector time series $\{X_t\}_{t=1}^{N}$ generated by a vector autoregressive process of order d, $VAR(d)$:

$$X_t = \sum_{j=1}^{d} A_j X_{t-j} + U_t \tag{7.6}$$

where $X_t \in \mathbb{R}^p$, the process has p-dimensional Gaussian disturbances $U_t \overset{iid}{\sim} \mathbb{N}_p(0, \Sigma)$. The stability of the process is expressed by the constraint $\mathbf{det}(I - \sum_{j=1}^{d} A_j z^j) \neq 0 \quad \forall\, |z| \leq 1$.

Equation 7.6 provides a model for the data which can be written as a multivariate least squares problem with correlated errors of the form $\mathbf{Y} = \mathbf{XB} + \mathbf{E}$. In particular, the response is the $(N-d) \times p$ matrix

$$\mathbf{Y} = (X_N, X_{N-1}, \ldots, X_{d+1})^T \tag{7.7}$$

and the regressors are lagged values represented in the $(N-d) \times (dp)$ matrix

$$\mathbf{X} = \begin{pmatrix} X'_{N-1} & X'_{N-2} & \cdots & X'_{N-d} \\ X'_{N-2} & X'_{N-3} & \cdots & X'_{N-(d+1)} \\ \vdots & \vdots & \ddots & \vdots \\ X'_{d} & X'_{d-1} & \cdots & X'_{1} \end{pmatrix} \tag{7.8}$$

and the coefficient matrix is $\mathbf{B}' = (A_1 A_2 \ldots A_d)$. One estimation strategy is to vectorize the problem as shown in equation 7.9 and apply ordinary least squares to estimate

the entries of the A_j matrices.

$$\text{vec}\mathbf{Y} = (\mathbf{I} \otimes \mathbf{X})\,\text{vec}\mathbf{B} + \text{vec}\mathbf{E} \tag{7.9}$$

Equation 7.9 then has the same form as equation 7.1. Noting this correspondence, estimation with sparsity in high-dimensional time series can be accomplished by first rearranging the multivariate least squares problem and then solving the LASSO problem (equation 7.2) for the resulting rearrangement.

The UoI implementation, shown as Algorithm 8, is consequently similar to UoI_{LASSO}, but with a bootstrap method appropriate for capturing temporal dependence in the input data (here, using a block bootstrap) and large matrix operations required to obtain a problem of the form shown in equation 7.2. Aside from these modifications, the Algorithm 8 is the same as UoI_{LASSO} Algorithm 7.

7.3 Scaling UoI_{LASSO} and UoI_{VAR}

UoI_{LASSO} and UoI_{VAR} exhibit a high degree of algorithmic parallelism. In each of the model selection and model estimation steps, the bootstrap subsamples (B_1 and B_2) can be parallelized, referred to as P_B parallelization. Additionally, parallelization over regularization parameters (λ_j) can be used (referred to as P_λ parallelization). An important point to consider is that the model selection and model estimation must occur in sequential order and cannot be parallelized.

7.3.1 Challenges in achieving parallelism

To achieve accuracy in selection and estimation, UoI-based methods utilize the notion of stability to perturbations, in this case multiple random resampling of the data. UoI_{LASSO}, in particular, requires random sub-samples generated from the data

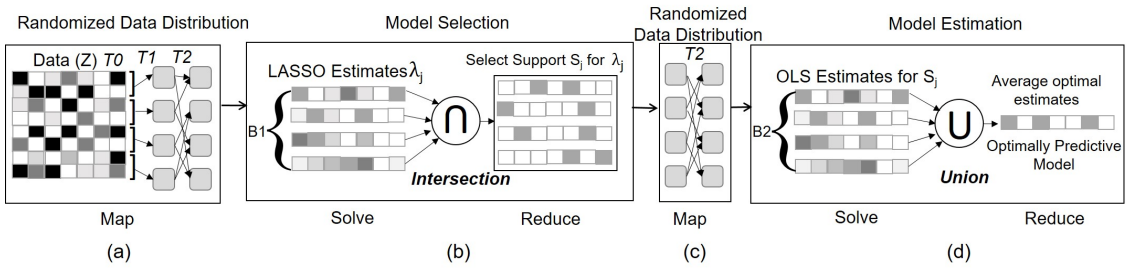


Figure 7.1: (A) a Three-tier (T0, T1 and T2) Distribution Strategy for Randomized Distribution of Data Set Across the Number of Sample from the Hdf5 Data File to the Cores of Knl. (B) Model Selection – Lasso Admm Is Used to 'solve' and **Intersection** Operation Is Used as 'reduce' to Select Family of Support S_j. (C) Data Randomization for Cross Validation Where Tier2 Random Distribution Is Employed to Randomly Reshuffle the Data. (D) Model Estimation – Ols Is Used to 'solve' and **Union** Operation Is Used to 'reduce' to Get an Optimally Predictive Model.

set in selection and estimation Map steps (Figure 1). In our initial experiments we have seen that repeated access to the data file in the file system takes a lot of data access time, and dividing the data set into chunks for faster access reduces selection and estimation accuracy [121]. Due to the smaller sized data set in UoI_{VAR} (recall that VAR problem scales $\approx p^3$), the centralized distribution strategy was adopted to distribute the data to compute cores. We found that the main challenge for UoI_{VAR} 'Map' is computing the Kronecker product and vectorization steps in a distributed method. As far as we are aware of, there has been no prior methods to distribute the Kronecker product computation and vectorization for VAR models.

7.3.2 Randomized Data Distribution Design using HDF5

We introduce randomized data distribution strategy for UoI_{LASSO} to improve the data read time from the file system and reduce the data distribution time to the compute cores. The synthetic data set matrices used in this evaluation have the "Samples" in rows and "Features" in columns. The data set size is the problem size for UoI_{LASSO}. We use `HDF5` application program interface for data input/output. `HDF5` offers parallel reading of the input file, albeit in contiguous chunks. The library

does not provide a random reading of input data without reading the file multiple times in a loop. To parallelize this operation, we introduce a novel randomized data distribution technique. First, the data is read in parallel from the input file into the computing cores in contiguous blocks. As shown in Figure 7.1, T0 or *Tier0* is the source HDF5 file. The contiguous reading by all the processes is done in T1, *Tier1*, using HDF5 hyperslabs [127]. *Tier0* and *Tier1* data distribution use an underlying HDF5-parallel library for parallel accesses and hyperslab creation. By creating hyperslabs, the application can read the data file and load them into the memory space created on each compute core. After loading the data from the input file, we employ MPI one-Sided communication to randomly distribute the subsamples (T2, *Tier2*). The input data is distributed via row-wise block-striping to distribute the samples. If N is the number of samples, p is the feature size, and B is the number of cores, each core receives $\frac{N}{B}$ rows and p columns. Each core then solves the constrained convex optimization problem using LASSO-ADMM (equation 7.5) and is responsible for computing its own objective (x) and constraint (z), and the quadratic term is updated to converge to a common value of estimates.

Since UoI_{VAR} is a time series model, the input data for this algorithm exhibit temporal dependence. To maintain this dependence, a block bootstrap approach was adopted by randomly selecting time series blocks for every bootstrap subsample. The Algorithm 8 lines 5 and 20-21, requires a column stacking vectorization step to construct Y_T^k and an identity Kronecker product step to construct X_T^k. In the serial version of the algorithm a simple vectorization and Kronecker product functions can be invoked, but in a distributed-memory parallel paradigm, this is not possible. Unlike UoI_{LASSO}, the synthetic data sets for UoI_{VAR} are relatively small (in order of MegaBytes) and the problem is created in the *Kronecker product* and *vectorization* (line 5) of Algorithm 8. The actual problem size increases in the order $\approx p^3$, where p

is the number of features. Due to the small size of the data, the T1 parallel reading layer cannot be deployed. To overcome this issue, we have developed a distributed Kronecker product and vectorization strategy using `MPI` one-sided communication with the windows created by the *n_reader* processes: a small number of processes (usually equal to the number of samples based on the availability of resources) read the data file in parallel and creates windows for MPI-One sided communication for distributed Kronecker product and vectorization.

The Kronecker product (Algorithm 8 line 5, 20, 21) $X_T^k = (\mathbf{I} \otimes \mathbf{X}_T^k)$, is an identity block diagonal matrix of $\mathbf{X}_T^k$ from equation 7.8. Similarly, vectorization of Y (Algorithm 8 line 5, 20, 21), $Y_T^k = \text{vec}\mathbf{Y}_T^k$ is from equation 7.7. Since $\mathbf{X}_T^k$ and $\mathbf{Y}_T^k$ are computed *a priori*, the cores holding these data structures create the MPI one-sided communication windows for building $(\mathbf{I} \otimes \mathbf{X}_T^k)$ and $\text{vec}\mathbf{Y}_T^k$. Since the LHS of equation 7.7 and 7.8 are the actual problem sizes (order of GBs and TBs), the communication strategy does not require explicit computation of the equation 7.7 and 7.8 on the computing cores. The main challenge is the increased communication time to create such large matrices as there are few cores (10s to 100s) holding the actual matrices to be distributed to hundreds of thousands of cores. This problem is quantitatively explained in the Weak Scaling sub-section of UoI_{VAR}, Section IV. Note that a conventional method, like computing the Knocker product and vectorization in a single core and distributing it to the other computing cores, is not possible due to the increased space to store the data and limited availability of space per node. Like UoI_{LASSO}, post Kronecker product and vectorization step, each core solves the convex optimization problem in equation 7.5 in a distributed manner.

7.4 Results

The single node and multi-node runs for this paper were conducted on Cori Knights Landing (KNL) supercomputer at NERSC. Cori KNL is a Cray XC40 supercomputer consisting of 9,688 nodes of 1.4 GHz Intel Xeon Phi processors with a single socket 68 cores per node. The aggregated memory for a single node in KNL is 16GB MCDRAM and a 96GB DDR. The UoI_{LASSO} and UoI_{VAR} algorithms were implemented in C++ using Eigen3 library [128] for linear algebra computations and Intel-MKL library [129] for BLAS operations for UoI_{LASSO} to utilize the inbuilt Single Instruction Multiple Data (`SIMD`) directives. The `MPI` framework was used for parallelization and communication between the processes supported by OpenMP multithreading with `OMP_NUM_THREADS` as four, which showed better performance. The performance analysis setup for UoI_{LASSO} and UoI_{VAR} is shown in Table I. Related to this, recently, the optimal configuration for executing neural networks (AlexNet) was calculated in [130]. Although the model shown in [130] could potentially be applied in our context by including the structure of the design matrix (e.g., columns X rows, as well as sparse vs. dense) to find a theoretically better configuration, the practical configuration depends on the realities of the hardware.

For all the evaluations in this paper, synthetic data sets ranging from 16GB to 8TB were generated for UoI_{LASSO}, and data sets that generate problem sizes of 16GB to 8TB were generated for UoI_{VAR}. The experiments were carried out in two phases, Single Node performance and optimizations, and Multi-Node scaling. The feature size for UoI_{LASSO} is kept a constant at 20,101 features across data sets to study the effect of communication. For UoI_{VAR}, the data set features range from 356 for a 128GB problem size to 1000 features for 8TB problem size and the number of samples are twice the size of the features. We evaluate the algorithms for single node

Performance Analysis	Data Size (GB)	No. of cores (UoI_{LASSO})	No. of cores (UoI_{VAR})
Single Node	16	68	68
Weak Scaling	128	4,352	2,176
	256	8,704	4,352
	512	17,408	8,704
	1024	34,816	17,408
	2048	69,632	34,816
	4096	139,264	69,632
	8192	278,528	139,264
Strong Scaling	1024	17,408	4,352
		34,816	8,704
		69,632	17,408
		139,264	34,816

Table 7.1: Performance Analysis Setup for UoI_{LASSO} and UoI_{VAR}.

performance, exploiting algorithmic parallelism and multi-node scaling experiments. Due to limited resource availability of computing resource the multi-node scaling runs were performed with no P_B and P_λ parallelism and dedicating all the cores to distributed LASSO-ADMM computation.

