AUTOMATED GENERATION OF BANKED MEMORY ARCHITECTURES
IN THE HIGH-LEVEL SYNTHESIS OF MULTI-THREADED SOFTWARE

by

Yu Ting Chen

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto

© Copyright 2017 by Yu Ting Chen

Abstract

Automated Generation of Banked Memory Architectures in the High-Level
Synthesis of Multi-Threaded Software

Yu Ting Chen

Master of Applied Science

Graduate Department of Electrical & Computer Engineering

University of Toronto

2017

The Legup High-Level Synthesis (HLS) tool permits the synthesis of multi-threaded software into parallel hardware, where parallel software threads are realized as concurrently operating hardware units. A common performance bottleneck in any parallel implementation is memory bandwidth – parallel threads demand concurrent access to memory resulting in contention that hurts performance. FPGAs contain an abundance of independently accessible memories offering high internal memory bandwidth. We describe an approach for leveraging such bandwidth in the context of synthesizing parallel software into hardware. Our approach applies trace-based profiling to determine how a program’s arrays should be *automatically* partitioned into sub-arrays, which are then implemented in separate RAM blocks within the target FPGA. The end result is that each thread, when implemented in hardware, has exclusive access to its own memories to the extent possible, significantly reducing contention and arbitration and thus raising performance.

Acknowledgements

I would like to thank my advisor Jason Anderson for his insight and support throughout my masters experience. Through your help and guidance, I have been given the confidence to pursue my ideas and develop not only research skills, but also as a person. I would also like to thank the LegUp group, in particular Andrew Canis, Blair Fort, James Choi, Lanny Lian, Bain Syrowik, Julie Hsiao, and Prof. Stephen Brown, as well as the other members of Jason's group, Xander Chin, Jin Hee Kim, Brett Grady, and Steven Niu for all your help and interest in the project.

Thank you to all my friends in Pratt 477 and surrounding labs for our discussions and your encouragement. Especially, I would like to thank Charles Lo, José Antonio Muñoz Cepillo, Joshua San Miguel, Karthik Ganesan, Mario Badr, Naif Tarafdar, Nariman Eskandari, Patrick Judd, and Sanket Pandit.

My masters experience also would not be complete without the support of friends, and I would like to sincerely thank all those I have gotten to know through these years in graduate school. You have opened my mind to the richness of life through our engaging conversations. I am grateful to have met all of you.

Finally, I would like to thank my parents for believing in me and for encouraging me throughout my studies. Thank you for showing your endless support, and most of all, for giving me opportunities through your sacrifices.

Contents

Acknowledgements	iii
Table of Contents	iv
List of Tables	vi
List of Figures	vii
1 Introduction	1
1.1 Contributions	3
1.2 Thesis Organization	4
2 Background	5
2.1 High-Level Synthesis Overview	5
2.2 LLVM Compiler	7
2.2.1 LLVM Intermediate Representation	7
2.3 Hardware Synthesis of Parallel Software	10
2.4 LegUp Memory Architecture Synthesis	10
2.5 Related Work	12
2.5.1 Memory Partitioning in HLS	12
2.5.2 Automatic Memory Partitioning	15
2.6 Summary	21

3	Array Partitioning in LegUp	22
3.1	Array Partitioning Algorithm	22
3.1.1	Array Partitioning Schemes	22
3.1.2	Array Partitioning Code Example	23
3.1.3	Array Partitioning Implementation	25
3.1.4	Non-predicated Array Partitioning Implementation	30
3.1.5	Limitation on Partitioning Configurations	33
3.1.6	Exploration Space	37
3.2	Array Partitioning Evaluation	41
3.2.1	Benchmark Descriptions	41
3.2.2	Performance Evaluation	43
3.2.3	Performance and Area Trade-off	48
3.3	Summary	55
4	Automatic Memory Partitioning	58
4.1	Automatic Memory Partitioning Tool Flow	58
4.1.1	Memory Trace Collection	59
4.1.2	Memory Simulator	61
4.2	Evaluation	65
4.2.1	Exploration Space	65
4.2.2	Memory Simulator Accuracy	67
4.2.3	Sources of Error	73
4.3	Summary	81
5	Conclusion	82
5.1	Future Work	83
	Appendices	85
A	Array Partitioning Full Results	86
	Bibliography	95

List of Tables

2.1	Summary of HLS tools.	13
3.1	Partition configuration set of allowable value ranges for number of partition, block size, and repetition factor parameters.	36
3.2	Baseline unpartitioned versus best partition scheme(s) execution cycles.	44
3.3	Cycle execution performance versus area results for <i>matrixadd</i> benchmark.	49
3.4	M10K usage for unpartitioned <i>matrixadd</i> benchmark	53
3.5	M10K usage for partitioning by 8 blocks <i>matrixadd</i> benchmark	54
3.6	Cycle execution performance versus area results for <i>histogram</i> benchmark.	54
4.1	Size of the partitioning configuration search space.	66
4.2	Memory simulator evaluation for 16 partitions and under.	67
A.1	<i>Matrixadd</i> benchmark full performance and area results (average of 5 synthesis runs).	87
A.2	<i>Histogram</i> benchmark full performance and area results (average of 5 synthesis runs).	88
A.3	<i>Matrixmult</i> benchmark full performance and area results.	89
A.4	<i>Matrixmult (cyclic)</i> benchmark full performance and area results.	90
A.5	<i>Matrixtrans</i> benchmark full performance and area results.	91
A.6	<i>Matrixtrans (block cyclic)</i> benchmark full performance and area results.	92
A.7	<i>Substring</i> benchmark full performance and area results.	93
A.8	<i>Los</i> benchmark full performance and area results.	94

List of Figures

2.1	Elaborated LegUp backend.	6
2.2	Array partitioning tool flow.	7
2.3	CFG for loop segment demonstrating usage of phi instructions.	9
2.4	LegUp memory arbiter architecture	11
2.5	Using linear transformation to achieve partitioning.	16
3.1	Partitioning schemes applied to 10×10 matrix in the row dimension. .	23
3.3	Select and or tree structure for 4 partitions.	28
3.4	CFG transformation of <i>matrixmult</i> benchmark	31
3.5	Visualization of array under multiple partition schemes, partition number on left and index number on right.	33
3.6	Possible block cyclic partitioning schemes for a 17 element array. . . .	39
3.7	Matrix Access Patterns for <i>matrixmult</i> and <i>matrixtrans</i>	43
3.8	<i>Matrixadd</i> array access pattern for block partitioning by 4.	46
3.9	<i>Matrixadd</i> array access pattern for block cyclic partitioning by 8 with a block size of 2.	47
3.10	Performance and area comparison for <i>matrixadd</i> benchmark with block cyclic partitioning across different number of partitions with block size 2.	50
3.11	Normalized area-delay product for <i>matrixadd</i> benchmark with block cyclic partitioning across different number of partitions with block size 2.	51

3.12	Performance and area comparison for <i>matrixadd</i> benchmark with block partitioning across different number of partitions.	52
3.13	Normalized area-delay product for <i>matrixadd</i> benchmark with block partitioning across different number of partitions.	53
3.14	Performance and area comparison for <i>histogram</i> benchmark with block partitioning across different number of partitions.	56
3.15	Normalized area-delay product for <i>histogram</i> benchmark with block partitioning across different number of partitions.	57
4.1	Memory access schedule of 2-threaded execution to memory access chain.	61
4.2	Memory simulator <i>matrixadd</i> predicted versus actual cycle of last memory access	69
4.3	Memory simulator <i>histogram</i> predicted versus actual cycle of last memory access	69
4.4	Memory simulator <i>matrixmult</i> predicted versus actual cycle of last memory access	70
4.5	Memory simulator <i>matrixmult</i> (<i>cyclic</i>) predicted versus actual cycle of last memory access	70
4.6	Memory simulator <i>matrixtrans</i> predicted versus actual cycle of last memory access	71
4.7	Memory simulator <i>matrixtrans</i> (<i>block cyclic</i>) predicted versus actual cycle of last memory access	71
4.8	Memory simulator <i>substring</i> predicted versus actual cycle of last memory access	72
4.9	Memory simulator <i>los</i> predicted versus actual cycle of last memory access	72

4.10	Comparison of inner loop basic block schedule for <i>matrixadd</i> as seen through the <i>scheduleviewer</i> . Blue blocks denote the cycles the instructions to the left of the block is scheduled into; yellow blocks denote instructions under analysis; red boxed instructions are dependencies of the yellow block instructions; orange boxed instructions depend on the yellow block instruction output.	75
4.11	<i>Matrixadd</i> final optimized IR without <code>-disable-iv-rewrite</code> option.	78
4.12	<i>Matrixadd</i> final optimized IR with <code>-disable-iv-rewrite</code> option enabled.	79

Chapter 1

Introduction

High-Level Synthesis (HLS), since its introduction almost four decades ago, is becoming a prevalent design methodology for FPGAs [14, 32]. The latest survey of the HLS landscape [32] shows a number of tools from both the commercial and research communities. Xilinx’s Vivado HLS (formerly AutoPilot’s AutoESL) [44], Intel’s HLS Compiler [25], Bluespec [3], ROCCC [39] from UC Riverside, and LegUp [7] from the University of Toronto are examples of commercial and academic HLS tools currently available. The main goal of HLS tools is to reduce the design effort involved in creating dedicated hardware blocks for computation. This allows hardware and software designers alike to take advantage of the higher performance and lower energy requirements of hardware accelerators. HLS is an active area of research with significant effort being put towards closing the performance, power, and area gap between HLS-generated hardware and human-expert-designed hardware.

The LegUp HLS tool provides many of these performance enhancing and area saving features, such as loop-pipelining, multi-pumping of DSP units, and producing parallel hardware units to exploit the parallel FPGA fabric. However, a frequently occurring culprit for limited performance in any hardware computing system is memory; that is, memory bandwidth and contention for memory among multiple requesters are common roots of limited computational throughput. Processors, and

the software programs that run on them, are typically based on the assumption of a single unified memory address space. FPGAs, on the other hand, contain many small memories, distributed spatially, which can each be accessed concurrently. In this thesis, we consider automated generation of banked memory architectures in the high-level synthesis of parallel software programs into parallel FPGA hardware, with the goal of reducing memory contention and raising bandwidth.

One of the primary aims in pursuing a hardware implementation versus a software implementation is to exploit spatial parallelism in the hardware. Towards this, some modern high-level synthesis (HLS) tools, such as the LegUp HLS framework from the University of Toronto, provide spatial exploitation capability at the software-specification stage by supporting the synthesis of parallel software (threads) into parallel hardware [10]. The parallel software threads in multi-threaded C programs written using the Pthreads or OpenMP standards, are realized as concurrently operating units in hardware. In this research, we are concerned with the synthesis of memory architectures for such spatially parallel implementations, where concurrently operating hardware units may vie for access to the same memory ports at the same time. We seek to alleviate such contention through automated synthesis of banked architectures such that hardware units have (partially) exclusive access to particular ports to the extent possible. This is fundamentally different from the notion of exposing more memory ports to enable multiple accesses in the same cycle **for a single module** in that the coordination between threads causes unpredictability in scheduling and thus we cannot strictly schedule memory accesses statically.

We implement our work in the LegUp HLS framework within its Pthreads parallel flow. We first augmented LegUp with user-constraint-driven array partitioning, similar to that available in Xilinx’s Vivado HLS tool. Based on user-supplied directives, the new functionality permits arrays to be partitioned in a variety of different ways by the compiler, where each partition is then implemented in separate RAMs within the FPGA implementation. However, three issues exist with the constraint-driven

approach: 1) it can be difficult to know which style of array partitioning is best for a given array as it depends on the memory access patterns of the parallel program and also the architecture of the target FPGA, 2) specifying array partitioning directives is non-standard, and we believe it to be particularly difficult for software engineers with no knowledge of FPGA hardware, and 3) unpredictability in thread contention patterns, i.e., memory access order of threads lead to difficulties in deciding on the array partitioning scheme.