7.4.1 Performance and Scaling of UoI_{LASSO}

Single Node Performance

The focus of single node performance analysis is to identify the potential bottlenecks in the program and optimize them. Post optimization, the performance improvement is calculated using a performance roofline model for both the program and the architecture (Xeon Phi) on which the program is executed. A $\approx 16GB$ data set with five selection and estimation bootstrap samples ($B_1 = B_2 = 5$) and eight regularization parameters(q) were chosen for single node optimization of the implementation.

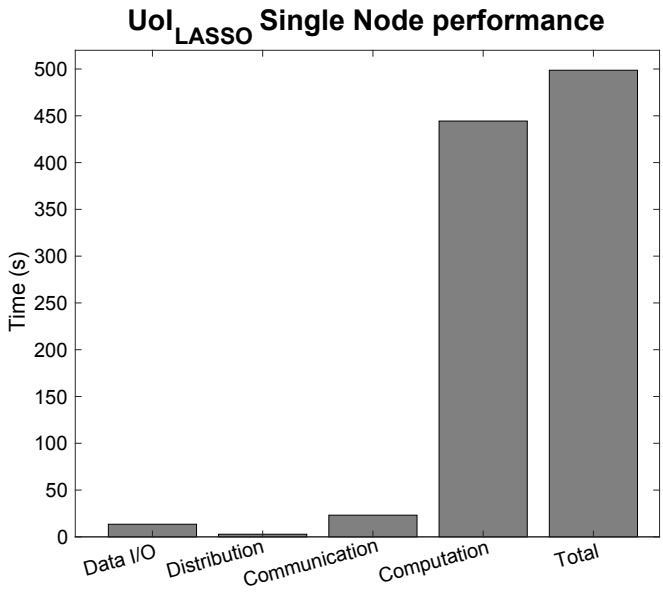


Figure 7.2: UoI_{LASSO} Runtime Number Using Intel-MKL Linear Algebra Library With $B_1 = B_2 = 5$ and $q = 8$.

Our initial analyses showed that the Matrix multiplication and Matrix-Vector product in LASSO-ADMM function were the bottlenecks. Execution of these operations showed very poor performance with native Eigen3 library on Cori KNL. To alleviate the poor performance we implemented the `BLAS` operations for matrix multiply and matrix-vector product using the Intel-MKL library. Figure 7.2 shows the runtime for single node run. Almost 90% of the runtime is dominated by computation and less than 10% by communication. All the `MPI` calls like `MPI_Bcast`, `MPI_Allreduce` etc., constitute the communication bar as shown in the Figure 7.2. More than 99% of the communication time comes from `MPI_Allreduce` call used to communicate the estimates by the distributed LASSO-ADMM function. MPI one-sided calls for distribution of the data is shown as 'Distribution', while parallel-HDF5 data loading and output saving is shown as 'Data I/O'. We analyzed the program in detail with Intel Advisor [131] tool for the performance of various sections of the code. The performance of matrix multiplication with Intel-MKL was 30.83 GFLOPS

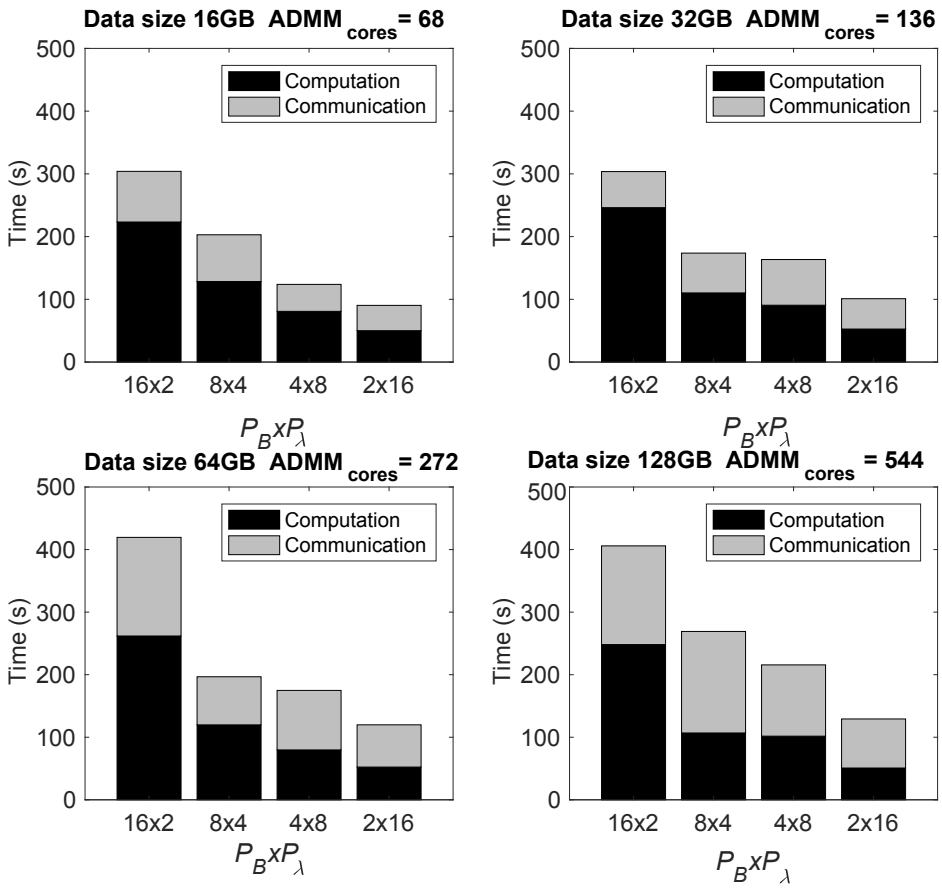


Figure 7.3: Exploiting P_B and P_λ Parallelism by Increasing the Data Set and ADMM_{Cores} by a Factor of 2.

(Giga-Floating Point operations per second) with an arithmetic intensity (Floating point operations per byte of data moved from memory) of 3.59 FLOPs/Byte and the performance of matrix-vector multiplication was 1.12 GFLOPS with an arithmetic intensity of 0.32 FLOPs/Byte. Both the `BLAS` operations were found to be DRAM memory bound. The performance of the triangular solve function used by LASSO-ADMM function for matrix decomposition was 0.011 GFLOPS with an arithmetic intensity of 0.075 FLOPs/Byte.

Exploiting Algorithmic Parallelism

The innate algorithmic parallelism exhibited by the UoI_{LASSO} was exploited by having bootstrap level (P_B), regularization parameter level (P_λ) and ADMM computation level parallelism. These runs were performed on lower end of data set spectrum, 16GB, 32GB, 64GB and 128GB with 2176, 4352, 8704 and 17,408 cores, respectively. The $P_B \times P_\lambda$ configuration used were 16×2, 8×4, 4×8 and 2×16 with $B_1 = B_2 = q = 48$ for all the runs. The data set size and the ADMM$_{cores}$ were doubled maintaining the parallelization configurations. The runtime of the different configurations are shown in Figure 7.3. Across various configurations the 2×16 has a better runtime. Also across the data set runs we can see a slight increase in the communication time for $ADMM_{cores} = 272$ and $ADMM_{cores} = 544$. This increase in the communication time was accounted by the `MPI_Allreduce` call from LASSO-ADMM implementation to collectively converge at an estimate value.

Comparison of Randomized Data Distribution Design with Conventional Distribution strategy

Conventional data distribution strategy involves reading from the data file by a single core using serial HDF5 by creating hyperslabs. The traditional methodology has three issues, namely: 1) it can read only a small chunk of data at a time, 2) it would repeatedly open the data file to read the data completely, and 3) it cannot store the loaded data due to limited space availability (aggregated memory of single KNL node is 96GB, but the data set size is in order of 100s of GBs and TBs). We implemented the conventional data distribution in C++ with serial HDF5 implementation. It should be noted that for UoI algorithms, different random bootstraps of data are required for model selection and model estimation steps (lines 3, and 14 of Algorithm 7).

Data Size (GB)	Conventional Method		Randomized Data Distr.	
	Read time (s)	Distr. time (s)	Read time (s)	Distr. time (s)
16	204.71	1.276	11.3191	0.33
128	1200.81	17.596	0.52	5.718
256	2204.52	36.46	1.46	2.62
512	5323.486	74.274	8.043	3.64
1024	11,732.48	158.016	8.781	3.774

Table 7.2: Randomized Data Distribution Design Improves the Data Read and Distribution Time Compared to Conventional Distribution Method. Beyond 1TB Data Set Size the Conventional Method's Data Read Time Crossed Beyond 5 Hours Whereas Randomized Data Distribution Read Time Was Below 100 Seconds.

The comparison of data read and distribution time between our Randomized Data Distribution Design (contribution of this paper) and the conventional design is shown in Table II. The number of cores used for runs in Table II is based on Table I. From Table II, it is quantitatively evident that the data read time and distribution time for the conventional method is a bottleneck because of the issues discussed above. From Table II, it should be noted that the read time for the 16GB is higher than the larger data sets because it was not striped into OSTs.

Multi-node Scaling

The multi-node scaling analysis is carried out for weak scaling and strong scaling of UoI_{LASSO} implementation. Parallel reading of the input file becomes an issue for multi-node scaling runs as 1000s of cores try to read the data in parallel. In an unoptimized run, the read time takes 10s of minutes which can worsen with an increase in the data size and the number of nodes. For large data sets, the HDF5 input files are stripped into different Object Storage Targets (OSTs), explained in detail; in [132]. The files are stripped for 160 OSTs to achieve a faster reading time, making the data read time of very large data sets to a few seconds.

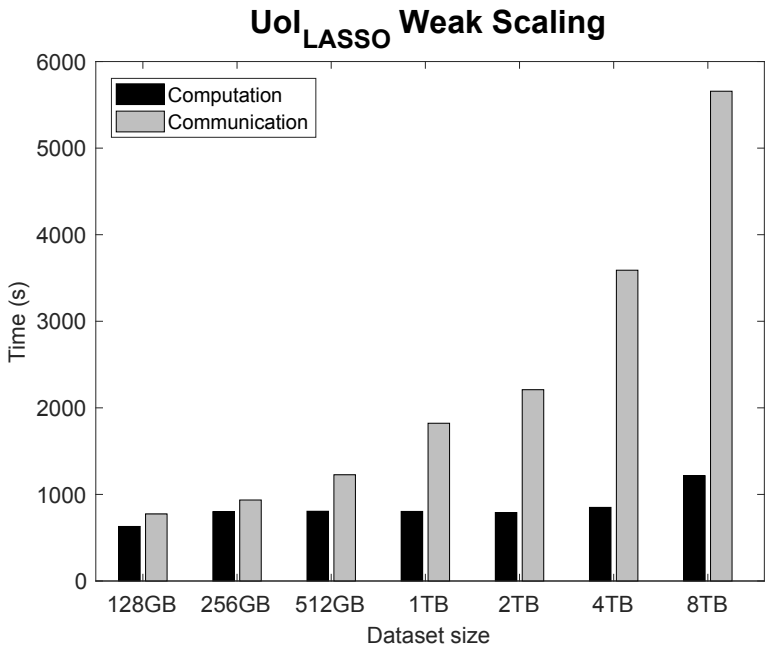


Figure 7.4: Weak Scaling Plot of UoI_{LASSO}. The Problem Size per Node Was Kept Fixed.