In light of these three issues, we propose a memory-trace profiling-based approach that automatically determines array partitionings to realize banked memory architectures offering low memory contention and high bandwidth for parallel spatial hardware as produced by LegUp’s Pthreads flow. An advantage of the proposed approach is that it requires a single HLS execution and hardware simulation of the generated hardware to extract a memory access trace, as opposed to more exhaustive approaches requiring many synthesis/simulation runs. Specifically, we have developed a memory-trace simulator that permits rapid evaluation of various memory partitioning schemes (banking architectures) using a single execution trace of the hardware with unpartitioned arrays through modelling contention and arbitration.

1.1 Contributions

The key contributions of this thesis are:

- Infrastructure for array partitioning implemented in the open-source LegUp-HLS framework.
- A lightweight memory simulator for automated trace-based array partition scheme detection to reduce execution cycles in HLS-generated hardware for multi-threaded software.

- Automated arbiter insertion and support for cases where “exact” thread-driven partitioning is not possible, for example, for scenarios wherein two threads *occasionally* access the same RAM.
- An experimental study demonstrating the efficacy of the automated partitioning technique in comparison with a brute-force approach.
- The contents of this thesis have been published [9].

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the LegUp HLS tool and the necessary accompanying concepts of HLS, as well as a summary of prior works in the area of HLS and automatic array partitioning. Chapter 3 describes the implementation of array partitioning in LegUp introduced by this work and shows a study of the area and performance results for a set of 8 benchmarks. Chapter 4 describes the automatic memory partitioning tool and presents an evaluation of the effectiveness of the tool for the same set of 8 benchmarks presented in Chapter 3. Finally, Chapter 5 concludes with a discussion of possible future works.

Chapter 2

Background

In this chapter, we will provide an overview of the LegUp tool in which we build our memory partitioning feature. First, a brief overview of HLS is provided along with brief coverage of the LLVM Compiler. Then, a description is provided of the current LegUp memory architecture. Following this, we provide some context for the state of HLS tools through a study of the currently available academic and research tools, as well as present the features of some of these tools. Finally, a study of the various research in the area of automatic memory partitioning concludes this chapter.

2.1 High-Level Synthesis Overview

High-level synthesis generally consists of 4 stages, shown in Figure 2.1: *i) allocation*, *ii) scheduling*, *iii) binding*, and *iv) code generation*. The allocation step determines the hardware resources, i.e. functional and storage units, required to implement the software algorithm. In the scheduling stage, each instruction is scheduled in a particular clock cycle and a finite-state machine is created for each function. LegUp uses the SDC scheduling algorithm from [15]. The scheduling problem is modelled and solved as a system of difference constraints, where dependency, resource, latency, clock period, and relative timing constraints are expressed as a set of equalities and

Figure 2.1: Elaborated LegUp backend.

inequalities that make up the system. A linear program solver is used to create a state transition graph, otherwise known as a finite-state machine, with the goal of satisfying all constraints while minimizing the latency of the schedule. Binding involves assignment of *i) operations to functional units*, and *ii) data to storage units*. LegUp implements bipartite weighted matching [20] to determine the optimal instruction to functional unit binding. In bipartite weighted matching, operations and functional units are grouped in two disjoint sets. Weights are assigned to edges between the two sets such that edges exist if the operation and functional unit pair represents a possible mapping. The weights are set in order to minimize a cost function, which is typically to reduce the sizes of muxes on inputs to the shared functional units. Code generation is when the in-memory hardware model is translated into HDL such as Verilog.

The LegUp HLS tool implements the 4 HLS stages within the open-source LLVM compiler [28]. The overall flow is depicted in Figure 2.2. Within LLVM, the input

Figure 2.2: Array partitioning tool flow.

program is represented in the compiler’s intermediate representation (IR). Prior to LegUp HLS, the program’s IR is subjected to LLVM’s compiler optimization passes. LegUp takes the optimized IR as input and performs the traditional HLS stages [16] of resource allocation, scheduling, binding and Verilog generation. In this work, array partitioning for the purposes of banked memory architecture synthesis is implemented as a generic LLVM optimization pass. A brief description of the LLVM Compiler and the LegUp memory architecture is given in the sections below.

2.2 LLVM Compiler

In general, a compiler is composed of 3 stages: the frontend, optimizations, and backend. The frontend performs the translation from an input language such as C/C++, Fortran, Java, etc. to an intermediate language known as the intermediate representation (IR). The IR is a language and architecture agnostic representation of an algorithm. The optimization stage is when various optimizations are performed on the IR to create more performant programs. In the backend, the architecture independent IR is translated into a machine executable depending on the targetted platform.

2.2.1 LLVM Intermediate Representation

The LLVM IR is a low-level RISC assembly code-like intermediate representation. It is composed of simple instructions such as `multiply`, `add`, `branch`, `call`, etc. The IR is organized in a way such that it exposes the underlying control flow of the program. This is achieved through the concept of control flow graphs and basic

blocks. A control flow graph (CFG) is a representation of a program and its control flow expressed in graphical form. A CFG consists of nodes, known as basic blocks, and edges. Basic blocks are collections of instructions that form straight line non-divergent code with a single entry point and single exit point. This means that if one instruction in the basic block is to be executed, then so too will the rest of the instructions in the same basic block. At the end of each basic block is a terminating instruction such as a **branch** or a **return**, which dictates the successors of the basic block.

Below is a subset of LLVM IR instructions [24] that we believe will be useful for the reader to understand in the context of this thesis. The output of each of the instructions is assigned to a variable, denoted by the prefix %.

Select Instruction

```
%a = select <ty1> %b, <ty2> %c, <ty2> %d
```

Chooses between two values of type <ty2> through a selector value of type <ty1>. The first value %c is selected when the selector value %b is true, otherwise %d is selected. The final output is assigned to %a.

GetElementPtr Instruction

```
%a = getelementptr <ty>, <ty>* %b[, [inrange] <ty> <idx>]*
```

The `getelementptr` instruction gets the address of a particular element in a data structure. It does not perform the memory access. The value %b is the base address of the data structure, the following list of arguments are the indices of the sub-element whose address we want to compute.

Phi Instruction

```
%a = phi i32 [ %b, %bb0 ], [ %c, %bb1 ], ...
```

Figure 2.3: CFG for loop segment demonstrating usage of phi instructions.

The `phi` instruction lists predecessor basic block and value pairs and implements the ϕ node as explained below.

The LLVM IR follows the static single assignment (SSA) form, meaning each variable is assigned only once. Multiple assignments to the same software variable imply that multiple versions of the variable will be created, each being assigned only once. To distinguish which variable is to be used, we will require an instruction known as the phi instruction. A phi instruction is used for disambiguating between the different versions of a software variable based on the control flow. For example Figure 2.3 shows the usage of the phi instruction in a C loop body (a), and the associated CFG (b). The loop performs an accumulation of values and stores the sum in the variable `v`, the loop iterator is the variable `i`. From the CFG, we see that `v` and `i` are initialized in the loop preheader `B0`. Basic block `B1` is the loop body and contains two phi instructions. The first phi instruction determines the value of the software variable `v`, which can be either `t1`, if the previously executed basic block was `B0`, or `t5`, if the previously executed basic block was `B1`. The second phi instruction determines the value of the software variable `i`, which can be either `t2`,

if the previously executed basic block was B0, or `t6`, if the previously executed basic block was B1. Here we have two temporary versions for each of the software variables `v` and `i`, in `t1`, `t5`, `t2`, and `t6`, making it possible to assign each variable only once.

2.3 Hardware Synthesis of Parallel Software

In LegUp’s Pthreads synthesis flow, parallel multi-threaded software is synthesized into parallel hardware whose behaviour closely matches the software semantics. Each software thread is synthesized into an instance of a hardware unit. Pthreads library function calls in the original C are replaced with LegUp-specific wrapper functions that cause specific hardware structures to be produced in the Verilog generation phase. For example, calls to `pthread_create` are translated into FSM functionality that invokes a hardware unit corresponding to a software thread. Support is provided for common thread synchronization approaches including mutexes (for critical sections) and barriers. Ultimately, the Pthreads flow permits a wide range of spatial parallel implementations to be realized through software changes alone. The interested reader is referred to [10, 11] for complete details. Our work here seeks to alleviate a specific (and common) performance bottleneck in such parallel hardware, namely, when multiple parallel hardware units contend for memory.

2.4 LegUp Memory Architecture Synthesis

LegUp HLS memory architecture synthesis supports 3 types of different memory designations: *local*, *shared-local*, and *global* [11]. LegUp uses “points-to” analysis in LLVM to designate arrays as one of these types. Local arrays are accessed by a single function in the C code. Shared-local arrays are accessed by a limited number of functions, determined statically at compile time. If the points-to analysis was not able to statically determine the requesters to the array, then it is designated as global. In hardware, such arrays are implemented behind a global memory controller

that uses a unique tag for each array to steer accesses to the correct SRAM block containing the array [8]. Global array accesses can be slow since accesses to all global memory must be made through a single global memory controller. For this work, we focus on shared-local arrays since global memory requests cannot be parallelized due to the limitation of the memory controller, and local memories do not experience the problem of contention between threads as they are not shared.

Figure 2.4: LegUp memory arbiter architecture

LegUp HLS implements each array in a program in a separate logical RAM. Depending on its size, the logical RAM may be implemented in multiple physical block RAMs in the FPGA implementation (e.g. when the array size exceeds the SRAM block size in the FPGA fabric). For RAMs that may be accessed concurrently by parallel hardware, LegUp automatically inserts arbitration circuitry. The arbiter permits single-cycle access in the absence of contention through two ports (a and b). Under contention, it is resolved in a round-robin style, in which case a hardware unit will stall until it is granted access, degrading performance. Such degradations are particularly common in the Pthreads synthesis flow.

Figure 2.4 provides a visualization of the connection of the arbiter signals to the requesting module instances and memory. In this example, we observe a circuit compiled from a multi-threaded input, where there are four instances of the `add` kernel

function instantiated as `add_inst0` through `add_inst3`. There are two shared ROMs `array0_inst` and `array1_inst`, and each ROM has two ports, `a` and `b`. Each port of each ROM has an associated arbiter, which receives request signals from each of the four requester modules. For the sake of space, we show only the arbiters connected to port `a`. The arbiter’s grant signal is used as the select for the multiplexer that chooses the address to use. At the output of the ROM, the data is passed through the `data_receiver` module, which is used in simulation to model different memory latencies.

To highlight the memory port contention issue, consider, for example, parallelized vector addition, $Z = A + B$, where Z, A, B are n -element arrays. In a typical multi-threaded implementation, each thread would operate on a portion of the original arrays. With t threads implemented as t parallel hardware units there would be significant contention on the ports of RAMs holding the arrays, even if the RAMs were in dual-port mode.

2.5 Related Work

In this section, we will first explore available HLS tools and their support of memory partitioning. Then we will present some of the current and past research in the area of automatic memory partitioning selection.

2.5.1 Memory Partitioning in HLS

A number of commercial and academic tools exist. A concise summary is presented in a study [32] showing all the active tools as of 2016. Table 2.1 shows all the currently active tools and whether they support memory partitioning to the best of our knowledge. The level of documentation varies widely between these tools and is sometimes unavailable to the public. Below we present a description of some of the common HLS tools and the optimizations they support.

Table 2.1: Summary of HLS tools.

Tool Name	Owner	Tool Supports Memory Partitioning ^a
eXCite [45]	Y Explorations	No
CoDeveloper [22]	Impulse Accelerated	No
Catapult-C [6]	Calypto Design Systems	No
Bluespec [3]	BlueSpec Inc.	No
CHC [1]	Altium	Yes
Stratus [5]	Cadence	No
DK Design Suite [31]	Mentor Graphics	No
GAUT [17] ^b	U. Bretagne	- ^c
MaxCompiler [29]	Maxeler	No
ROCCC [39]	Jacquard Comp.	No
Synphony C [37]	Synopsys	No
CyberWorkBench [34]	NEC	No
VivadoHLS [44]	Xilinx	Yes
Intel HLS Compiler [25]	Intel Corp.	- ^d
LegUp [7]	U. Toronto	Current work
Bambu [4]	PoliMi	No
DWARV [33]	TU. Delft	No
Trident [38]	Los Alamos NL	No
CHiMPS [35]	U. Washington	No
Kiwi [18]	U. Cambridge	Yes
gcc2verilog [21]	U. Korea	No
HercuLeS [27]	Ajax Compiler	No

^aRefers to array partitioning as part of the tool flow, the arrays can always be re-written into multiple partitions by the software designer.

^bAt the time of this thesis, the website was under maintenance

^cDocumentation unclear

^dLimited documentation

EXCite [45] is a tool that takes as input an ANSI-C program and produces Verilog or VHDL RTL for the FPGA or ASIC logic synthesis tools. The optimizations it supports are loop pipelining, bitwidth reduction, and other common compiler level optimizations, however it does not support memory partitioning.

CoDeveloper [22] provides an environment to create parallel computing modules in a CPU-accelerator environment, which extends ANSI-C with the Impulse C library [23]. The tool targets the domains of image processing, digital signal processing, and compression/encryption algorithms and does not support memory partitioning.

Catapult-C [6] is an ANSI-C++ and SystemC HLS tool suited for full-chip synthesis. While the tool is able to determine interface protocols automatically to decipher whether the necessary memory interface should be streaming, single/dual-ported, FIFO, etc, it does not perform memory partitioning.

Bluespec HLS [3] provides a platform for users to create high-level descriptions of designs in the Bluespec System Verilog language to be compiled through the Bluespec Compiler into Verilog RTL or SystemC executable models. There is no support for memory partitioning.

MaxCompiler [29] is a tool from Maxeler Technologies which takes as input a language called MaxJ, a variant of Java, and creates a computing platform composed of kernels; managers, which manage the I/O of the kernels; and a host CPU application, which sets up communication between the CPU and FPGA. There is no support for memory partitioning.

Riverside Optimizing Compiler for Configurable Computing (ROCCC) [26] is a C to VHDL HLS tool focused on FPGA acceleration for streaming data. The aim of ROCCC is to create hardware that achieves high throughput, minimizes off-chip memory accesses, and minimizes area. The user is given fine-grained control through the GUI for the output hardware, especially for loops. ROCCC provides a multitude of hardware specific compiler optimizations such as loop pipelining, temporal common sub-expression elimination, and systolic arrays. However, it does not support memory partitioning.

Vivado HLS from Xilinx [44] is a commercially available tool that provides support for array partitioning through the `array_partition` pragma. The motivation for array partitioning in Vivado HLS is to increase the memory bandwidth to support loop optimizations, such as loop pipelining and loop unrolling. There are 3 supported modes of partitioning: *i) block*, *ii) cyclic*, and *iii) complete*, which can be applied to either a single dimension of the array or all dimensions. Another optimization that can be performed to arrays is reshaping, through the `array_reshape` pragma. The purpose of this pragma is also to improve memory bandwidth in loops. However, the

extra bandwidth is achieved through reshaping the array, such that the original data width of the elements is increased to pack more words into each access. Vivado HLS supports automatic decision making in array partitioning through the configuration parameter `config_array_partition`. This parameter allows for “automatic” partitioning to the extent that it is capable of determining whether scalarization of the array (splitting up individual elements) will allow the design to reach the specified throughput. However, it does not perform selection of partition configurations. Other tools that support memory partitioning do not provide enough publicly available documents to draw any conclusion.

2.5.2 Automatic Memory Partitioning

A number of recent works have considered memory banking in HLS, and below we highlight those we believe are most relevant to the present research. However, an important distinction of our work is the focus on the memory architecture synthesis in the multi-thread HLS context. This technique specifically targets the synthesis of banked architectures that result in reduced contention among the hardware implementation of parallel software; no prior work has considered this scenario.

Automatic memory partitioning techniques can broadly be grouped into 2 categories: *i) static* and *ii) profiling-driven*. Firstly, we discuss in detail the different static solutions.

In an influential journal article by Cong et al. [13], we are introduced to a static automatic partitioning algorithm that is meant to target the improvement of memory throughput while reducing power usage in loop pipelining use cases in HLS. The technique’s main goal is to combine scheduling and memory partitioning ideas such that the loop pipeline initiation interval (II) – the number of cycles between consecutive loop iterations, ideally 1 – is minimized. The partitioning problem is separated into 2 parts, initially a set of partitioning schemes is computed that satisfies the access pattern of an array in a loop nest subject to the constraints of a certain target

0	1	2	0
1	2	0	1
2	0	1	2
0	1	2	0

Figure 2.5: Using linear transformation to achieve partitioning.

throughput II, and the memory port limitation N. Candidate solutions are passed to the next step, which selects the best candidate partition in order to balance the access to all memory banks. Essentially, the second part tries to determine a possible schedule honouring the previously chosen partitioning schemes to eliminate contention, as scheduling of instructions has a significant impact on memory port contention. This is known as the throughput problem, which is solved using the notion of conflict graph colouring. A conflict graph is constructed, where the nodes represent the memory access instructions and the edges capture the pairwise conflicts within any given iteration. The goal then, is to find a schedule that minimizes the maximum clique – the largest number of adjacent vertices in the graph. The maximum clique directly corresponds to the number of required partitions. The maximum clique problem is solved using integer linear programming (ILP) formulation and the objective function is to maximize the number of independent sets – set of vertices that are not adjacent. For small-sized problems, the ILP formulation would suffice. In larger problems, a heuristic approach is taken instead to find optimal solutions in less time.

In a paper by Wang et al. [40] a static analysis is presented, which automatically chooses partitioning schemes for affine memory accesses. Affine memory accesses can be represented by equation 2.1, where I is an integer vector of size n denoting the indices in each array dimension of the access, M is an integer matrix of size $n \times l$, l is the depth of the loop nest, and V is a constant vector of size n .

$$\begin{bmatrix} m_{11} & m_{12} & \dots & m_{1l} \\ m_{21} & m_{22} & \dots & m_{2l} \\ \vdots & \ddots & & \\ m_{n1} & m_{n2} & \dots & m_{nl} \end{bmatrix} \times \begin{bmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \quad (2.1)$$

This method aims to find linear transformation functions to the accessed memory with the ultimate optimization goal of minimizing the number of memory banks while satisfying the constraint of conflict-free memory accesses. The bank mapping function, f , paired with the inner bank offset function, g , uniquely maps each access from the unpartitioned to the partitioned space. The algorithm starts by constructing a polytope, n-dimensional polygon, corresponding to the n-dimensional space representing the data domain – all memory elements that are accessed in any loop iteration. The partitioning algorithm then performs an exhaustive search to find the linear transformation vector, $\vec{a}$, for which the dot product of $\vec{a}$ and $\vec{x}$ represents a hyperplane of the polytope (data domain), where $\vec{x}$ is the d-dimensional index for the d-dimensional array. Once the optimal linear transformation function is found, the inner bank offset function is constructed to ensure no access conflicts. The following example demonstrates how linear transformations are used to translate between array indices from the original to new partitioned arrays. Figure 2.5 shows a 4×4 2-dimensional matrix where the elements are partitioned amongst 3 sub-arrays, the partition number is shown within each block. We use the following form for the bank mapping function:

$$f(\vec{x}) = \left\lfloor \frac{\vec{a} \cdot \vec{x}}{B} \right\rfloor \% N \quad (2.2)$$

where $\vec{x}$ is the memory reference vector, B is the block size, and N is the number of partitions. In order to achieve the partitioning shown in Figure 2.5, $B = 1$, $N = 3$, and we can use a partition vector $\vec{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Wang [41] takes the same approach as Wang [40] and extends the work to multi-

ported memories, and supports a wider variety of partitioning schemes as opposed to only cyclic support. It is thus called the generalized memory partitioning method.

In a study of memory partitioning’s effect on area by Gallo et al. [19], the authors demonstrate that the hyperplane method is not capable of finding all partitioning solutions and under certain circumstances, and will offer solutions, which are highly undesirable from an area perspective. In the circumstances under which this happens, the memory accesses exhibit a phenomenon known as *bank switching*. This occurs when the same memory access expression, for example `arr[2i][2j+1]`, does not always map to the same memory bank in the hyperplane solution between different loop iterations. This means that multiplexers are needed to select, depending on the state, which bank the memory access is intended for. Bank switching also requires multiplexers after the memory bank to direct the access to the correct downstream functional units in the datapath.

The work by Cilaro et al. [12] applies mathematical techniques to the memory partitioning problem. A polyhedral model is used to represent memory access patterns in loops and legal code transformations, and an integer lattice approach is used for memory partitioning, where the objective is to: 1) minimize contention, and 2) reduce “waste” (unused portions of the partitions).

The work by Meng et al. [30] is another linear transformation-based solution. However, this work extends the optimization objectives to cover not only reduction of II, but also limitation of the number of banks used, as well as to reduce the storage overhead of padding. The linear transformation vector is first computed mathematically, where the component in each dimension corresponding to an index position in the index vector is computed as the product of the maximum linearized difference between the indices of all parallel memory accesses. In essence, the computation of the linear transformation vector $\vec{a}$, when applied to each index vector in a particular parallel access pattern, will transform each index vector to a unique linearized value. The value is then converted to bank numbers through the modulo operation. The

algorithm offers a fast way to find the linear transformation vector but does not offer flexibility in determining multiple solutions.

The application of geometric (e.g. polyhedral) representations of memory accesses in a loop body to the memory architecture synthesis problem are not directly transferable to the related problem in the multi-thread memory contention context. While static approaches may work well to find solutions for conflict free array partitionings, it is deficient in cases where perfect partitioning cannot be expected. Furthermore, static methods are insufficient for detecting memory port conflicts that may potentially lead to large cycles delays due to the conflicts occurring very often from the conflicts that may not cause significant cycles slow down in the overall application.

Apart from static analyses, there are several profiling-driven works as well. Asher [2] describes a trace-based approach for automatic memory banking, again, with the ultimate goal of improving memory bandwidth to allow for an II of 1 for loop pipelining cases. The proposed tool flow implements binary instrumentation through the LLVM compiler in order to obtain a memory trace, where each entry shows the accessed address, the memory access ID corresponding to the instruction call site, and to which loop body the instruction belongs. By iterating through the memory trace multiple times, a list of Linear Memory Patterns (LMP) is found. The LMP records access information for each memory instruction including the upper and lower bounds of the addresses accessed, as well as the stride of the accesses. The stride represents the distance between the two closest memory addresses accessed by the instruction being examined. With this formulation, we can solve the mapping of LMPs to memory banks as a budgeted graph colouring problem. Here, each LMP is a node in the graph, and each memory bank is represented by a colour. An edge exists between LMPs that cannot be put in the same bank, i.e. there is a parallelism-conflict. Since memory banks have finite sizes, we need to also make sure that the graph colouring problem takes into account this constraint, where the total size of the required address space for all LMPs mapped to a specific bank (colour) must be less than or equal to the

size of that bank. Thus, the problem is slightly more complex than a regular graph colouring problem. The authors chose to use integer linear programming to solve this budgeted graph colouring problem. The objective function is to minimize the number of memory banks used, while satisfying the constraints to ensure that the memory assignment is valid. The proposed synthesis tool also needs to transform the pointers in the program from pointing to the address in the original memory space, to now point to a new address in the one of the new memory banks. This is done through a simple translation, which consists of a division and either addition or subtraction operation. The authors argue that these operations take a negligible overhead as the division becomes trivial when the stride (divisor) is a power of 2. In case of an error in pointer computation, i.e. the pointer is in an invalid range, the host will abort the run on the hardware accelerator and instead rerun on the CPU.