Weak Scaling: In weak scaling, the problem size associated with each compute core stays constant and additional computing cores are added when the size of the input data set increases. We maintain a factor of 2 for our weak scaling runs, meaning as the data set size is doubled the number of cores were also doubled (refer Table 7.1). Figure 7.4 shows the weak scaling of UoI_{LASSO}. Since matrix multiplication contributes the most to the computation time, and since the problem size per compute core is almost the same across different configurations, we find that computation exhibits nearly ideal weak scaling with slight increase for 8TB. It is seen that the communication time scales proportional to the increase in the core count. On further analysis of the communication time, we find that the `MPI_Allreduce` calls contributes almost 99% of the communication time. The error modeling of one MPI_Allreduce call for all the data points used for weak scaling as shown in Figure 7.5. The feature size of all the data sets is kept a constant at 20,101 features, so the array size for

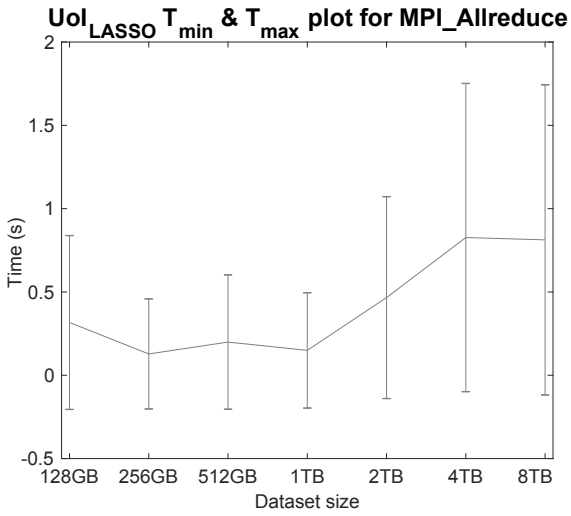


Figure 7.5: T_{min} & T_{max} Plot for UoI_{LASSO}.

MPI_Allreduce communication is uniform across all the cores. The difference in T_{max} and T_{min} for the MPI_Allreduce indicates performance variability of communications. However, despite this we observe good scalability. As future work we are evaluating non-blocking MPI and asynchronous execution models to enable further scaling.

Strong Scaling: In strong scaling, the problem size to be analyzed is kept as 1TB and the number of computing cores is increased from 17,408 to 139,264 (refer Table 7.1). Figure 7.6 shows the results of the strong scaling run. The computation time shows a decreasing trend across different configurations due to the increase in the number of cores for the same data set size. At 139,264 cores the computation goes below expected computation strong scaling trend, the reason being that the total size of the problem per core becomes small, which Intel-MKL library takes advantage of the AVX512 extensions making the matrix multiplication computed per core faster. The superlinear computation time can also be attributed to the reduced DRAM accesses due to a smaller chunk of data distributed per core. As seen in the weak scaling runs communication time increase with increasing number of cores, but beyond 69,632 cores the LASSO-ADMM converges faster making the communication

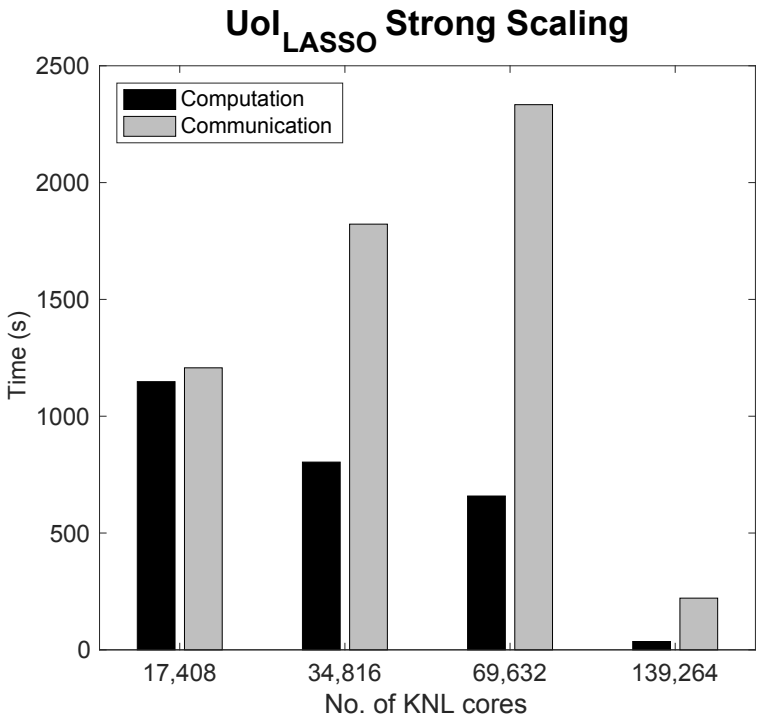


Figure 7.6: Strong Scaling Plot of UoI_{LASSO}. The Problem Size Was Kept Fixed At 1TB.

time almost equal to the ideal strong scaling.

7.4.2 Performance and Scaling of UoI_{VAR}

Single Node Performance

The Algorithm 8 creates a high dimensional matrix by Kronecker product for each bootstrap subsample. The resultant matrix has a block diagonal structure with high sparsity. From Algorithm 8, if the input data is dense the sparsity of the problem can be calculated as $1 - \frac{1}{p}$, where p is the number of features of the input data set. A problem size of ≈16GB with $B_1 = B_2 = 5$ and $q = 8$ and number of lambda parameters $q = 8$ were chosen for single node optimization. For example, if a data set has 95 features, the resultant matrix post Kronecker product has a sparsity of 98.94%. So it is intuitive to exploit this sparsity by utilizing sparse linear algebra

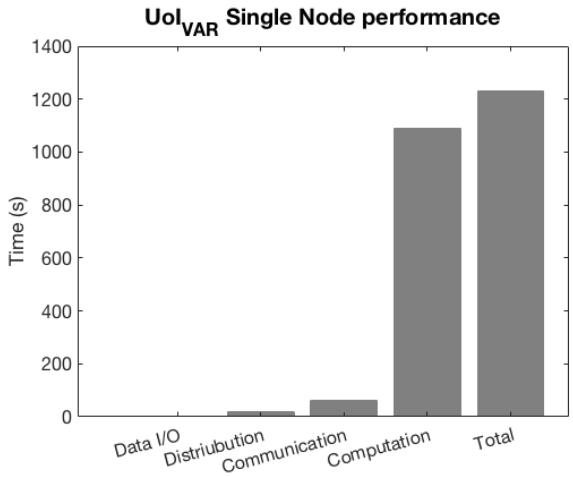


Figure 7.7: UoI_{VAR} Single Node with $B_1 = B_2 = 5$ and $q = 8$.

libraries. Figure 7.7 shows the single node run of the UoI_{VAR} implementation with Eigen3 Sparse C++ LASSO-ADMM.

Figure 7.7 shows the runtime analysis for UoI_{VAR}. Computation contributes 88% of the total runtime. Due to the problem size explosion, communication time for MPI_AllReduce can be seen to increase. The distributed Kronecker product and vectorization MPI calls are included in the distribution time constitutes more than 98% of the distribution time. UoI_{VAR} implementation was also analyzed with the Intel Advisor software for performance metrics. The performance of sparse matrix multiplication was 1.08 GFLOPS with 0.15 arithmetic intensity and the performance of matrix-vector multiplication was 2.08 GFLOPS/sec with 0.33 arithmetic intensity.

Exploiting Algorithmic Parallelism

The runs were carried out for problem set sizes of 16GB, 32GB, 64GB and 128GB. The number of ADMM$_{cores}$ were doubled with doubling the problem size. The runs were performed for $B_1 = B_2 = 32$ and $q = 16$. The computation dominates the execution time, which decreases with increases in parallelism of P_λ as shown in the Figure 7.8. It can also be noted that as the P_λ parallelism increases the Kronecker

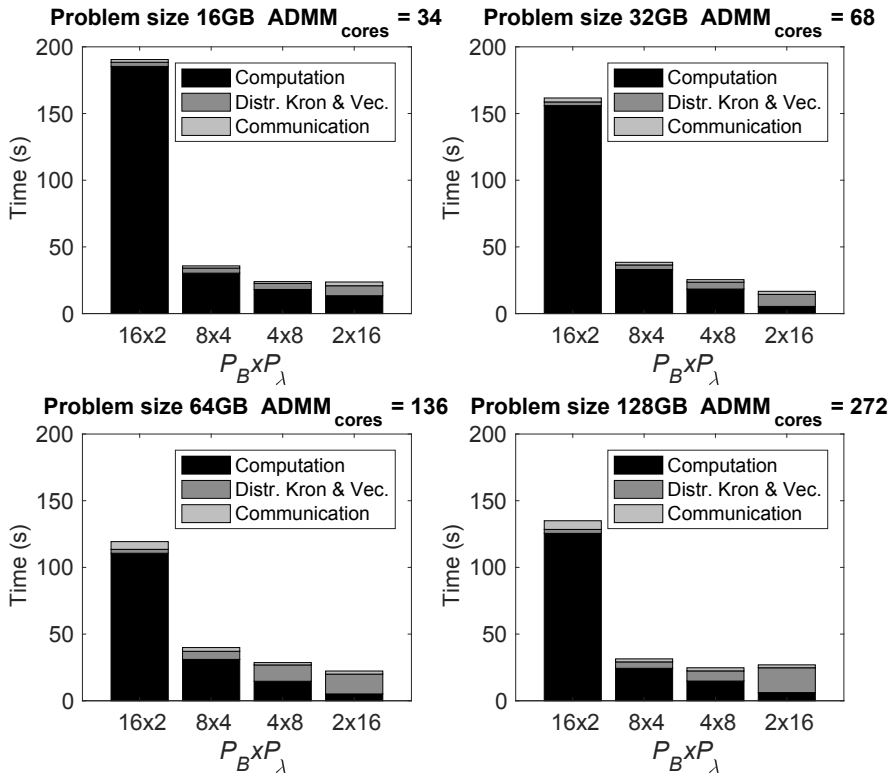


Figure 7.8: Exploiting Algorithmic Parallelism of UoI_{VAR}.

product and vectorization time increases. From Algorithm 8 (lines 5, 20 and 21) the distributed Kronecker product and vectorization is done for each bootstrap, and thus by reducing $\mathtt{P}_B$ parallelization increases the distribution time across different problem sets.

Multi-node Scaling

The data set size is very small for UoI_{VAR} compared to the problem size that is created during runtime. Unlike UoI_{LASSO} distribution strategy, only a few processes read the actual data set in parallel and the distributed Kronecker product routine builds the problem via MPI one-sided communication.

Weak Scaling: The weak scaling plot for UoI_{VAR} is shown in the Figure 7.9

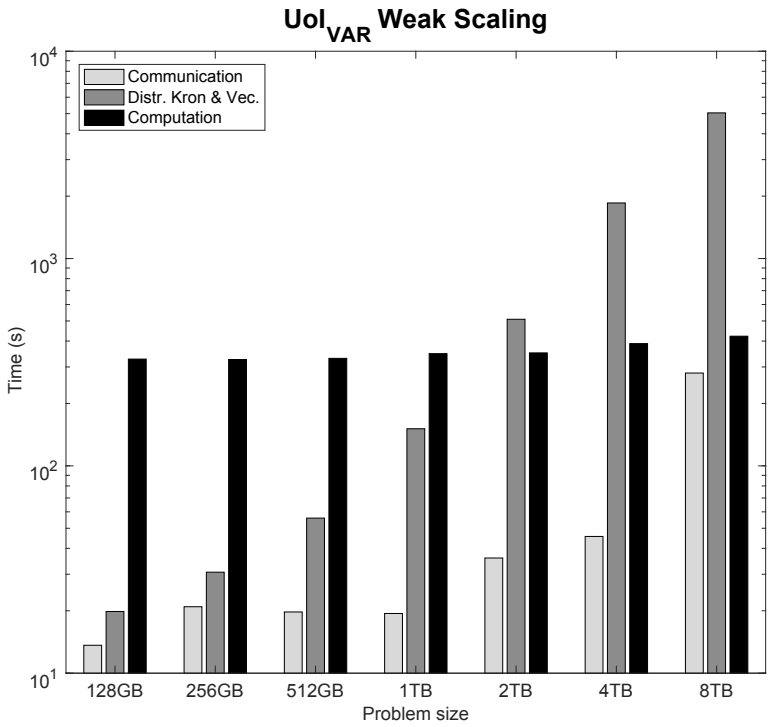


Figure 7.9: Weak Scaling Plot of UoI_{VAR} in Logarithmic Scale. The Problem Size per Node Was Kept Fixed.

for $B_1 = 30, B_2 = 20, q = 20$, with no P_B or P_λ parallelization. The Y-Axis in Figure 7.9 is given in a log-scale to show logarithmic increase in the distribution time. It can be seen that computation has almost ideal weak scaling, and the communication time also increases with increase in core count as seen in UoI_{LASSO}. The distributed Kronecker product and vectorization is proportional to the increase in the cores and problem size. One of the main reasons for this trend is the cubical increase of the problem size to the features of the input data set. Since only a few cores are responsible to read and distribute the data to thousands of computing cores during analysis, there is a communication bottleneck between the reader cores and the computing cores.