As with this work, a recent work by Zhou et al. [46] applied a trace-based approach, **TraceBanking**, to memory banking and employed a conflict-graph-based approach to map memory addresses to banks, with emphasis on multiplexer size minimization. The **TraceBanking** approach initially performs code instrumentation such that the execution of software will result in a memory trace. The memory trace is organized into lists of simultaneous accesses, where the notion of time is represented by **steps**. All accesses in the same step are meant to be executed in parallel. The memory trace is compressed such that time steps containing the same memory accesses are combined, then the trace is analyzed to find **mask bits**. Mask bits are bits of the addresses in the address trace that can be used to uniquely differentiate between other addresses in the same step. For example, for the hypothetical addresses 4'b0000 and 4'b0010, the mask bit would be the second least-significant bit. Once the potential mask bits are found, the problem can be modelled as a graph colouring problem. The **mask IDs**, the enumerated mask bits, are mapped to the different memory banks so as to reduce the number of bank conflicts. The **TraceBanking** approach is capable of producing results for both affine and non-affine memory access patterns. A satisfiability modulo

theories (SMT)-based approach is used to verify the mapping was indeed *conflict-free* in the case where the trace was incomplete or input dependent. As opposed to our approach, [46] solely handles the conflict-free cases (there is no arbitration), which means it will always choose the solution that produces a conflict-free design. However, in the interest of area, we may opt for a solution that is not completely conflict-free in order to reduce the number of partitions.

2.6 Summary

In this chapter, we gave an introduction to the steps in HLS, as well as an overview of the LegUp tool flow. We also summarized the operation of the LLVM compiler and described a few relevant instructions of the LLVM IR. A high-level description of the LegUp memory architecture is given along with a description of the round-robin arbiters used for arbitration of resources in LegUp. In the related work section, we summarize some of the currently available HLS tools and their support for memory partitioning. Finally, we discussed past work related to automatic memory partitioning selection, which fall into either static or profiling-driven methods.

Chapter 3

Array Partitioning in LegUp

In this chapter, we describe the available array partitioning schemes, as well as how the LLVM transformation pass performing array partitioning in LegUp works. We also present a performance and area evaluation of 8 benchmarks under various array partitioning schemes.

3.1 Array Partitioning Algorithm

3.1.1 Array Partitioning Schemes

Partitioning is available in any single dimension of a multi-dimensional array. There are 4 supported partitioning schemes for multi-dimensional arrays: *complete*, *block*, *cyclic*, and *block cyclic*. These are similar to the schemes supported by Vivado HLS [43]. For the purposes of this work, our focus is on globally declared data structures – i.e. those which are not locally scoped and therefore, can be shared between multiple functions or threads of the same function. In the *complete* scheme, each array element is designated to a separate partition. In the *block* scheme, partitions contain contiguous pieces of the original array. The *cyclic* scheme is an interleaved approach, where array elements at fixed distances apart are allocated to the same partition. *Block cyclic* combines the block and cyclic schemes.

Figure 3.1: Partitioning schemes applied to 10×10 matrix in the row dimension.

The visual example in Figure 3.1 shows all partitioning schemes applied to a 2-dimensional matrix where the rows of the matrix are partitioned completely (a), by 2 blocks of 5 rows each (b), cyclically as 2 partitions (c), and as block cyclic where the block size is 2 and there are 2 partitions (d).

In this work we will use several terms in reference to the partitioning schemes:

- **Size (S):** Number of elements of the multi-dimensional array in the dimension of the partitioning.
- **Number Partitions (n):** Number of total partitions to be created.
- **Block Size (b):** Size of contiguous elements in the dimension of partitioning that constitute the partition.
- **Repetition Factor (r):** The product of n and b , which is a useful value for block cyclic partitioning.

3.1.2 Array Partitioning Code Example

In this section, we show the mechanism for array partitioning through a simple code walk-through. In the following section, we will describe how this mechanism is implemented.

Listing (3.1) Array partitioning analogous C level transformations (Before).

```

1 // global input array
2 array[M][N] = {...};
3
4 int add (...) {
5     int sum = 0;
6     for (i=startrow; i<endrow; i++) {
7         for (j=startcol; j<endcol; j++) {
8             sum += array[i][j];
9         }
10    }
11    return sum;
12 }

```

Listing (3.2) Array partitioning analogous C level transformations (After).

```

1 // global input arrays
2 // block partition by 2 in rows dimension
3 sub_array_0[M/2][N] = {...};
4 sub_array_1[M/2][N] = {...};
5
6 int add (...) {
7     int sum = 0;
8     for (i=startrow; i<endrow; i++) {
9         for (j=startcol; j<endcol; j++) {
10            int new_i = i % b; // b is blocksize
11            int new_p = i / b;
12            array_0_load = (new_p == 0) ? sub_array_0[new_i][j] : 0;
13            array_1_load = (new_p == 1) ? sub_array_1[new_i][j] : 0;
14            sum += (array_0_load | array_1_load);
15        }
16    }
17    return sum;
18 }

```

Consider the following code examples in Listing 3.1 and 3.2. Listing 3.1 shows the simplified kernel code of a benchmark that sums up all the elements within an input array. The input array `array` is declared globally and contains M rows and N columns. Listing 3.2 shows the equivalent C code modifications for array partitioning under block partitioning in the row dimension by a factor of 2. Firstly, the declaration of `array` is split into two separate declarations where each partition contains half the number of rows as the original partition (line 3, 4). In the inner-loop body (line 10-14), the new row index is computed as $i \% b$ such that accesses to both partitions can be mapped to the correct rows (line 10). The partition number is computed as i / b (line 11). Lines 12 and 13 perform predicated loads where the predicate is the comparison of `new_p == 0` and `new_p == 1`. If the predicate evaluates to true, then the array is read, otherwise, the intermediate variables `array_0_load`

and `array_1_load` are set to 0. In line 14, the accumulated sum uses the value of the bitwise or of `array_0_load` and `array_1_load` to determine the final load value. It is important to note that no extra control flow has been introduced by the transformation.

3.1.3 Array Partitioning Implementation

Figure 2.2 in Section 2.1 shows the overall flow of LegUp, where an input C program is first translated to LLVM IR using the LLVM frontend, `clang-3.5`, then the unoptimized IR is transformed through various standard LLVM and LegUp-specific passes into the final optimized IR. Examples of standard LLVM optimization passes are loop unrolling, loop invariant code motion, loop strength reduction, function inlining, dead-code elimination, and constant propagation. LegUp specific passes are software passes aimed at creating or easing the translation to more performant hardware. For example, loop pipelining exploits loop-level parallelism by creating hardware that is able to concurrently execute multiple iterations of a loop body by pipelining the computation within the loop. LegUp also has a pass that transforms data of standard bitwidths to arbitrary bitwidths to reduce area and power consumption in the final hardware design. The final optimized IR is then passed through the LegUp backend and is converted into Verilog.

Array partitioning was implemented as a compiler pass in LLVM and it is one of the various passes applied to the IR during the optimization stage. The pass takes advantage of the predicated load/store operations supported by LegUp. Predicated load/stores differ from regular load/store instructions in that the memory enable signal is no longer only FSM-state dependent, but also dependent on a precomputed predicate value. Predicates are typically the true/false outcome of a comparison.

The array partitioning pass needs to perform 2 tasks: *i) replace the declaration of the array on which we are performing the partitioning*, and *ii) replace all accesses*

referencing the original array under partitioning to appropriately reference the newly created partitions.

Listing 3.3: Array partitioning pass algorithm

```

1  replace_array_accesses(array, pconfig) {
2
3      for each GEP gep referencing array {
4          newgeps = insert_gep_insts(gep)
5
6          for each user u of gep {
7              if (u is a load) {
8                  insert_predicated_load(pconfig, gep, newgeps)
9              }
10             else if (u is a store) {
11                 insert_predicated_store(pconfig, gep, newgeps)
12             }
13         }
14     }
15 }
16
17 insert_predicated_load(pconfig, gep, newgeps) {
18     selectList = [] // initialize as empty list
19     partTrans = insert_partition_translation(pconfig)
20     // create the predicated load for each partition
21     for each partition number p {
22
23         // create new load instructions
24         predicate = createICmpEq(partTrans, p)
25         newLoad = createLoad(newgeps[p])
26         newLoad.setPredicate(predicate)
27
28         // create select instructions
29         newSelect = createSelect(predicate, newLoad, 0)
30         selectList.insert(newSelect)
31     }
32 }
33 return createOrTree(selectList)
34 }
35
36 insert_predicated_store(pconfig, gep, newgeps) {
37     partTrans = insert_partition_translation(pconfig)
38     // create the predicated store for each partition
39     for each partition number p {
40         // create new store instructions
41         predicate = createICmpEq(partTrans, p)
42         newStore = createStore(newgeps[p])
43         newStore.setPredicate(predicate)
44     }
45 }
46 }

```

Replacement of the original array is done by creating new array declarations, where the original content of the array is copied into the new partitions corresponding to the partition scheme. The algorithm for the replacement of the accesses referenc-

ing the original unpartitioned array is shown in Listing 3.3. All array references are specified by the `getelementptr(GEP)` instructions in the LLVM IR. GEP instructions compute the address of a memory reference, given a reference to the starting location and optionally, the offset into the memory in the form of array indices. The array partitioning pass will find each GEP instruction that references the array under partitioning and insert a new GEP instruction for each partition created, as well as an address translation instruction to get the new index number in the dimension of partitioning (line 4). Now, for each instruction that used the previously found GEP instruction, either a load or store instruction, the pass will create a new instruction to compute the new partition number, `partTrans` (line 19, 37). The predicate is then created as the comparison of the newly computed `partTrans` and the partition number corresponding to each new GEP instruction (line 24, 41).

Leveraging the predicated load/store, the array partitioning pass can, for each memory access to a partitioned array, create multiple predicated memory instructions for each newly created partition (line 25, 42). The predicate is appended to the metadata of the IR instruction (line 26, 43), where it will be used in the Verilog generation of the circuit to create predicated memory instructions. The compare instruction (`icmp eq`) will evaluate to `true` only for one of these load instructions at run-time. Therefore, the transformation does not require extra memory bandwidth.

In the case of store instructions, the original store instructions are replaced by a set of new predicated store instructions, one for each partition, where each store instruction references different sub-arrays and, where one address reference will correspond to the original store location.

For load instructions, the original load instructions will also be replaced by a set of predicated load instructions similar to stores, however, we will need to implement a selection scheme to correctly identify which predicated load instruction returned the value we requested. For this, we implement an `or` tree structure. To implement the `or` tree, a `select` instruction is first created for each predicated load to select between the loaded result or 0's based on the predicate value (line 29). Then, these selection

results are logical `or`'d together such that the result of all the `or`'s will be the final loaded result (line 33). Figure 3.3 shows the resulting hardware. Here, we show the loaded results from 4 sub-array partitions `sub_array_0` through `sub_array_3`, where `pred0` to `pred3` are the 4 corresponding predicate values and `read_val` is the final load value. This circuit is necessary because we do not want to introduce additional control flow into the IR.