Strong Scaling: The strong scaling plot for UoI_{VAR} is shown in the Figure 7.10. Across increasing core sizes, computation time has an almost ideal strong scaling. The

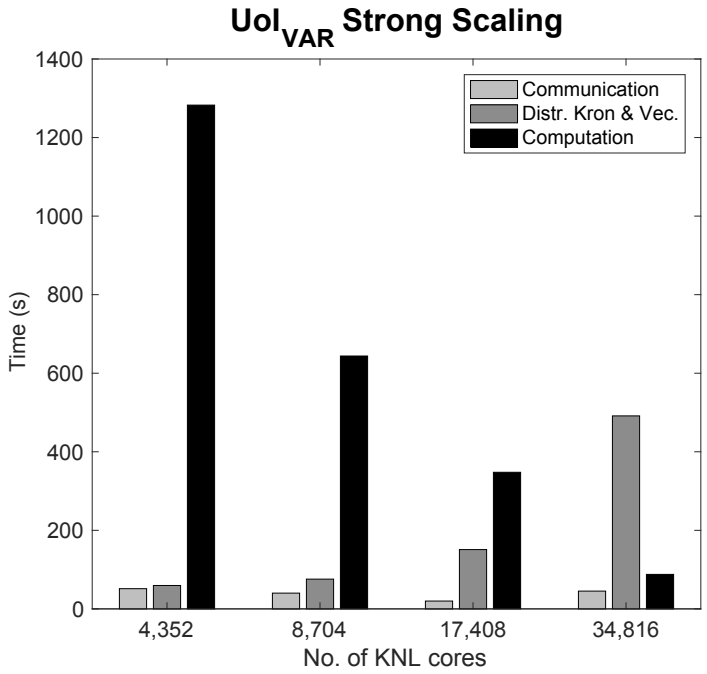


Figure 7.10: Strong Scaling Plot of UoI_{VAR}. The Problem Size Was Kept Fixed at 1TB.

reason for an ideal computation time and as discussed earlier, Sparse Eigen C++ is used to compute the matrix-vector and matrix multiplication. Even though the communication does not have an ideal scaling it minimally affects the total runtime of the program. The distributed Kronecker product and vectorization scales exponentially to the increase in the number of cores like the weak scaling.

7.5 Application of UoI_{VAR} to Real data sets

We use a financial time series data set to illustrate a data analysis using UoI_{VAR} and to illustrate computing runtime for UoI_{VAR} in a real application: the data are daily closes on the S& P Index for the years 2013 - 2018. Two different subsets are used to illustrate (i) Granger causality analysis using UoI_{VAR} and (ii) computing runtime with real data: a smaller subset was chosen for the Granger causality analysis to allow easier interpretation of the results; and a larger subset for the runtime analysis was

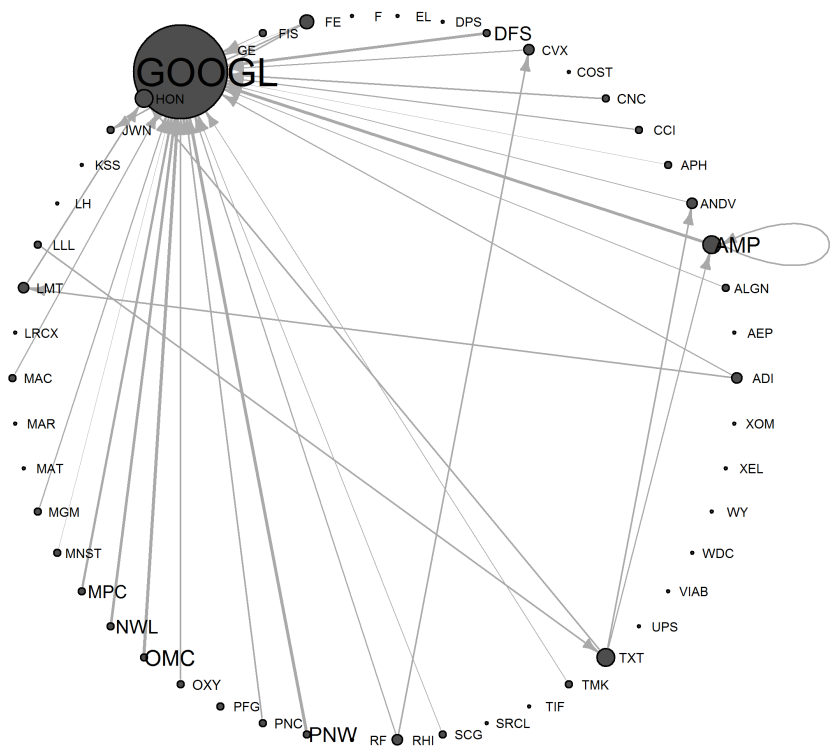


Figure 7.11: Parameter Estimates of $AVR(1)$ Model for First Differences of Weekly Closes of 50 Randomly Chosen Companies on the S& P 500 Index During 2013 and 2014.

chosen to represent compute times representative of larger-scale applications.

For the Granger causality analysis, we randomly chose 50 companies on the index in the years 2013 and 2014, aggregated the data to weekly closes, and took first differences to obtain a plausibly stationary vector time series. A first-order VAR model was then fit to the first differences using the UoI_{VAR} algorithm with hyperparameters $B_1 = 40, B_2 = 5$, selected to create a strong pressure toward sparse parameter estimates. The matrix of parameter estimates is represented in Figure 7.11 as a directed graph with nodes for each vector component (company), plotted with node sizes proportional to node degree and labeled according to company ticker, and with directed edges from node j to node i shown when the estimate of a_{ij} is nonzero, with

line thickness proportional to estimate magnitude. The result is quite sparse, with fewer than 40 edges, and suggests a complex structure of dependence of Google on a variety of other companies spanning several industry sectors. Thus, the UoI_{VAR} algorithm produces a highly interpretable output.

For the runtime analysis, we retained all 470 companies that were on the index from January 2013 through December 2016, and performed the same aggregation and differencing as in the example analysis for 195 samples. The problem size for this data set is ≈80GB, and scaling it on 2,176 cores yielded a computation time of 376.87s, and a total communication time of 4.74s. The Kronecker product and vectorization time was found to be 16.409s. In Figure 7.11 the nodes in graph are vector components and edges are nonzero parameter estimates. Our method identified very few edges thereby showing the effective dependence of Google's share price on other companies.

In addition to the financial data set, a single session non-human primate reaching task data set [133] was analyzed using UoI_{VAR} to illustrate computing runtime for a neuroscience application. Monkey reaching behavioral tasks were recorded in [133] with two monkey subjects. Some of the recorded data sets consist of spikes for both the motor cortex (M1) and, the somatosensory cortex (S1) recordings for 192 electrodes as features. The recorded spikes had 51111 samples recorded for one session. In the VAR model, the data set created a problem size ≈ 1.3TB. The problem was executed on 81,600 cores on Cori KNL. The computation and communication times were found to be 96.9s and 1598.72s, respectively. The distribution time recorded was 3034.4s.

7.6 Discussion

We found a trade-off between computation and communication in UoI_{LASSO}, shown in Figure 7.4. When the data size per core increases the computation time

increases because the computation bottlenecks are `BLAS gemm` and `gemv` operations. On the other hand for large data sets, the runtime of the code is determined by communication via MPI_Allreduce call, whereas the computation has a near ideal scaling. Almost 98% of the communication time seen in weak and strong scaling is from model selection module of the algorithm. This is due to the fact that the size of the problem solved in the model estimation module is greatly reduced relative to the model selection module. To reduce the communication runtime, P_B and P_λ parallelism can be adopted as shown in Figure 7.3 based on availability of resources.

In contrast to UoI_{LASSO}, we found a trade-off between computation and distribution in UoI_{VAR}, as shown in Figure 7.9. For smaller problem sizes computation dominates the program runtime and for larger problem sizes (especially for problem sizes 2TB and above) distribution dominates the total program runtime. The reason for this being the problem size explosion, where for a small input data size the distributed Kronecker product and vectorization creates a large matrix. One of the ways to avoid the problem is by utilizing P_B parallelism. Another way to alleviate this issue is by using communication avoiding algorithms and using local computation modules to create the matrix and then have a one-time communication to create the large matrix. With our UoI_{VAR} scaling analyses, we have implemented computations for the largest-scale VAR estimation problem (1,000 nodes, which corresponds to 1,000,000 parameters) we are aware of.

Chapter 8

CGRA COMPILATION AND SIMULATION FRAMEWORK

8.1 Background

To test and prototype the effectiveness of the CGRA compilation methods proposed so far, there has been few to no open-source CGRA simulation framework available. A simulation framework like GPGPUSim [134] or gem5-gpu [135], that is robust and versatile, can help researchers to have a hands-on experience with the functionality of CGRA and prototype the application-level performance comparisons. The CGRA Modelling and Simulation Framework (CGRA-ME) [], is a good exploratory tool but is restrictive in mapping algorithms used (supports only Simulated Annealing) and does not have a cycle-accurate CGRA simulator.

Typically, for an application that needs to accelerate on CGRA, the CGRA compiler should, (i) extract the desired loop to be accelerated, (ii) convert the loop into a Data Flow Graph (DFG), (iii) schedule the nodes of the DFG on a time extended CGRA, (iv) map the nodes of the DFG onto the processing elements (PEs) for execution, and (v) generate instructions to be executed or the CGRA and insert appropriate instructions for transferring the control between both Core and CGRA during the application execution. The scheduling and mapping honor the data dependencies, and the generated instructions should also support data dependencies in the graph to ensure the correctness of the execution.

The CGRA hardware is simple, as shown in Figure 1.1, where the instruction memory delivers instructions to the PEs at each cycle. For the example shown in Figure 1.1, 16 instructions can be fetched and executed in parallel in each cycle. The

data memory is used to store and retrieve data. Each PE also contains a register files (RFs) to store intermittent data computed during the loop execution. The PE also contains arithmetic logic unit (ALU) that executes the instructions. The computed value in each PE can be communicated to other connected PEs in the next cycle. In the following section we will present the CGRA Compilation-simulation Framework (CCF).

8.2 Overview of CCF

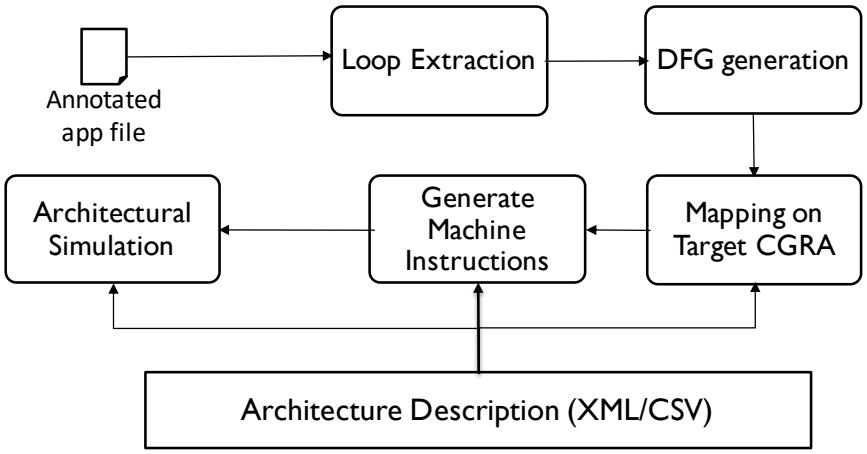


Figure 8.1: The CGRA Compilation-simulation Framework.

Figure 8.1 shows the overview of the CCF working. The CCF framework starts with the annotated application file. Since CCF strives to achieve minimal interference from the user, the user has to select the loop from the application file with a pragma (`#pragma CGRA`) to the desired loop. The user also needs to specify the CGRA configuration like the number of PEs in each row and column of the CGRA grid in the architecture description file.

The CCF uses LLVM [108] to extract the loop annotated by the user and convert the loop into a DFG. While extracting the loops, the LLVM passes also adds control-transfer functions, which transfers the function from CPU to CGRA when

the program sequence comes to the loop, and control from CGRA to CPU when the loop execution completes in CGRA. The CPU sections are compiled for ARM V7 architecture using `arm gcc` cross-compiler,

CCF contains `C++` passes to schedule and map the generated DFG, and generate instruction for the DFG mapped onto the CGRA. All these files have the user-defined architecture description file as one of the inputs.

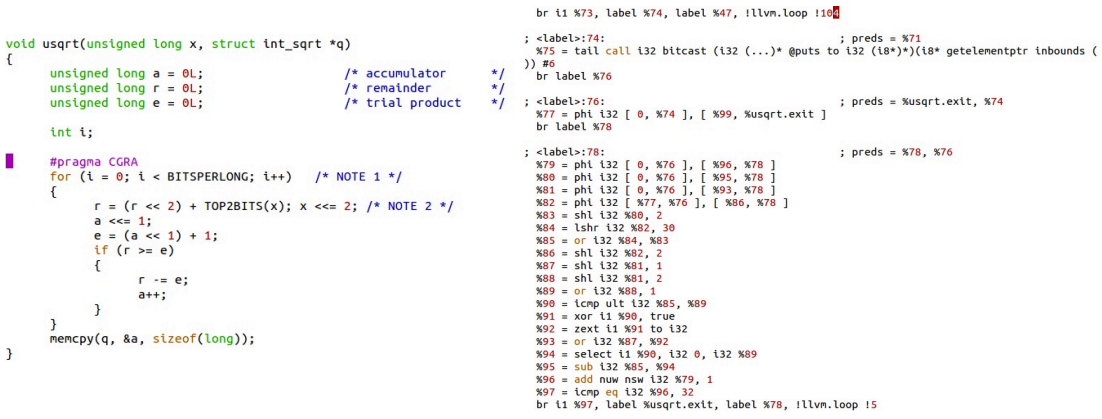
Finally, the generated instructions for CPU and CGRA are executed in the microprocessor architecture framework, gem5 [135].

8.3 LLVM Frontend

The CCF uses LLVM [108] as a frontend. The `clang` compiler converts the application program file into an intermediate representation (IR). This IR is independent of any instruction set and can be optimized thereafter. The `clang` has been modified to recognize the pragma and tag the loop IR with `CGRA.enable`. A sample loop that has been annotated with a pragma is shown in Figure 8.2a. This example shown is for a loop in the basicmath application from the MiBench benchmark suite.

The tagged-IR is then passed to the DFG creator pass developed as a transformation pass in LLVM. An example IR with `loop.llvm` is shown in Figure 8.2b. The `!5` tagged to the `llvm.loop` is further tagged to `CGRA.enable`. This DFG pass identifies the loop from the IR and creates nodes and edges based on the instruction type and data dependencies, respectively. Along with the instruction opcode, the node also stores the type of the instruction, i.e., whether it is an integer or a floating-point operation.

The partial predication [76] method for compiling if-then-else has also been implemented. The partial predication method executes both the if-path and the else-path of the control flow and selects the correct output based on the condition. Each load



```
void usqrt(unsigned long x, struct int_sqrt *q)
{
      unsigned long a = 0L;                   /* accumulator      */
      unsigned long r = 0L;                   /* remainder        */
      unsigned long e = 0L;                   /* trial product    */

      int i;

      #pragma CGRA
      for (i = 0; i < BITSPERLONG; i++)   /* NOTE 1 */
      {
            r = (r << 2) + TOP2BITS(x); x <<= 2; /* NOTE 2 */
            a <<= 1;
            e = (a << 1) + 1;
            if (r >= e)
            {
                  r -= e;
                  a++;
            }
      }
      memcpy(q, &a, sizeof(long));
}
```

```
  br i1 %73, label %74, label %47, !llvm.loop !10

; <label>:74:                                     ; preds = %71
  %75 = tail call i32 bitcast (i32 (...)* @puts to i32 (i8*)*)(i8* getelementptr inbounds (
)) #6
  br label %76

; <label>:76:                                     ; preds = %usqrt.exit, %74
  %77 = phi i32 [ 0, %74 ], [ %99, %usqrt.exit ]
  br label %78

; <label>:78:                                     ; preds = %78, %76
  %79 = phi i32 [ 0, %76 ], [ %96, %78 ]
  %80 = phi i32 [ 0, %76 ], [ %95, %78 ]
  %81 = phi i32 [ 0, %76 ], [ %93, %78 ]
  %82 = phi i32 [ %77, %76 ], [ %86, %78 ]
  %83 = shl i32 %80, 2
  %84 = lshr i32 %82, 30
  %85 = or i32 %84, %83
  %86 = shl i32 %82, 2
  %87 = shl i32 %81, 1
  %88 = shl i32 %81, 2
  %89 = or i32 %88, 1
  %90 = icmp ult i32 %85, %89
  %91 = xor i1 %90, true
  %92 = zext i1 %91 to i32
  %93 = or i32 %87, %92
  %94 = select i1 %90, i32 0, i32 %89
  %95 = sub i32 %85, %94
  %96 = add nuw nsw i32 %79, 1
  %97 = icmp eq i32 %96, 32
  br i1 %97, label %usqrt.exit, label %78, !llvm.loop !5
```

(a) (b)

Figure 8.2: (A) a Sample Loop Annotated with a Pragma in Basicmath Benchmark. (B) an Example IR of the Loop Annotated with a Pragma in Basicmath Benchmark.

and store instruction has two nodes in CCF, one for load/store address generation and one to load/store the data. Loads take two cycles to access the memory and retrieve the data from a memory location, whereas the store takes only one cycle to store the value into a memory location. The *select* operations are denoted by diamond-shaped nodes. The conditional input to the select nodes is denoted with the green edge. The loads and stores are denoted by blue nodes and the memory dependencies (edge between the load/store address nodes) is denoted as a dashed arrow. Inter-iteration dependencies, i.e., if a value of a node is required by another node of the next cycle, are denoted by red arrows with weight being the distance of the dependency.

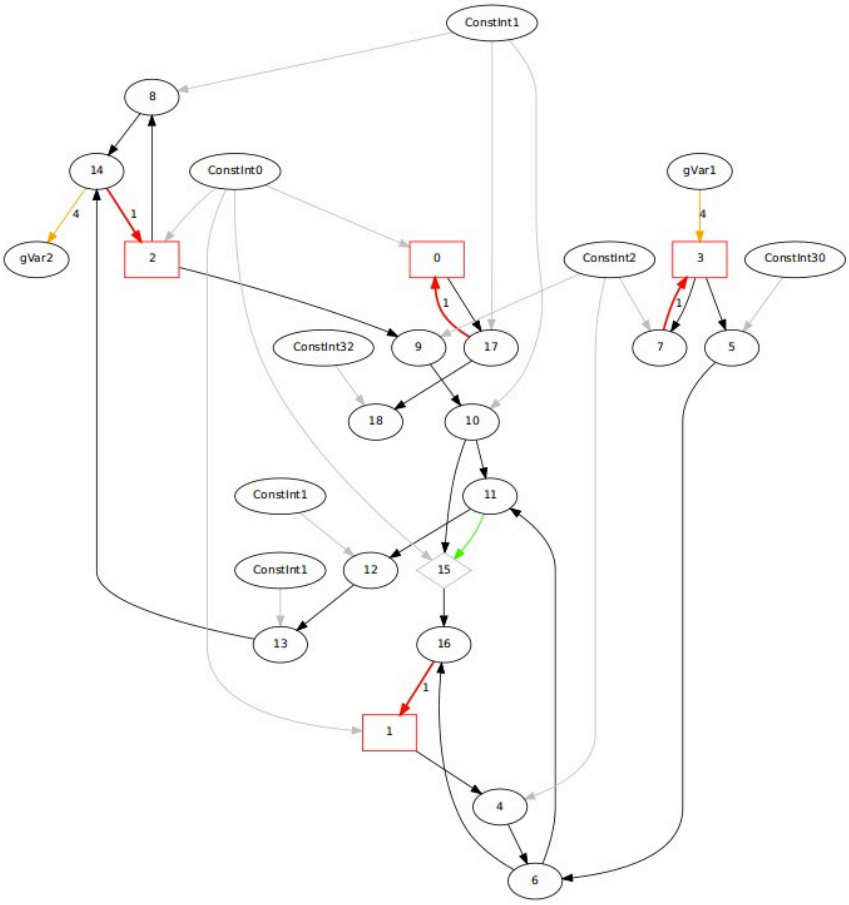


Figure 8.3: DFG of the Annotated Loop from Basicmath.

8.4 Scheduling and Mapping

In CCF, we have implemented Iterative Modulo Scheduling [34] and CRIMSON [112] for scheduling the DFG onto the CGRA architecture. The modulo scheduling algorithms allot a timeslot to the nodes of the DFG, honoring the data dependencies. Along with the scheduling algorithms we have also implemented mapping algorithms like Simulated Annealing [26], GraphMinor [30], RAMP [32], and PathSeeker (proposed in this thesis). Figure 8.4 shows the mapping of the annotated loop in basicmath on a 4×4 CGRA.

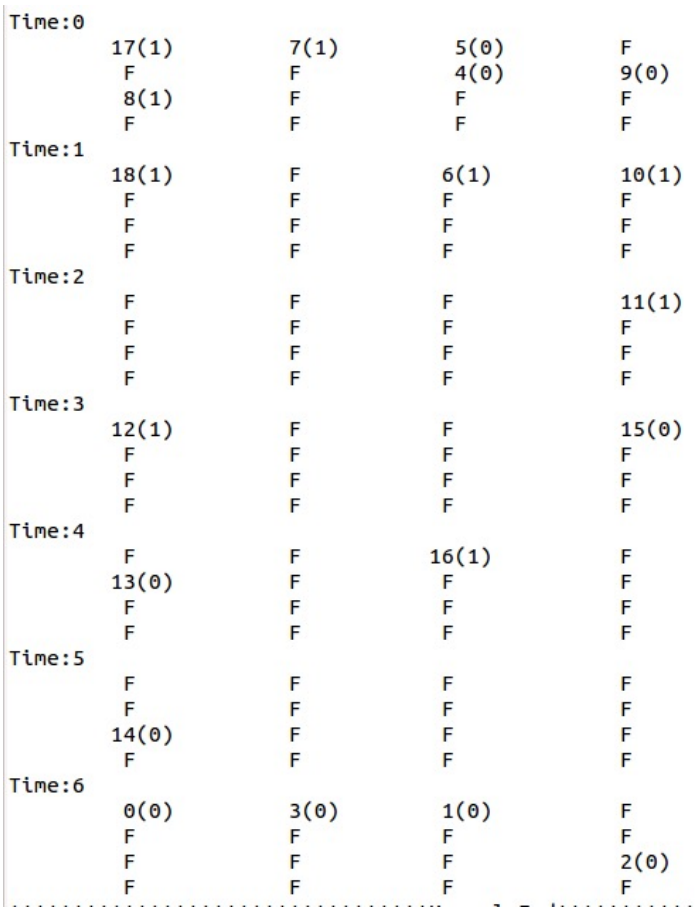


Figure 8.4: Scheduling and Mapping of the Annotated Loop on a 4×4 CGRA.