Figure 3.3: Select and or tree structure for 4 partitions.

It is worthwhile to explain the detailed behaviour surrounding predicated memory accesses, arbitration, and partitioning. Consider a four-thread program, where initially all threads access a single array. When synthesized to hardware, contention and delays due to arbitration may arise because the array, implemented as a logical RAM in the FPGA, has four requesters and at most two ports.

To resolve the contention, assume the array is partitioned into four pieces, where each thread accesses the partitions in a near-exclusive fashion, for example, each thread accesses one partition 95% of the time, but 5% of the time accesses one of the other three partitions. In such a scenario, the hardware implementation of each thread

does indeed issue a predicated access to all four partitions; however, it is *only* the access for which the predicate is true that is sent to the arbiter. That is, the accesses associated with false predicates do not create contention in our implementation.

Listings 3.4 and 3.5, show the changes in the IR in accordance to the algorithm described above. Notice that the original GEP instruction in Listing 3.4 line 5 has been replicated for both partitions in this example as lines 6 and 9 in Listing 3.5, and the index translation instructions are on lines 4 and 7. The partition number is computed on line 10 of Listing 3.5 and the predicate on lines 11 and 16. Since the predicate is only referenced in the metadata, which can be optimized away by LLVM, the predicate may also be optimized away. In order to preserve the metadata and the predicate associated with an instruction, we call upon the dummy function `__legup_preserve_value` and pass in the predicate as an argument. LLVM will not optimize away function calls, and thus it will not optimize away the predicate as it is an argument to the function call. We call upon the `-restore-metadata` LegUp-specific pass to restore any metadata that has been optimized away by LLVM since it is able to get information about the metadata through examining the `__legup_preserve_value` function calls. The LegUp backend recognizes the dummy function call and will not schedule it so there is no impact to the hardware performance. The extra calls to the function `__legup_preserve_value_*` are dummy calls to preserve the predicates, they are on lines 12, 17. Since each function has a unique function prototype, if we want to preserve multiple types of variables, we would need to create multiple `__legup_preserve_value_*` function prototypes. Hence we create a unique signature based on the variable type the function is preserving, represented by `_*`. Notice that the new loads on lines 14 and 19 of Listing 3.5 now have metadata associated to them, which point to the predicate value computed in prior instructions. Metadata in the LLVM IR appears at the end of the instruction preceded by a `!`. Finally, the predicated load values are selected based on the predicate, and logical `or`'d on lines 20 and 21 in Listing 3.5.

Listing 3.4: LLVM IR prior to array partitioning pass

```

1 ; <label>:9                                ; preds = %9, %.preheader
2 %sum.12 = phi i32 [ %sum.04, %.preheader ], [ %11, %9 ]
3 %j.01 = phi i32 [ 0, %.preheader ], [ %12, %9 ]
4 // addr computation
5 %scevgep = getelementptr [128 x [128 x i32]]* @input, i32 0, i32 %8, i32 %j.01
6 // load
7 %10 = load i32* %scevgep, align 4, !tbaa !7
8 %11 = add nsw i32 %10, %sum.12
9 %12 = add nsw i32 %j.01, 1
10 %exitcond1 = icmp eq i32 %12, 128
11 br i1 %exitcond1, label %13, label %9

```

Listing 3.5: LLVM IR after application of array partitioning pass

```

1 ; <label>:9                                ; preds = %9, %.preheader
2 %sum.12 = phi i32 [ %sum.04, %.preheader ], [ %14, %9 ]
3 %j.01 = phi i32 [ 0, %.preheader ], [ %15, %9 ]
4 %and_1 = and i32 %8, 63
5 // addr computation for partition 0
6 %GEP_sub_input_0_2 = getelementptr [64 x [128 x i32]]* @sub_input_0, i32 0, i32 %and_1, i32 %j.01
7 %and_3 = and i32 %8, 63
8 // addr computation for partition 1
9 %GEP_sub_input_1_4 = getelementptr [64 x [128 x i32]]* @sub_input_1, i32 0, i32 %and_3, i32 %j.01
10 %rshift_5 = ashr i32 %8, 6
11 %10 = icmp eq i32 %rshift_5, 0, !legup.ap_pred_id !7
12 call void @__legup_preserve_value_0_1(i1 %10), !legup.ap_pred_id !7
13 // predicated load to partition 0
14 %Load_sub_input_0_7 = load i32* %GEP_sub_input_0_2, !legup.ap_pred !7
15 %11 = select i1 %10, i32 %Load_sub_input_0_7, i32 0
16 %12 = icmp eq i32 %rshift_5, 1, !legup.ap_pred_id !8
17 call void @__legup_preserve_value_0_1(i1 %12), !legup.ap_pred_id !8
18 // predicated load to partition 1
19 %Load_sub_input_1_9 = load i32* %GEP_sub_input_1_4, !legup.ap_pred !8
20 %13 = select i1 %12, i32 %Load_sub_input_1_9, i32 0
21 %Or_10 = or i32 %11, %13
22 %14 = add nsw i32 %Or_10, %sum.12
23 %15 = add nsw i32 %j.01, 1
24 %exitcond1 = icmp eq i32 %15, 128
25 br i1 %exitcond1, label %16, label %9

```

3.1.4 Non-predicated Array Partitioning Implementation

It is possible to implement an array partitioning pass without using predicated memory accesses, however, the resultant circuit is usually highly undesirable due to the performance overheads associated. These performance issues are discussed here to illustrate the importance of using predication.

Instead of predication, partitioned array accesses can be achieved through introducing additional control flow. This means, instead of creating multiple load or store instructions associated with each partition and letting these resolve at run-time using

the predicate value, we instead use switch statements to determine which partition to load from or store to. The control flow graph of the original design will then be altered, as elaborated below.

Figure 3.4: CFG transformation of *matrixmult* benchmark

Consider the following example, a basic block B_0 contains two load instructions and are to be modified to access an array divided into 8 partitions. The partitioning pass will create a basic block B_1 , which contains all the instructions of the original basic block B_0 up to the first load instruction. Then, the new GEP instructions associated with each partition, the instruction to compute the run-time partition number, p_d , and the switch statement to determine the branch to take using the partition number are created at the end of B_1 . Each branch of the switch statement jumps

to a newly created basic block, B_2 to B_9 , which contain the single load instruction from the different sub-arrays that the program will require at run-time. All branches of the switch statement will then converge back to a basic block B_{10} , which contains all the instructions of the original B_0 after the first load instruction up to the second load instruction, with the addition of a `phi` instruction to select the correct loaded value after branching. The `phi` instruction is necessary as a result of LLVM IR being of single static assignment form; it is used when a value can take on multiple possible values depending on which control path the execution took. Refer to Section 2.2.1 for a more detailed description of the `phi` instruction. The same process is repeated for the second load instruction.

Figure 3.4 visually shows the transformation on the CFG previously described, it is taken from the kernel function for the *matrixmult* benchmark. Each node in the CFG represents a different basic block and the directed edges between nodes show the possible control flow paths. Figure 3.4a shows the CFG of the kernel function before the array partitioning pass, where the B_0 basic block described is the inner-loop basic block shown. Figure 3.4b shows the CFG of the kernel function after the array partitioning pass. The numbers next to the basic blocks denote the number of scheduled cycles for each basic block after LegUp HLS. The inner loop basic block contains two loads, one from `matrixA` and one from `matrixB`. Notice that after the transformation, the original inner loop basic block breaks up into multiple basic blocks with divergent control flows representing loads from different sub-array partitions.

A few problems arise from this implementation. Firstly, branching to different basic blocks incurs an additional cycle for each basic block transition. Note that this would occur even if the memory operations were independent of one another, the sequential ordering of these operations is strictly due to the insertion of control flow. In general, the generic compiler passes will not be able to transform the control flow in a way that could enhance throughput of memory requests. Secondly, multiple loads that could originally be scheduled in the same cycle, can no longer be scheduled

in the same cycle since the newly introduced control flow enforces an ordering of the multiple accesses. Therefore, the intrinsic parallelism in their accesses is lost. Finally, control flow limits the possibility of loop pipelining since LegUp loop pipelining does not currently support code containing divergent control flow. In some cases, these effects may not be completely devastating to performance, especially when they can be compensated by increasing the number of threads large enough to reduce the effects of the overhead. However, in many cases we need a thread count which is unrealistic due to area costs. Overall, this is not the preferred approach for partitioning arrays.

3.1.5 Limitation on Partitioning Configurations

Figure 3.5: Visualization of array under multiple partition schemes, partition number on left and index number on right.

In order to access a memory location in the newly partitioned memory scheme, the new access will require knowledge of both the new partition number, as well

as a new index into the partition where the memory location now resides. The computations needed to obtain the new partition number and index contain expensive operations such as integer division, modulo, and multiplication by arbitrary values, derived below. These costly operations cause the partitioning of arrays by arbitrary block sizes and/or by arbitrary number of partitions to be infeasible for meeting realistic performance goals. Therefore, we need to restrict the block sizes and number of partitions supported for practicality reasons. In this section, we will first define the equations needed to perform computations for the new partition number and index, and then we will declare the restrictions on partitioning configuration values.

To generalize the equations for finding the new partition number and index, let us individually examine the cases of block and cyclic partitioning shown in Figure 3.5. In this figure, we show the partitioning of an array which contains 17 elements. Each cell represents an element in the array. The numbers to the left of the array element denote the partition number. The numbers to the right of the array elements denote the index of the element in the sub-array partition it belongs.

Firstly, we look at the block partitioning case, where we divide up the array into 3 partitions. To compute the new partition number of the element, we would need to perform an integer division of the index by the block size, 6 in this case.

$$p_d = \left\lfloor \frac{i_d}{b} \right\rfloor \quad (3.1)$$

where i_d is the original index in the dimension of partitioning, d , and b is the block size. To compute the new index number within the sub-array of the element, we would need to perform a modulo operation of the index by the block size.

$$new_i_d = i_d \% b \quad (3.2)$$

Secondly, we look at the cyclic partitioning case, where we again divide up the array into 3 partitions. To compute the new partition number of the element, we

would need to perform a modulo operation of the index by the number of total partitions, 3 in this case.

$$p_d = i_d \% n \quad (3.3)$$

where n is the total number of partitions. To compute the new index number within the sub-array of the element, we would need to perform integer division operation of the index by the number of total partitions.

$$new_i_d = \left\lfloor \frac{i_d}{n} \right\rfloor \quad (3.4)$$

Finally, the block cyclic partitioning case combines the ideas of block and cyclic partitioning. In fact, block cyclic partitioning is the general case of block and cyclic partitioning. Block partitioning is block cyclic partitioning when the repetition factor, $r \geq S$, where S is the size of the array in the dimension being partitioned. Cyclic partitioning is block cyclic partitioning when the block size is 1. To compute the new partition number of an element, we need to first divide the index by the block size, 2 in this case, and then perform a modulo operation on that value by the number of partitions, 3 in this case.

$$p_d = \left\lfloor \frac{i_d}{b} \right\rfloor \% n \quad (3.5)$$

Similarly, to compute the new index, we also need to combine the ideas of block and cyclic partitioning.

$$new_i_d = (i_d \% b) + \left\lfloor \frac{i_d}{b \times n} \right\rfloor \times b \quad (3.6)$$

The indices into the other dimensions of the array remain unchanged since the other dimensions were not partitioned.

Notice that in the case of *complete* partitioning, b is 1 and n is S , the size of the array in dimension d of *complete* partitioning. Therefore, the equation to determine the new partition number is simply,

$$p_d = i_d \quad (3.7)$$

Table 3.1: Partition configuration set of allowable value ranges for number of partition, block size, and repetition factor parameters.