8.5 Generating CGRA Machine Instruction

8.5.1 Instruction Formats

CCF's CGRA instruction set has two formats (a) R-Type (regular instructions) and (b) P-Type (special instructions) instructions. The R-type and P-type formats are given in Table 8.1 and Table 8.2. The 27th bit, P, helps in selecting the format of the instruction. The MSB 34-32 bits in both the format denotes the datatype of the instruction. Currently, CCF supports integer and 32-bit single-precision floating-point computations. Bits 31-28 represent the opcode. The IR to CCF Virtual opcode conversion chart is shown in Table 8.4. The CCF Virtual Opcodes are used internally

by the CCF until the instruction generation state. Since CGRA ISA has only 4 bits for opcode another level of conversion from CCF Virtual Opcode to the CGRA Machine Code is required. This conversion is presented in Table 8.5.

34-32	31-28	27	26-24	23-21	20-19	18-17	16-15	14	13	12	11-0
Datatype	Opcode	P	LMux	RMux	R1	R2	RW	WE	AB	DB	Immediate

Table 8.1: R-type Instruction Format for CGRA.

34-32	31-28	27	26-24	23-21	20-19	18-17	16-15	14-12	11-0
Datatype	Opcode	P	LMux	RMux	R1	R2	RP	PMux	Immediate

Table 8.2: P-type Instruction Format for CGRA.

The R-type instruction covers all the arithmetic instructions, as shown in Table 8.5, whereas the P-type instructions cover special instructions like LDI, LDMI, LDUI, Sel, address_generator, etc. While loading large constants or addresses for loop execution, the instruction generator loads these constants as a part of the immediate of three instructions namely, LDI (Load Immediate), LDMI (Load Middle Immediate), and LDUI (Load Upper Immediate), since these values are 32-bits and the immediate bits are 12 for the instruction. These constants are stored to the RF in the PE that will be using them as a part of the initialize cycle pre-prolog. The `sel` instruction is the select instruction from partial predication that selects between two paths based on a condition.

The 26-24 and 23-21 bits are input muxes for the instruction. The input to these two muxes is from the neighboring PEs in the previous cycle. Table 8.3 shows the selection of the muxes. These muxes are chosen based on the data dependencies from the DFG.

R1 indicates the register number for input1 if LMux indicates register file as source R2 indicates the register number for input2 if RMux indicates register file as source

RW indicates the register number of register file to which result should be written WE determine whether the PE should write the result back to register file or not. AB indicates asserting address bus for the memory access. DB indicates asserting data bus for the memory access. Immediate defines the static constant value, which can be supplied to the PE. For P-type instructions, PMux indicates the input source for the predicated multiplexer of the PE, and RP indicates the register number for input3, and PMux indicates the register file as source.

#	Source
0	Register
1	Left Neighbor
2	Right Neighbor
3	Upper Neighbor
4	Bottom Neighbor
5	Data Bus
6	Immediate
7	Self (Output Latch)

Table 8.3: Input Multiplexer Selection for PEs.

Opcode #	LLVM Opcode	VOPC
0	Add	add
1	Sub	sub
2	Mul	mult
3	SDiv	div
4	Shl	shiftl
5	Ashr	shiftr
6	And	andop
7	Or	orop
8	Xor	xorop
9	ICMP_SGT	cmpSGT
10	ICMP_EQ	cmpEQ
11	ICMP_NE	cmpNEQ
12	ICMP_SLT	cmpSLT
13	ICMP_SLE	cmpSLEQ
14	ICMP_SGE	cmpSGEQ
15	ICMP_UGT	cmpUGT
16	ICMP_ULT	cmpULT
17	ICMP_ULE	cmpULEQ
18	ICMP_UGE	cmpUGEQ
19	Load	ld_add
20		ld_data
21	Store	st_add
22		st_data
23	reserved	ld_add_cond
24	reserved	ld_data_cond
25	special function	loopctrl
26	Select	cond_select
27	Special function	route
28	reserved	llvm_route
29	PHI	select
30	ConstantIntVal	constant
31	SRem	rem
32	SExt	sext
33	LShr	shiftr_logical
34	default	rest

Table 8.4: Translation of LLVm IR Opcode to CCF Virtual Opcode.

Instruction Format	Opcode #	Machine Opcode	VOPC
R-Type	0	Add	add ld_data st_data
	1	Sub	sub
	2	Mult	mult
	3	AND	andop
	4	OR	orop
	5	XOR	xorop
	6	cgraASR	shiftr
	7	NOP	-
	8	cgraASL	shiftl
	9	Div	div
	10	Rem	rem
	11	LSHR	shiftr_logical
	12	EQ	cmpEQ
	13	NEQ	cmpNEQ
	14	GT	cmpSGT cmpSGEQ cmpUGT cmpUGEQ
	15	LT	cmpSLT cmpSLEQ cmpULT cmpULEQ
P-Type	0	setConfigBoundary	-
	1	LDI	select
	2	LDMI	-
	3	LDUI	-
	4	sel	cond_select
	5	loopexit	loopctrl
	6	address_generator	ld_add st_add
	7	NOP	-
	8	signExtend	sext
	9-15	Reserved	-

Table 8.5: Translation of CCF Virtual Opcode to CCF Machine Opcode.

8.5.2 Managing Live-in and Live-out constants

Live-in values are input to the loop execution, for example, a dynamic iteration count or a starting address of array access. These live-in values can be a constant value or an address. The CCF's instruction generator (IG) records the PE to which the nodes using these live-in values are mapped. IG then loads the live-in values to RF in the same PE using LDI, LDUI, and LDMI instructions as a part of initializing cycle before the beginning of the loop execution on CGRA (prologue). Large constants that cannot be passed as a part of the immediate field are similarly stored in the RF.

Live-out variables are loop output values. These values should be passed to the CPU as these values can be used by the trailing CPU codes. As a part of the LLVM pass, we identify the basicblock in which the live-out variables are used after the loop and load those values appropriately to ensure correct execution. As a part of CGRA loop execution, the instruction generator identifies the live-out value that needs to be stored, the PE it is mapped. The address to which the value needs to be stored is loaded to the RF before the prologue execution. After the loop execution (after the epilogue) instruction generator generates a store address and stores value instructions to store the value to the appropriate memory address. The store-address instruction is mapped to the PE in which the address is loaded. The store-data instruction is executed on the PE in which the value to be stored is computed.

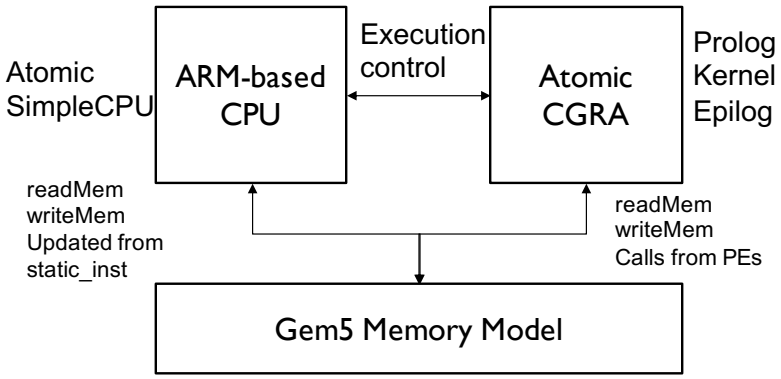


Figure 8.5: CPU+CGRA Model Based on Gem5's Atomic Timing Model.

An ARM core along with a CGRA has been implemented as a part of gem5 [135]'s CPU framework. The CPU codes are compiled for the ARMv7 processor and are executed as a part of gem5's ARM core. The instructions are divided into pre-prolog, prolog, kernel, and epilog. These configurations are stored as a part of the CGRA instruction memory space. When the program enters the loops portion execution, the CPU triggers the CGRA. CGRA starts fetching the pre-prolog instructions from the CGRA instruction space. The CGRA model implementation is shown in Figure 8.5.

The CPU core is based on an ARMV7 processor implemented in the atomic version of gem5. After the CPU execution when the program flow is to execute the loop, the control is transferred to CGRA, where the CGRA executes the prolog, kernel, and epilog of the loop. After the completion of the epilog, the CGRA transfers the execution control to the CPU to execute the remaining non-accelerated sections. The execution of CPU+CGRA is performed using `syscall emulation`, SE mode, in gem5.

The gem5-CGRA implementation has configurable CGRA X-dimension and Y-dimension, and the execution directly accesses the user-defined architecture file. This setup ensures a push-button solution once the user has annotated the application and

included the configuration file denoting the CGRA dimensions. The gem5-CGRA models, PEs with integer ALU's and single-precision floating points ALU's in each PE. Each PE contains an integer register file and a floating-point register file to aid both computation models. Each PE also has an integer output latch and a floating-point output latch to aid a complete floating computation.

8.7 Challenges in Framework Development

8.7.1 Challenges is LLVM framework

Traditionally, LLVM has been used to analyze and transform the sections of the program like a function, loop, or module. In the CCF framework we have used LLVM to:

- Identify the loop using `pragma CGRA`, from clang.
- Convert the loop section into a Data flow graph.
- Remove the loop from the execution and insert appropriate calling functions to execute loop on the CGRA.

Pragmas are inserted by the user (`pragma CGRA`), for the loop to be executed on CGRA. This was a challenging task as the pragma assertion and analysis is performed by `clang`. If pragmas are not inserted, then all the loops in a program are converted into a Data Flow Graph (DFG) as potential candidates for execution on CGRA due to `runOnLoop` function in LLVM. We modified the clang source code in LLVM to tag the pragma to the loop as a metadata. A simple metadata comparison for the CGRA tag in `runOnLoop` will help us eliminate the loops that are not tagged with pragma.

With `-O0` optimization the IR generated for the program are verbose and unoptimized. The loops contain specific basicblock headers for if and else statements for

loops with conditionals. When migrating from `-O0` to `-O3`, to apply all the compiler optimizations, we lose the header information. We currently analyze the branch instructions in each basicblock in the loop to identify the dependency and create additional *select* instructions to compute the nested if-else. We also create the proper data dependencies between the *compare* and *select* instructions for correct execution. Without this analysis loop with nested conditionals will not execute correctly.

For execution of wide variety of benchmark loops floating point datatype support is required. This was very challenging due to the ISA limitation from previous versions of CCF. We introduced a datatype analysis in DFG generation pass and created a datatype attribute to each node. This also includes the explicit typecasting, i.e., while loading a floating point data from memory location, the 32-bit address computation is `integer` whereas the value loaded from that location is a `floating point`.

For supporting the LiveIn and LiveOut variables of the loop, we designed the global variable (gVar) for data accesses and global pointer (gPtr) for array and pointer accesses. The gVar and gPtr are global address accessible through symbol table in ELF. We perform a data dependency analysis on the LiveIn and LiveOut variables usage outside the loop basicblocks. For LiveIn variables we perform explicit stores of the usage into the global variables/pointers for it be accessible by CGRA. For LiveOut variables, CGRA epilog stores the computed values in the global variables/pointers, which is later loaded into appropriate registers before its usage in the consecutive basicblock (basicblocks after the loop execution).

Deleting the loop and inserting the correct functions for CGRA loop execution are paramount for correct program execution. After converting the loop into the DFG, the mapping step modulo schedules the DFG onto the CGRA architecture producing Prolog, Kernel, and Epilog instructions for the loop. The loop in the main program should be deleted and appropriate functions should be added to transfer the execution

flow from CPU to CGRA, and execute the loop on CGRA. We add two additional functions for this support. `CreateCGRA` – This function initializes the appropriate address spaces for the prolog, kernel, and epilog execution. `accelerateOnCGRA` – This function first analyzes analyzes the instructions in prolog, kernel, and epilog and sets the address spaces for the same. Then this function transfers the control from CPU to CGRA. From the LLVM `runOnLoop` we need to delete the loop in the LLVM IR and insert hooks to the above explained two functions.

Inlined functions pose a challenge in extracting the correct loop. The issue is when a loop inside a function (other than main) is pragma tagged by the user and the function is called from the main function, the LLVM IR generates two loop instances, one inside the functional call and the other inside the main (due to inlining). Both the loops are tagged with the pragma and will be converted into DFG by the transformation pass. For example, let us consider a simple loop (with CGRA pragma) to compute 2^n where n is the user input value. This loop is inside a function $intpower(intn)$, and the main function calls the $intout = power(7)$. LLVM inlines the power function in main. So there are two loop instances in the LLVM IR, one for the loop in power function basicblock and the other for the loop in the main function. If we perform the DFG transformation first for the IR, there are two DFG sets L1 (DFG for loop in power function) and L2 (DFG for loop in main function) created. Then performing loop deletion from the IR deletes the loops in the main function, but should call L2 for correct execution. L1 loop should not be called from the program execution as it is not reachable from execution. Calling L1 inside the main will not properly compute the LiveIn and LiveOut addresses, which will result in incorrect results from the program. To overcome this issue, we integrated the DFG generation and the loop deletion into a single pass. There are two advantages with this methodology, (1) since we are analyzing the LLVM IR in a sequential fashion,

we know which loop is currently being converted into a DFG, and (2) based on the execution flow, we can call only one loop for the correct execution.

8.7.2 Scheduling, Mapping, and Generating Instructions

The schedule and mapping of the DFG nodes, i.e., the timeslot in which a node should be schedule and the PE in which the node execute is determined by the scheduling and mapping algorithm. Since the instruction generation program is common to the CCF workflow, integrating various mapping algorithms to map the DFG and create correct intermediate files was a major issue. We have implemented four popular mapping algorithms, defined by the user at compile time from `CGRA_config.txt` file, which produces correct intermediate file for CGRA instruction generation.

The previous version of the instruction generator was restrictive due to CGRA ISA width of 32-bits. For executing of floating point operations, we expanded the ISA to 35 bits, with the 3 MSB bits denoting the datatype. We have added additional support for P-type and R-type instructions. The improved 35-bits CGRA ISA instructions are formatted as `unsigned long` type for proper reading from gem5.

8.7.3 Challenges in gem5 Development

The program to be executed after the LLVM transformation and instruction generation is compiled using `arm-cross-compiler` and executed on `AtomicCGRA`. AtomicCGRA simulates the execution of the program on an ARMv7 like core and a CGRA (as coprocessor). A support was added to incorporate more CGRA sizes. We modified the CGRA execution flow to parameterize CGRA dimensions as variables and multiple CGRA sizes were added to `AtomicCGRA.py`. Additional CGRA sizes can be added by the users easily.

Our CGRA design has rowwise memory buses. The load address nodes mapped

to a row of CGRA triggers the load access by setting the address bus. The data, accessed by the `readMem` function in gem5, is made available in the data bus of that row so that load data node can access that value. The same functionality holds good for `writeMem` function except the store data function stores the value via the data bus of the row. We improved the readMem and writeMem accesses rowwise by including status registers and allocating correct memory packets. An improvement to load the floating point variable with explicit typecasting enables the proper loading of the values.

Floating point execution unit with corresponding MUXes, output, and input wires were added to each PE. CGRA instructions generated for floating point computation are executed in the floating point execution unit. This added an additional level of complexity in loading floating point constants. Integer constant values until the value 2^{12} could be sent as a part of instruction, because of 12-bits immediate support. Integer constant greater than 2^{12} are converted into 32 bit constants and loaded into the registers of PE that is using this value with three instructions, LDi, LDMi, and LDUi during the *preprolog* execution. LD-type CGRA instructions split the 32-bit immediate values into 3 sets of 12 immediate each and accumulate them into an integer register file in the corresponding PE. For floating point immediate value, we cannot pass them via the immediate bits in the CGRA instructions. We implemented a IEEE-754 single precision format and converted the floating point numbers into sign, exponent, and mastissa. The sign, exponent, and the mantissa are then passed as LDi, LDMi, LDUi and then recreated as a floating point number into a floating pointer register. This enables correct execution of instructions using floating point immediate values.

Arm-cross-compiler is used to compile the source program along with `cgra.c` file, which includes the additional functions (`CreateCGRA` and `accelerateOnCGRA`) to

execute the loop. There is a size mismatch between the instruction generation, arm-cross-compiler, and gem5 source code for `unsigned long`, which is the format of the CGRA instructions. The `unsigned long` for instruction generation and gem5 source files are 8-bytes, whereas arm-cross-compiler has `unsigned long` as 4-bytes size (This happens for the current version of CCF. Please refer to the version document in CCF GitHub). This mismatch leads to CGRA instruction not properly read and interpreted by gem5. To tackle this problem the two functions in `cgra.c` has the instructions in `unsigned long long` that is 8-bytes.

8.8 Discussion

We have presented a comprehensive CGRA compilation simulation framework in this chapter. The following are some remarks on CCF:

- LLVM based loop extraction and Data-Flow graph conversion, along with partial predication to compile arbitrary level if-then-else in the loop.
- Two scheduling algorithms and four mapping algorithms implemented.
- Libraries in CCF can easily be inherited to implement new mapping algorithms by the user. CCF can enhance and accelerate CGRA research.
- Comprehensive CGRA ISA that covers almost all of the opcodes from LLVM. Easily customizable ISA in CCF's instruction generator.
- Gem5 based Core+CGRA architecture model. Easily scalable with precompiled CGRA configurations.

REFERENCES

[1] National Research Council et al. *Frontiers in massive data analysis*. National Academies Press, 2013.

[2] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B Zdonik. Tupleware:" big" data, big analytics, small clusters. In *CIDR*, 2015.

[3] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[4] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[5] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[6] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[7] Deep learning accelerators. `\\https://en.wikipedia.org/wiki/Deep\_learning\_processor`.

[8] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. Nero: A near high-bandwidth memory stencil accelerator for weather prediction modeling, 2020. `https://arxiv.org/pdf/2009.08241.pdf`.

[9] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. Soda: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[10] H. E. Yantır, A. M. Eltawil, and K. N. Salama. Efficient acceleration of stencil applications through in-memory computing. 6(11):622. `https://doi.org/10.3390/mi11060622`.

[11] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Soda: A high-performance dsp architecture for software-defined radio. *IEEE Micro*, 27(1):114–123, 2007.

[12] Ralf Karrenberg. *Automatic SIMD vectorization of SSA-based control flow graphs*. Springer, 2015.

[13] Thomas Schaub, Simon Moll, Ralf Karrenberg, and Sebastian Hack. The impact of the simd width on control-flow and memory divergence. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–25, 2015.

[14] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[15] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, 28(4):13–27, 2008.

[16] Cedric Nugteren. *Improving the programmability of GPU architectures*. PhD thesis, Citeseer, 2014.

[17] Samuel H Fuller and Lynette I Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[18] Alexander S van Amesfoort, Ana Lucia Varbanescu, Henk J Sips, and Rob V Van Nieuwpoort. Evaluating multi-core platforms for hpc data-intensive kernels. In *Proceedings of the 6th ACM conference on Computing frontiers*, pages 207–216, 2009.

[19] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 14(2):1–16, 2018.

[20] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[21] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[22] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[23] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 66–81. Springer, 2008.

[24] Chris Nicol. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. *Wave Computing White Paper*, 2017.

[25] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: using epimorphism to map applications on cgras. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1284–1291. ACM, 2012.

[26] B Mei, M Berekovic, and JY Mignolet. Adres & dresc: Architecture and compiler for coarse-grain reconfigurable processors. In *Fine-and coarse-grain reconfigurable computing*, pages 255–297. Springer, 2007.

[27] Hyunchul Park, Kevin Fan, Manjunath Kudlur, and Scott Mahlke. Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 136–146. ACM, 2006.

[28] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[29] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *ACM Sigplan Notices*, volume 44, pages 21–30. ACM, 2009.

[30] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21, 2014.

[31] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. REGIMap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proceedings of the 50th Annual Design Automation Conference*, page 18. ACM, 2013.

[32] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*, 2018.

[33] S Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pages 184–189. IEEE, 2017.

[34] B Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1), 1996.

[35] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, and Yunheung Paek. A graph drawing based spatial mapping algorithm for coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(11):1565–1578, 2009.

[36] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ureca: A compiler solution to manage unified register file for cgras. In *Proceedings of the 21st International Conference on Design Automation and Test in Europe (DATE)*, 2018.

[37] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, 49(5):465–481, 2000.

[38] Ethan Mirsky, Andre DeHon, et al. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.

[39] Carl Ebeling, Darren C Cronquist, and Paul Franklin. Rapid—reconfigurable pipelined datapath. In *International Workshop on Field Programmable Logic and Applications*, pages 126–135. Springer, 1996.

[40] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. Hsra: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 125–134. ACM, 1999.

[41] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 329–334. IEEE, 2012.

[42] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[43] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Branch-aware loop mapping on cgras. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.

[44] Jonghee W Yoon, Aviral Shrivastava, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, and Yunheung Paek. Spkm: A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 776–782. IEEE Computer Society Press, 2008.

[45] Min Zhu, Leibo Liu, Shouyi Yin, Yansheng Wang, Wenjie Wang, and Shaojun Wei. A reconfigurable multi-processor soc for media applications. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 2011–2014. IEEE, 2010.

[46] Minwook Ahn, Jonghee W Yoon, Yunheung Paek, Yoonjin Kim, Mary Kiemb, and Kiyoung Choi. A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 363–368. European Design and Automation Association, 2006.

[47] Jong-eun Lee, Kiyoung Choi, and Nikil D Dutt. An algorithm for mapping loops onto coarse-grained reconfigurable architectures. In *ACM Sigplan Notices*, volume 38, pages 183–188. ACM, 2003.

[48] Jongeun Lee, Kiyoung Choi, and Nikil D Dutt. Compilation approach for coarse-grained reconfigurable architectures. 2003.

[49] Toan X Mai and Jongeun Lee. Efficient software-based runtime binary translation for coarse-grained reconfigurable architectures. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 132–140. IEEE, 2014.

[50] Nikhil Bansal, Sumit Gupta, Nikil Dutt, Alex Nicolau, and Rajesh Gupta. Interconnect-aware mapping of applications to coarse-grain reconfigurable architectures. In *Field Programmable Logic and Application*, pages 891–899. Springer, 2004.

[51] Akira Hatanaka and Nader Bagherzadeh. A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[52] Giovanni Ansaloni, Kazuyuki Tanimura, Laura Pozzi, and Nikil Dutt. Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(12):1803–1816, 2012.

[53] Giovanni Ansaloni, Laura Pozzi, Kazuyuki Tanimura, and Nikil Dutt. Slack-aware scheduling on coarse grained reconfigurable arrays. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–4. IEEE, 2011.

[54] Ganghee Lee, Kyungwook Chang, and Kiyoung Choi. Automatic mapping of control-intensive kernels onto coarse-grained reconfigurable array architecture with speculative execution. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–4. IEEE, 2010.

[55] Ganghee Lee, Kiyoung Choi, and Nikil D Dutt. Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(5):637–650, 2011.

[56] Jong Kyung Paek, Kiyoung Choi, and Jongeun Lee. Binary acceleration using coarse-grained reconfigurable architecture. *ACM SIGARCH Computer Architecture News*, 38(4):33–39, 2010.