Scheme	Number Partitions	Block Size	Repetition Factor
Complete	$N_{cmp} = \{n n = S\}$	$B_{cmp} = \{b b = 1\}$	$R_{cmp} = \{r r = S\}$
Block	$N_b = \{n 1 < n < S\}$	$B_b = \{b 1 < b < S, b \in \{2^i i \in \mathbb{N}\}\}$	$R_b = \{r r \geq S\}$
Cyclic	$N_c = \{n 1 < n < S, n \in \{2^i i \in \mathbb{N}\}\}$	$B_c = \{b b = 1\}$	$R_c = \{r r \geq S\}$
Block Cyclic	$N_{bc} = \{n 1 < n < S, n \in \{2^i i \in \mathbb{N}\}\}$	$B_{bc} = \{b 1 < b < S, b \in \{2^i i \in \mathbb{N}\}\}$	$R_{bc} = \{r r \geq S\}$

and the new index is simply equal to 0.

Observe that the above equations heavily rely on the division, modulo, and sometimes multiplication operators. These operations are costly from both the performance and area perspectives in LegUp-generated hardware. For example, in the division/modulo module LegUp instantiates, the number of cycles required is proportional to the bitwidths of the division/modulo operands. This implies that the cycle-count penalty incurred to compute partition number and index can overcome any cycle-count reduction afforded by partitioning in the first place.

Consequently, we limit the solution space such that the division operations can be substituted by right shift operations and the modulo operations can be substituted by bit-wise logical-AND operations, which are both trivial in hardware. Table 3.1 shows the values of number partitions (n), block size (b), and repetition factor (r) that each partitioning scheme allows. There is only one possible setting for complete partitioning. In block partitioning, we can have any integer value greater than 1 and less than S number of partitions and our block size must be a power-of-2 integer, greater than 1 and less than S , where S is the size of the array in the dimension of partitioning. In cyclic partitioning, we need a power-of-2 number of partitions which is greater than 1 and less than S and a block size of 1. In block cyclic partitioning, we must have a number of partitions and block size which are both power-of-2 integers greater than 1 and less than S . Under these constraints, equations 3.1 to 3.6 will simplify to the following equations.

For block partitioning:

$$p_d = i_d \gg \log_2(b) \quad (3.8)$$

$$new_i_d = i_d \ \& \ (b - 1) \quad (3.9)$$

For cyclic partitioning:

$$p_d = i_d \ \& \ (n - 1) \quad (3.10)$$

$$new_i_d = i_d \gg \log_2(n) \quad (3.11)$$

For block cyclic partitioning:

$$p_d = i_d \gg \log_2(b) \ \& \ (n - 1) \quad (3.12)$$

$$new_i_d = (i_d \ \& \ (b - 1)) + (i_d \gg \log_2(b \times n) \ll \log_2(b)) \quad (3.13)$$

3.1.6 Exploration Space

In this section, we present equations to quantify the number of possibilities of partitioning for any given program containing any number of multi-dimensional arrays.

The total number of partitioning schemes that may be applied to a single multi-dimensional array is:

$$NUM_{total} = NUM_{cmp} + NUM_b + NUM_c + NUM_{bc} \quad (3.14)$$

where NUM_{cmp} , NUM_b , NUM_c , and NUM_{bc} represent the number of solutions for *complete*, *block*, *cyclic*, and *block cyclic* partitioning, respectively. Note that we do not include the null option of not performing partitioning. We observe that:

$$NUM_{cmp} = D \quad (3.15)$$

where D is the number of dimensions of the multi-dimensional array; that is, for each array dimension, there is a single *complete* partitioning.

We also need to define a function Φ that finds the largest power-of-2 number less than the input value. With reference to the variables in Table 3.1, we have:

$$NUM_b = |B_b| = \sum_{d=0}^{D-1} \log_2(\Phi(S_d)) \quad (3.16)$$

, where S_d is the number of elements in the d^{th} array dimension. That is, the solution-set size for block partitioning is equal to the cardinality of the set of allowed values for the block size (in each dimension we have the option to choose any power-of-2-sized block size). For cyclic:

$$NUM_c = |N_c| = \sum_{d=0}^{D-1} \log_2(\Phi(S_d)) \quad (3.17)$$

Likewise, the solution-set size for cyclic partitioning is equal to the cardinality of the set of allowed values for the number of partitions (in each dimension we have the option to choose any power-of-2 number of blocks). For block cyclic partitioning:

$$NUM_{bc} = \sum_{d=0}^{D-1} \sum_{i=1}^{\log_2(\Phi(S_d))-1} \log_2(\Phi(S_d)) - i \quad (3.18)$$

The outer sum is over all dimensions. The inner sum is the solution space size for each dimension. The inner sum index i corresponds to the legal block sizes for block cyclic partitioning: the block sizes are 2^i , and therefore range from 2^1 to $2^{\log_2(\Phi(S_d))-1}$ in powers-of-2. The reason that block sizes of $2^0 = 1$ and $2^{\log_2(\Phi(S_d))}$ are excluded from the sum bounds is that these solutions are standard cyclic, and block partitioning, respectively (already counted above in the NUM_c and NUM_b terms). For each legal block size, $\log_2(\Phi(S_d)) - i$ counts the number of ways the block size can be assigned to banks (number of partitions).

Figure 3.6 illustrates the possible block cyclic partitioning schemes for the 17 element array shown in Figure 3.5. Since this is a 1-dimensional array, we can ignore

Figure 3.6: Possible block cyclic partitioning schemes for a 17 element array.

the outer sum in equation 3.18. Focusing on the inner sum, we know that S_d is 17 in this case, and, recalling that the Φ function returns the largest power-of-2 number less than the input value, we can elaborate the inner sum as $3 + 2 + 1 = 6$. Each of these terms is the size of the solution space for a given block size. There are three possibilities for partitioning when we set the block size as 2, there are two possibilities for partitioning when we set the block size as 4, and one possibility for a block size of 8, block sizes larger than 8 (i.e. 16) would not be block cyclic partitioning since the block size is too large to allow for recurrence of blocks in the partitioning.

To better understand this equation, recall that the smallest allowable block size is 2. The largest allowable block size is the largest power-of-2 number less than half of the size of the array in that dimension, which, in our case is 8. Note that a block size any larger than 8 will result in a non-block cyclic partitioning configuration. The number of partition schemes available when the block size is 2 is $\log_2(\Phi(S_d)) - 1$. This equation can be interpreted as answering the following question: How many possible valid number of partitions are available when the block size is 2? It is also equivalent to finding the number of block partitioning schemes when the array dimension is divided by 2, the block size. Thus, we subtract 1 from the equation to determine the number of block partitioning schemes to account for the block size of 2 constraint. Similarly, we subtract 2 for when the block size is 4, and subtract 3 for when the block size is 8.

Consider the simple case of a 2-dimensional array of 33×16 elements, D in this case is 2, S_0 is 16 and S_1 is 33. Let us take a look at the row dimension (dim 1), $|P_{cmp}| = 1$, $|P_b| = 5$, $|P_c| = 5$, and $|P_{bc}| = 10$. If we elaborate the block cyclic set we get $|P_{bc}| = 4 + 3 + 2 + 1$, where each term corresponds to the number of solutions for when the block size is 2, 4, 8, and 16, respectively. If we look in the column dimension (dim 0), $|P_{cmp}| = 1$, $|P_b| = 3$, $|P_c| = 3$, $|P_{bc}| = 3$. In total, there are 32 ways to partition this matrix if we also include the option to not perform partitioning.

The above equations 3.15, 3.16, 3.17, and 3.18 compute the solution-set size for a single array. When there are multiple arrays within the same program, the exploration

space becomes a Cartesian product of all the possible partitioning schemes of each array, which may become very large even for moderately sized arrays. For example, if we now consider the matrix multiplication of two 2-dimensional arrays where the sizes are 33×16 and 16×33 , then we potentially have $32 \times 32 = 1024$ partitioning possibilities.

3.2 Array Partitioning Evaluation

3.2.1 Benchmark Descriptions

The evaluation of array partitioning on circuit performance, as well as the accuracy of the memory simulator were performed using 8 multi-threaded benchmarks: *matrixadd*, *histogram*, *matrixmult*, *matrixmult (cyclic)*, *matrixtrans*, *matrixtrans (block cyclic)*, *substring*, and *los*.

Matrixadd performs the summation of the elements of a 128×128 input integer matrix, where each thread is responsible for the summation of a group of rows. *Histogram* reads a 32768 element input array and counts the number elements which belong to a certain value range into a local array. *Matrixmult* performs matrix multiplication on two 32×32 input matrices – `matrixA` and `matrixB`, where each thread is responsible for computing the results of a group of rows. *Matrixmult (cyclic)* is a variation on the *matrixmult* benchmark, where each thread is responsible for computing the output for each row in a cyclic manner, where thread 0 will compute the outputs of row 0, 8, 16, etc. Additionally, *matrixmult* was modified such that each thread accesses *matrixB* with an offset to a different set of columns to allow for parallelization of accesses to columns. *Matrixtrans* performs matrix transpose on a 128×128 input matrix, where each thread transposes each row of the input for a contiguous slice of columns into the output. For example, thread 0 is responsible for transposing all rows between columns 0 - 15 for the input matrix. *Matrixtrans (block cyclic)* is a variant of the *matrixtrans* benchmark, where each thread transposes a group of rows of the

input for all columns into the output in a block cyclic manner. For example, thread 0 is responsible for transposing rows 0-3, 32-35, 64-67, and 96-99 for all columns of the input matrix. *Substring* is a benchmark which takes in a DNA subsequence of size 3 and performs substring matching against a larger DNA sequence of size 16384 to determine the number of matches. Each thread performs substring matching on a contiguous group of the string. *Los* (line of sight) consists of a 64×64 obstacle map in which each pixel can take on the values of either 1 or 0, where 1 represents an obstacle and 0 represents unoccupied space and a human is assumed to be situated in the middle of the obstacle map. The threads will sweep through the obstacle map at each map coordinate and steps in the direction of the human located at the centre to determine whether there is an obstacle between the beginning location and the human. If there is, then that coordinate location is not within the line of sight of the human and thus will be marked as 0 on the output map, otherwise it will be marked 1. This benchmark is input-dependent and is not intuitive in terms of which partitioning scheme is the most optimal. Each benchmark contains the input vectors in the benchmark itself. With the exception of *substring* and *los*, all the benchmarks are input-independent. This means that the performance will not change depending on the contents of the input, it is only a function of the size of the input matrices. While most of the benchmarks presented exhibit highly regular access patterns, these chosen applications are commonly the targets of HLS tools. In future work, array partitioning performance benefits can be demonstrated on graphing applications such as triangle counting, where sets of 3 adjacent vertices in a graph G is counted; PageRank, the algorithm used for ranking webpage relevance for search engines such as Google, and others. Error checking is performed against a golden output for each benchmark at the end of the test to ensure functional correctness.

3.2.2 Performance Evaluation

In the performance evaluation of our array partitioning pass, we run each benchmark with 8 threads across all partitioning schemes, where the number of partitions does not exceed 16. Since the exploration space is very large for benchmarks containing more than one array, we manually prune the evaluation space such that we end up evaluating the combinations of partitioning schemes that will most likely lead to the best performance.

Figure 3.7: Matrix Access Patterns for *matrixmult* and *matrixtrans*.

In particular, we restrict the number of partitioning combinations for the arrays in the original and variations of the *matrixmult* and *matrixtrans* benchmarks. This is because the accesses to the two matrices of interest in these benchmarks, **matrixA** and **matrixB** in *matrixmult*, and **input_array** and **output** in *matrixtrans*, are in such a way where the first matrix is accessed in a row-major fashion and the second matrix is accessed in a column-major fashion shown in Figure 3.7. This means, if a certain partitioning scheme leads to the best performance for one matrix, then the same partitioning scheme should be applied to the other matrix in the other dimension – i.e. apply row partitioning to column, or column to row.

The notation used for the partitioning scheme for the rest of the paper is as follows:

$$\langle dimension\ of\ partition \rangle \langle partition\ type \rangle \langle n \rangle [-\langle b \rangle]$$

where *dimension of partition* labels each dimension of a multi-dimensional matrix starting from 0, the rightmost dimension in the array declaration; *partition type* is one of the following **, b, c, bc*, these stand for *complete*, *block*, *cyclic*, or *block cyclic*.

For example, if we wanted to partition an $M \times N$ matrix into 2 blocks, where the first block contains the elements in all the rows for columns in range 0 to $N/2 - 1$ and the second block contains the elements in all the rows for columns in range $N/2$ to $N - 1$, then the partitioning scheme would be *0b2*, which stands for block partition into 2 blocks in the column dimension.

Table 3.2: Baseline unpartitioned versus best partition scheme(s) execution cycles.

benchmark	parallel region cycles		last memory cycle	cycles speedup	partition schemes
	unpartitioned	'best' case partitioned			
matrixadd	16396	4157	4163	3.94	1b4, 1b8, 1b16, 0c16, 0bc8_2, 0bc16_2, 1bc4_16, 1bc8_8, 1bc16_4
histogram	33077	24013	23987	1.38	0b8, 0b16
matrixmult	32839	16799	16793	1.95	0b16, 1b4, 1b8, 1b16, 0c8, 0c16, 1c16, 0bc4_2, 0bc8_2, 1bc4_4, 1bc8_2
matrixmult (cyclic)	32837	16794	16789	1.95	0b16, 1b16, 0c8, 0c16, 1c4, 1c8, 1c16, 0bc4_2, 0bc8_2, 1bc4_2, 1bc8_2
matrixtrans	16395	6423	6419	2.55	0b4, 0b8, 0b16, 0c8, 0c16, 0bc4_2, 0bc8_2, 0bc8_4, 0bc8_8, 0bc16_2, 0bc16_4
matrixtrans (block- cyclic)	16482	6348	6344	2.60	0c8, 0c16, 1c16, 1b16, 0bc4_2, 0bc8_2, 0bc16_2, 1bc4_4, 1bc4_8, 1bc8_2, 1bc8_4, 1bc8_8, 1bc16_2, 1bc16_4
substring	16396	4120	4117	3.98	0c4, 0c8, 0c16, 0b8, 0b16, 0bc8_2, 0bc8_1024, 0bc16_2, 0bc16_512
los	83217	79543	79537	1.05	1bc16_2

First, we present the execution cycle count reduction results in Table 3.2. In these results, we only consider the parallel kernel function execution. The second column shows the cycle count for the baseline unpartitioned case; the third column shows the cycle count of the partitioned cases leading to the lowest cycle count; the fourth column shows the cycle of the last memory access to the arrays under partitioning; the fifth column shows the cycle count speedup of the partitioned over the unpartitioned case; and column six shows all the partitioning scheme(s) that led to the best cycle count. In *matrixmult*, *matrixmult (cyclic)*, *matrixtrans*, and *matrixtrans (block cyclic)* the benchmarks each contain two matrices which can both be partitioned. The results presented show the partitioning schemes leading to the best cycle count for the matrix **matrixA** in *matrixmult* and *matrixmult (cyclic)*, and for the matrix

`input_array` in *matrixtrans* and *matrixtrans (block cyclic)*. Since the access patterns for the two matrices in these four benchmarks are the transpose of one another, i.e. for *matrixmult*, `matrixA` is accessed row by row, while `matrixB` is accessed column by column, the partition scheme which leads to the best cycle count in one matrix can simply be applied to the other matrix in the opposite dimension. This means, the partitioning scheme leading to one of the lowest cycle counts for *matrixadd* is *0b16* for `matrixA` and *1b16* for `matrixB`. The geomean cycle count speedup across all 8 benchmarks is $2.21\times$.

From Table 3.2, we observe that generally multiple partitioning schemes may yield the same cycle count. However, we know intuitively that partitioning schemes requiring more partitions will yield designs with lower clock frequencies and costs more area since higher numbers of partitions means more arbitration logic. We will discuss area versus performance trade-offs in the following section.

The reader may find the fact that several different partitioning schemes lead to the same cycle count to be non-intuitive, especially when some of these schemes are quite different from each other. For example, for the *matrixadd* benchmark, we see that the possible results are: *1b4*, *1b8*, *1b16*, *0c16*, *0bc8_2*, *0bc16_2*, *1bc4_16*, *1bc8_8*, *1bc16_4*. For the access pattern of *matrixadd*, we find that the expected optimal partitioning schemes *1b8* and *1b16* indeed produce the best cycle counts. However, results such as *1b4* or *0bc8_2* may not be expected. Here, we offer the reader insight into this phenomenon through analyzing the memory bandwidth requirement as well as the memory access pattern.

Figures 3.8 and 3.9 show a visualization of the accesses to the input matrix under the *1b4* and *0bc8_2* partitioning schemes respectively. This visualization may be useful to guide the reader’s understanding. The figures depict the cycles in which memory accesses are satisfied. Each coloured block represents a request. Each column represents the requests of a different requester, where requester 0 is the `main` module, and requesters 1 to 8 are the 8 threads of the kernel function. The value within the block show the partition the memory access belonged to (value before decimal point),

	requestor								
cycle	0	1	2	3	4	5	6	7	8
40		0.1		1.1		2.1		3.1	
41			0.1		1.1		2.1		3.1
42		0.1		1.1		2.1		3.1	
43			0.1		1.1		2.1		3.1
44		0.1		1.1		2.1		3.1	
45			0.1		1.1		2.1		3.1
46		0.1		1.1		2.1		3.1	
47			0.1		1.1		2.1		3.1
48		0.1		1.1		2.1		3.1	
49			0.1		1.1		2.1		3.1
50		0.1		1.1		2.1		3.1	
51			0.1		1.1		2.1		3.1
52		0.1		1.1		2.1		3.1	
53			0.1		1.1		2.1		3.1
54		0.1		1.1		2.1		3.1	
55			0.1		1.1		2.1		3.1
56		0.1		1.1		2.1		3.1	
57			0.1		1.1		2.1		3.1
58		0.1		1.1		2.1		3.1	
59			0.1		1.1		2.1		3.1
60		0.1		1.1		2.1		3.1	
61			0.1		1.1		2.1		3.1
62		0.1		1.1		2.1		3.1	
63			0.1		1.1		2.1		3.1
64		0.1		1.1		2.1		3.1	
65			0.1		1.1		2.1		3.1

Figure 3.8: *Matrixadd* array access pattern for block partitioning by 4.

and whether the request was granted by the arbiter in that cycle (value after decimal point; 1 for granted, 0 for not granted). As seen from these figures, the number of accesses satisfied at steady state in both partitioning schemes are the same.

Examining the *matrixadd* kernel, we observe that there is 1 memory access to the input matrix in each inner loop iteration. Each inner loop basic block has a schedule length of 2 cycles. This means, under no contention, each thread will issue 1 memory access instruction every 2 cycles. Therefore, with 8 threads, the memory bandwidth requirement is 4 memory accesses per cycle. Although the memory bandwidth of 4 dual-ported RAMs is 8 memory access per cycle, as we would expect, in the 1b4 partitioning case, the hardware produced only provides 4 memory accesses per cycle. This is due to an inefficiency in the current LegUp port binding algorithm in

	requestor								
cycle	0	1	2	3	4	5	6	7	8
46		7.1		5.1		4.1		2.1	
47			6.1		5.1		3.1		2.1
48		7.1		6.1		4.1		3.1	
49			7.1		5.1		4.1		2.1
50		0.1		6.1		5.1		3.1	
51			7.1		6.1		4.1		3.1
52		0.1		7.1		5.1		4.1	
53			0.1		6.1		5.1		3.1
54		1.1		7.1		6.1		4.1	
55			0.1		7.1		5.1		4.1
56		1.1		0.1		6.1		5.1	
57			1.1		7.1		6.1		4.1
58		2.1		0.1		7.1		5.1	
59			1.1		0.1		6.1		5.1
60		2.1		1.1		7.1		6.1	
61			2.1		0.1		7.1		5.1
62		3.1		1.1		0.1		6.1	
63			2.1		1.1		7.1		6.1
64		3.1		2.1		0.1		7.1	
65			3.1		1.1		0.1		6.1
66		4.1		2.1		1.1		7.1	
67			3.1		2.1		0.1		7.1
68		4.1		3.1		1.1		0.1	
69			4.1		2.1		1.1		7.1
70		5.1		3.1		2.1		0.1	
71			4.1		3.1		1.1		0.1

Figure 3.9: *Matrixadd* array access pattern for block cyclic partitioning by 8 with a block size of 2.

multi-threaded cases. This inefficiency arises because LegUp does not recognize that each thread is a separate instance during the scheduling and binding phases of the pthread function. Instead, LegUp generates hardware for one thread and replicates the hardware module multiple times to act as different thread instances. This means an opportunity is lost for further parallelizing between threads. For example, if we have 2 instances of a pthread function, where the bandwidth required for each thread is 1 access per cycle to the same thread, LegUp will bind the memory accesses of both of the instances to the same port, say port a. In reality, the memory bandwidth can be doubled if LegUp realized that it can schedule the accesses of one thread to port a and the accesses of the other thread to port b. For the same reason, under the 1b4

partitioning scheme, the design can only provide a memory bandwidth of 4 memory accesses per cycle.

Partitioning under the *0bc8_2* scheme provides double the required memory bandwidth for the benchmark at 8 memory accesses per cycle. However, consider the accesses shown in Figure 3.9, if we reduce the number of partitions to 4 (i.e. *0bc4_2*), there would be contentions given the fact that all the elements belonging to partitions 4, 5, 6, and 7 would map to partitions 0, 1, 2, and 3. In this case, the memory access is the limiting factor.

3.2.3 Performance and Area Trade-off

We use Quartus II v15.0 to synthesize the LegUp-generated designs to obtain the area and timing results. We target the Cyclone V FPGA and chose the 5CGXFC9E6F35C7 device since it has a larger amount of dedicated memory compared to other devices in the Cyclone V family. As a representative example, we examine the *matrixadd* and *histogram* benchmarks in detail and assess its performance vs. area trade-offs. The detailed results for all 8 benchmarks can be found in Appendix A.

Matrixadd Performance vs Area Results

Table 3.3 shows the area and clock frequency results after synthesis for the partitioning schemes leading to the best cycle count found through ModelSim simulation. The synthesis results shown are the average of 5 runs with different seeds. Again, the cycle counts presented account for the parallel kernel code regions only. We also present the computed wall-clock time as the product of cycle count and clock period from synthesis. The normalized area-delay product is calculated using the relative area scaling factors presented in [36]. Rashid et al. [36] presents the area scaling factors for the Stratix V family of FPGAs normalized to the tile area used for each Adaptive Logic Module (ALM), thus each scaling factor is presented as equivalent

Table 3.3: Cycle execution performance versus area results for *matrixadd* benchmark.