[57] S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[58] Shouyi Yin, Chongyong Yin, Leibo Liu, Min Zhu, and Shaojun Wei. Configuration context reduction for coarse-grained reconfigurable architecture. *IEICE TRANSACTIONS on Information and Systems*, 95(2):335–344, 2012.

[59] ShouYi Yin, ShengJia Shao, LeiBo Liu, and ShaoJun Wei. Mapreduce inspired loop mapping for coarse-grained reconfigurable architecture. *Science China Information Sciences*, 57(12):1–14, 2014.

[60] Gregory Dimitroulakos, Michalis D Galanis, and Costas E Goutis. Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In *Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on*, pages 161–168. IEEE, 2005.

[61] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W Yoon, Doosan Cho, and Yunheung Paek. High throughput data mapping for coarse-grained reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(11):1599–1609, 2011.

[62] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(4):42, 2011.

[63] Shouyi Yin, Xianqing Yao, Dajiang Liu, Leibo Liu, and Shaojun Wei. Memory-aware loop mapping on coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1895–1908, 2016.

[64] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *ACM Sigplan Notices*, volume 43, pages 151–160. ACM, 2008.

[65] Yongjoo Kim, Jongeun Lee, Toan X Mai, and Yunheung Paek. Improving performance of nested loops on reconfigurable array processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):32, 2012.

[66] Jongeun Lee, Seongseok Seo, Hongsik Lee, and Hyeon Uk Sim. Flattening-based mapping of imperfect loop nests for cgras? In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[67] Hyeonuk Sim, Hongsik Lee, Seongseok Seo, and Jongeun Lee. Mapping imperfect loops to coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(7):1092–1104, 2016.

[68] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8. IEEE, 2013.

[69] Shouyi Yin, Dajiang Liu, Leibo Liu, Shaojun Wei, and Yike Guo. Joint affine transformation and loop pipelining for mapping nested loop on cgras. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 115–120. EDA Consortium, 2015.

[70] Shouyi Yin, Dajiang Liu, Yu Peng, Leibo Liu, and Shaojun Wei. Improving nested loop pipelining on coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):507–520, 2016.

[71] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Affine transformations for communication and reconfiguration optimization of loops on cgras. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 2541–2544. IEEE, 2013.

[72] Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. Dnestmap: mapping deeply-nested loops on ultra-low power cgras. In *Proceedings of the 55th Annual Design Automation Conference*, page 129. ACM, 2018.

[73] Kyuseung Han, Kiyoung Choi, and Jongeun Lee. Compiling control-intensive loops for cgras with state-based full predication. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1579–1582. EDA Consortium, 2013.

[74] Jihyun Ryoo, Kyuseung Han, and Kiyoung Choi. Leveraging parallelism in the presence of control flow on cgras. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 285–291. IEEE, 2014.

[75] Kyungwook Chang and Kiyoung Choi. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. In *SoC Design Conference, 2008. ISOCC'08. International*, volume 1, pages I–362. IEEE, 2008.

[76] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(2):8, 2013.

[77] ShriHari RajendranRadhika, Aviral Shrivastava, and Mahdi Hamzeh. Path selection based acceleration of conditionals in cgras. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 121–126. EDA Consortium, 2015.

[78] Yeonghun Jeong, Seongseok Seo, and Jongeun Lee. Evaluator-executor transformation for efficient pipelining of loops with conditionals. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):62, 2013.

[79] Shouyi Yin, Pengcheng Zhou, Leibo Liu, and Shaojun Wei. Acceleration of nested conditionals on cgras via trigger scheme. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 597–604. IEEE Press, 2015.

[80] Aviral Shrivastava, Jared Pager, Reiley Jeyapaul, Mahdi Hamzeh, and Sarma Vrudhula. Enabling multithreading on cgras. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 255–264. IEEE, 2011.

[81] Dani Voitsechov and Yoav Etsion. Single-graph multiple flows: Energy efficient design alternative for gpgpus. In *ACM SIGARCH computer architecture news*, volume 42, pages 205–216. IEEE Press, 2014.

[82] Dani Voitsechov and Yoav Etsion. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. *arXiv preprint arXiv:1801.05178*, 2018.

[83] Ricardo Ferreira, Virginia Duarte, Waldir Meireles, Manuela Pereira, Luigi Carro, and Simon Wong. A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 188–195. IEEE, 2013.

[84] Yongjun Park, Hyunchul Park, and Scott Mahlke. Cgra express: accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 271–280. ACM, 2009.

[85] Rafael Maestre, Fadi J Kurdahi, Nader Bagherzadeh, Hartej Singh, Román Hermida, and Milagros Fernández. Kernel scheduling in reconfigurable computing. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 90–96. IEEE, 1999.

[86] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 191–200. ACM, 2009.

[87] Jonghee W Yoon, Jongeun Lee, Sanghyun Park, Yongjoo Kim, Jinyong Lee, Yunheung Paek, and Doosan Cho. Architecture customization of on-chip reconfigurable accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(4):52, 2013.

[88] Ganghee Lee, Seokhyun Lee, Kiyoung Choi, and Nikil Dutt. Routing-aware application mapping considering steiner points for coarse-grained reconfigurable architecture. In *Reconfigurable computing: architectures, tools and applications*, pages 231–243. Springer, 2010.

[89] Yoonjin Kim, Ilhyun Park, Kiyoung Choi, and Yunheung Paek. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture. In *Proceedings of the 2006 international symposium on Low power electronics and design*, pages 310–315. ACM, 2006.

[90] Zion Kwok and Steven JE Wilton. Register file architecture optimization in a coarse-grained reconfigurable architecture. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 35–44. IEEE, 2005.

[91] Kyuseung Han, Jong Kyung Paek, and Kiyoung Choi. Acceleration of control flow on cgra using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429–432. IEEE, 2010.

[92] Ganghee Lee, Seokhyun Lee, and Kiyoung Choi. Automatic mapping of application to coarse-grained reconfigurable architecture based on high-level synthesis techniques. In *SoC Design Conference, 2008. ISOCC'08. International*, volume 1, pages I–395. IEEE, 2008.

[93] Dipal Saluja. Register file organization for coarse-grained reconfigurable architectures: Compiler-microarchitecture perspective. Master's thesis, Arizona State University, 2014.

[94] Sangyun Oh, Hongsik Lee, and Jongeun Lee. Efficient execution of stream graphs on coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12):1978–1988, 2017.

[95] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[96] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. EGRA: A coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(6):1062–1074, 2011.

[97] Gerald Aigner. An overview of the suif2 compiler infrastructure. *http://suif. stanford. edu/*, 2000.

[98] Gurobi Optimization. Gurobi Optimizer. `http://www.gurobi.com/`, 2017.

[99] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings-Computers and Digital Techniques*, 150(5):255, 2003.

[100] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[101] Mahdi Hamzeh. *Compiler and Architecture Design for Coarse-Grained Programmable Accelerators*. Arizona State University, 2015.

[102] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[103] Dajiang Liu, Shouyi Yin, Yu Peng, Leibo Liu, and Shaojun Wei. Optimizing spatial mapping of nested loop for coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(11):2581–2594, 2015.

[104] Matthew Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *WWC*, 2001.

[105] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[106] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[107] Shail Dave and Aviral Shrivastava. Ccf: A cgra compilation framework. 2018.

[108] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[109] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. HyCUBE: A CGRA with Reconfigurable Single-Cycle Multi-Hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[110] Panagiotis Theocharis and Bjorn De Sutter. A bimodal scheduler for coarse-grained reconfigurable arrays. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–26, 2016.

[111] Ashay Dharwadker. The clique algorithm, 2006.

[112] Mahesh Balasubramanian and Aviral Shrivastava. CRIMSON: Compute-Intensive Loop Acceleration by Randomized Iterative Modulo Scheduling and Optimized Mapping on CGRAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3300–3310, 2020.

[113] Laura Astolfi, Febo Cincotti, Donatella Mattia, M Grazia Marciani, Luiz A Baccala, Fabrizio de Vico Fallani, Serenella Salinari, Mauro Ursino, Melissa Zavaglia, Lei Ding, et al. Comparison of different cortical connectivity estimators for high-resolution eeg recordings. *Human brain mapping*, 28(2):143–157, 2007.

[114] Kendrick N Kay, Thomas Naselaris, Ryan J Prenger, and Jack L Gallant. Identifying natural images from human brain activity. *Nature*, 452(7185):352, 2008.

[115] Jonathan W Pillow, Jonathon Shlens, Liam Paninski, Alexander Sher, Alan M Litke, EJ Chichilnisky, and Eero P Simoncelli. Spatio-temporal correlations and visual signalling in a complete neuronal population. *Nature*, 454(7207):995, 2008.

[116] Kristofer E Bouchard, James B Aimone, Miyoung Chun, Thomas Dean, Michael Denker, Markus Diesmann, David D Donofrio, Loren M Frank, Narayanan Kasthuri, Chirstof Koch, et al. High-performance computing in neuroscience for data-driven discovery, integration, and dissemination. *Neuron*, 92(3):628–631, 2016.

[117] Kristofer E. Bouchard et al. International neuroscience initiatives through the lens of high-performance computing. *Computer*, 51(4):50–59, April 2018.

[118] Bijan Pesaran, Martin Vinck, Gaute T Einevoll, Anton Sirota, Pascal Fries, Markus Siegel, Wilson Truccolo, Charles E Schroeder, and Ramesh Srinivasan. Investigating large-scale brain dynamics using field potential recordings: analysis and interpretation. *Nat Neurosci*, 21(7):903–919, 2018.

[119] Clive WJ Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica: Journal of the Econometric Society*, pages 424–438, 1969.

[120] Helmut Lütkepohl. *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[121] Kristofer E. Bouchard et al. Union of Intersections (UoI) for Interpretable Data Driven Discovery and Prediction. In *Advances in Neural Information Processing Systems*, pages 1078–1086, 2017.

[122] Trevor Ruiz, Mahesh Balasubramanian, Kristofer E Bouchard, and Sharmodeep Bhattacharyya. Sparse, low-bias, and scalable estimation of high dimensional vector autoregressive models via union of intersections. *arXiv preprint arXiv:1908.11464*, 2019.

[123] Pinghua Gong and Jieping Ye. Honor: Hybrid optimization for non-convex regularized problems. In *Advances in Neural Information Processing Systems*, pages 415–423, 2015.

[124] Stephen Boyd et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

[125] Sumanta Basu and George Michailidis. Regularized estimation in sparse high-dimensional time series models. *Ann. Statist.*, 43(4):1535–1567, 08 2015.

[126] Jianqing Fan, Jinchi Lv, and Lei Qi. Sparse high-dimensional models in economics. *Annu. Rev. Econ.*, 3(1):291–317, 2011.

[127] Mike Folk et al. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[128] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[129] Endong Wang et al. Intel Math Kernel Library. In *High-Performance Computing on the Intel Xeon Phi™—*, pages 167–188. Springer, 2014.

[130] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86. ACM, 2018.

[131] K O'Leary, I Gazizov, A Shinsel, R Belenov, Z Matveev, and D Petunin. Intel Advisor Roofline Analysis: A New Way to Visualize Performance Optimization Trade-offs. *Intel Software: The Parallel Universe*, 27:58–73, 2017.

[132] Mark Howison et al. Tuning HDF5 for lustre file systems. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2010.

[133] Joseph E O'Doherty, MMB Cardoso, JG Makin, and PN Sabes. Nonhuman Primate Reaching with Multichannel Sensorimotor Cortex Electrophysiology. *Zenodo http://doi. org/10.5281/zenodo*, 583331, 2017.

[134] Tor M Aamodt, Wilson WL Fung, I Singh, A El-Shafiey, J Kwa, T Hetherington, A Gubran, A Boktor, T Rogers, A Bakhoda, et al. Gpgpu-sim 3. x manual, 2012.

[135] Jason Power, Joel Hestness, Marc S Orr, Mark D Hill, and David A Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.