Partition Scheme	cycles	fmax [MHz]	wall-clock time [μ s]	wall-clock time speedup	ALMs	registers	membits	RAMs	DSPs	normalized area-delay product
none	16396	93.35	175.64	1.00	2688.8	4025.6	525312.0	68.0	0.0	1.00
1bc4_16	4157	89.61	46.39	3.79	3754.2	4931.2	525312.0	68.0	0.0	0.33
1b4	4157	82.65	50.3	3.49	3850.2	5141.6	525312.0	68.0	0.0	0.37
1bc8_8	4157	87.11	47.72	3.68	5138.6	6135.4	525312.0	60.0	0.0	0.43
0bc8_2	4157	85.45	48.65	3.61	5252.4	6110.2	525312.0	60.0	0.0	0.44
1b8	4157	82.57	50.35	3.49	5283.0	6349.8	525312.0	60.0	0.0	0.46
1bc16_4	4157	82.00	50.69	3.46	8011.2	8503.8	525312.0	68.0	0.0	0.67
0c16	4157	81.46	51.03	3.44	8479.4	8467.4	525312.0	68.0	0.0	0.71
0bc16_2	4157	81.75	50.85	3.45	8473.4	8468.0	525312.0	68.0	0.0	0.71
1b16	4157	79.12	52.54	3.34	8354.0	8713.6	525312.0	68.0	0.0	0.72

ALMs. The Stratix V family is of the 28nm process technology, as is the Cyclone V family for which we are presenting the area results. The scaling factor presented in the paper for the M20K memory is 40, which we have reduced linearly to 20 for our M10K memories. The DSP scaling factor presented is 30. The area-delay product is normalized to the baseline unpartitioned case.

It is also important to point out that partitioning leads to diminishing returns in terms of cycle count, but may incur significant area costs. Figure 3.10a and Figure 3.10b show the performance and area of the *matrixadd* benchmark, respectively, under block cyclic partitioning in the column dimension (dim 0) for various numbers of partitions and a block size of 2. Figure 3.12a and Figure 3.12b show the performance and area of the *matrixadd* benchmark, respectively, under block partitioning in the row dimension (dim 1) for various numbers of partitions. These two configurations were chosen as the exemplar because they encompass two of the partition schemes (1b4 and 0bc8_2) that lead to both the second and fourth best area-delay results while containing enough data points (number of partitions available in the dimension of partitioning) to show the area-performance trend.

For the block cyclic partitioning cases, observe that the cycle count reaches a minimum for 8 and 16 partitions, however, the clock frequency reduces between 8

(a) Performance result comparing total execution cycles of benchmark, as well as the number of resulting contentions and wall-clock time computed using the Fmax achieved.

(b) Area result comparing total number of ALMs, registers, block RAMs, as well as the Fmax achieved after static timing analysis.

Figure 3.10: Performance and area comparison for *matrixadd* benchmark with block cyclic partitioning across different number of partitions with block size 2.

Figure 3.11: Normalized area-delay product for *matrixadd* benchmark with block cyclic partitioning across different number of partitions with block size 2.

partitions to 16 partitions from 85.45 MHz to 80.83 MHz while the area usage increases as we expect. For the block partitioning cases, observe that the execution cycles reach a minimum for 4, 8, and 16 partitions, however, the clock frequency reduces slightly from 82.65 MHz at 4 partitions to 79.12 MHz at 16 partitions, and the area usage also increases as we expect. Figure 3.11 and Figure 3.13 plot the normalized area-delay product for the block cyclic and block partitioning cases. In Figure 3.11, we see the area-delay product for block cyclic partitioning along dimension 0 (columns) with block size of 2 and various values for the number of partitions, normalized to the baseline unpartitioned case. Observe that the minimum area-delay product occurs at around 4 and 8 partitions, after which, the area-delay product sharply increases due to the high hardware cost and stagnant performance. Similarly, in Figure 3.13, we see the normalized area-delay product for block partitioning along dimension 1 (rows). Observe that the minimum area-delay product occurs at 4 partitions, after

(a) Performance result comparing total execution cycles of benchmark, as well as the number of resulting contentions and wall-clock time computed using the Fmax achieved.

(b) Area result comparing total number of ALMs, registers, block RAMs, as well as the Fmax achieved after static timing analysis.

Figure 3.12: Performance and area comparison for *matrixadd* benchmark with block partitioning across different number of partitions.

Figure 3.13: Normalized area-delay product for *matrixadd* benchmark with block partitioning across different number of partitions.

which, the area-delay product increases once again since more partitioning does not improve the performance but only adds to the area cost.

In both partitioning schemes, notice that the RAM usage at 8 partitions reaches a minimum even while the memory bits required across all partitioning schemes was consistent, 525312 bits. The resource usage breakdown of the unpartitioned case is shown in Table 3.4, and the partition-by-8 case in Table 3.5. Notice that in order to store 16384 elements (our input matrix) in the unpartitioned case, we would need 52 M10K blocks, however, 64 are allocated.

Table 3.4: M10K usage for unpartitioned *matrixadd* benchmark

Array Name	Port Depth	Port Width	M10K blocks
main_preheader210	16	64	4
input_var0	16384	32	64

There is a reduction in the number of M10K blocks used when we compare the

Table 3.5: M10K usage for partitioning by 8 blocks *matrixadd* benchmark

Array Name	Port Depth	Port Width	M10K blocks
main_preheader210	16	64	4
sub_input_0_inst	2048	32	7
sub_input_1_inst	2048	32	7
sub_input_2_inst	2048	32	7
sub_input_3_inst	2048	32	7
sub_input_4_inst	2048	32	7
sub_input_5_inst	2048	32	7
sub_input_6_inst	2048	32	7
sub_input_7_inst	2048	32	7

resource usage across different partition numbers. In the case when there are 8 partitions, there are 60 M10K blocks used as opposed to 68. This is likely due to the heuristic based algorithm used in the synthesis tool.

Histogram Performance vs Area Results

Table 3.6: Cycle execution performance versus area results for *histogram* benchmark.

Partition Scheme	cycles	fmax [MHz]	wall-clock time [μ s]	wall-clock time speedup	ALMs	registers	membits	RAMs	DSPs	normalized area-delay product
none	33077	91.14	362.92	1.00	3430.8	5401.6	1048576	128.0	0.0	1.00
0b8	24013	73.56	326.45	1.11	5548.4	7420.6	1048576	128.0	0.0	1.22
0b16	24013	71.62	335.3	1.08	8231.8	9755.4	1048576	112.0	0.0	1.61

Table 3.6 shows the area and clock frequency results after synthesis for the partition schemes leading to the best cycle count found through ModelSim simulation. The synthesis results are shown for the average of 5 runs with different seeds. Again, the cycle counts presented account for the parallel kernel code regions only. We also present the computed wall-clock time as the product of cycle count and clock period from synthesis. The normalized area-delay product is calculated in the same manner as for the *matrixadd* benchmark.

From these results, we observe that the normalized area-delay product of both partitioned results are higher than in the unpartitioned case. If we look at the case

of the *0b8* partitioning scheme, we will see that the wall-clock time speedup was 1.11. In this case, the cycle count improved by 28%, while the Fmax decreased by 19%. Fmax decreased due to increased arbitration logic that is needed for 8 partitions. The modest improvement in wall-clock time was therefore not able to counter-balance the increased area usage.

Figure 3.14a and Figure 3.14b show the performance and area of the *histogram* benchmark, respectively, under block partitioning in the column dimension (dim 0) for various numbers of partitions.

Observe that the cycle count reaches a minimum for 8 and 16 partitions. However, the improvement in cycle count between 2 partitions and 16 partitions is only 5.03% while the Fmax decreased by 11.5%. This example shows the diminishing returns effect of partitioning, which occurs when increasing partitions does not improve cycle count significantly, but may lead to major degradation of the clock frequency.

Figure 3.15 shows the associated normalized delay-area product for these partitioning schemes. Here, the lowest area-delay product occurs at block partitioning of 2. In this case, the partitioning was able to significantly improve the cycle count, while the area usage increase and Fmax degradation was not large enough such that the area-delay product exceeded the unpartitioned case.

3.3 Summary

In this chapter, we described the implementation of the array partitioning mechanism in LegUp. The array partitioning mechanism, implemented as an LLVM optimization pass is able to perform array partitioning on multi-dimensional arrays in one of its dimensions in one of four ways: *complete*, *block*, *cyclic*, and *block cyclic*. We showed a performance and area evaluation on 8 benchmarks and discussed the limitations of our implementation and the practical impacts on the number of partitioning configurations considered. Lastly, we provided a detailed area versus performance study of two benchmarks, *matrixadd* and *histogram*.

(a) Performance result comparing total execution cycles of benchmark, as well as the number of resulting contentions and wall-clock time computed using the Fmax achieved.

(b) Area result comparing total number of ALMs, registers, block RAMs, as well as the Fmax achieved after static timing analysis.

Figure 3.14: Performance and area comparison for *histogram* benchmark with block partitioning across different number of partitions.

Figure 3.15: Normalized area-delay product for *histogram* benchmark with block partitioning across different number of partitions.

Chapter 4

Automatic Memory Partitioning

The previous chapter describes an array partitioning pass capable of performing array partitioning on globally declared arrays and gives detail on the possible partitioning schemes available. A study is also presented showing the performance benefits of array partitioning, as well as the associated area costs. These results show us that array partitioning is an important optimization to decrease the effects of the memory bottleneck on our multi-threaded programs when they are transformed into hardware circuits. In this chapter, we present a tool that uses dynamic profiling data to advise designers on the best choices of partitioning schemes.

4.1 Automatic Memory Partitioning Tool Flow

At a high level, the automated array partitioning tool is implemented as follows:

1. The multi-threaded software program, without any partitioning applied is synthesized into hardware using LegUp HLS.
2. The generated RTL is then simulated with ModelSim under typical input vectors.

3. A memory access trace is extracted, where, for each array/memory access, the address of the access and the ID of the thread making the access is tracked. Stalls due to memory arbitration are also recorded.
4. The trace and a *hypothetical* array partitioning scheme are input to our memory simulator, which, for the proposed scheme, estimates its performance benefits.

With the single HLS execution and trace generation, we are able to assess the consequences of various partitioning schemes. And, by simulating a variety of different partitioning schemes, an optimal memory banking architecture can be determined. We elaborate on the steps in the subsections below.

4.1.1 Memory Trace Collection

We modified the RTL-generation of the LegUp HLS tool to produce the relevant trace data in ModelSim simulation. Specifically, a `$display` statement is inserted whenever a request is made to shared memory. The request signals to shared memories can easily be detected as each shared RAM/ROM requires an arbiter to arbitrate access between multiple requesters. We therefore monitor the `request_in` signal vector of the associated arbiter for each access. The information gathered for each shared memory request is: *i)* the cycle of the initial request, *ii)* the specific block RAM being requested, *iii)* the port name, *iv)* the function trying to access the RAM, *v)* the instance number, equivalent to the numerical thread ID, in the case of multi-threaded functions, and *vi)* the address of the request to be used later for determining possible partition schemes. A sample of the trace is shown in Listing 4.1 for the *matrixadd* benchmark. Line 1 shows the output message in the event when a contention is detected. The contention occurs on the RAM `input_var0` in cycle 24 and lines 2 and 3 show the actual requests that lead to the contention. Line 2 shows that the request to RAM `input_var0` on port `a` from instance 0 of the function module `add` to address 0003 was denied at cycle 24, while line 3 shows that the request to the same RAM and port by instance 2 of the function module `add` to address 1000 was granted.

Listing 4.1: Sample memory trace.

```

1 # Contention at cycle: 24 arbiter: round_robin_arbiter_inst_arbiter_input_var0_a ram: input_var0 request: 00000101
2 # Grant denied at cycle: 24 RAM: input_var0 Port: _a Function: add Instance: 0 Address: 00000000000011 (0003)
3 # Grant given at cycle: 24 RAM: input_var0 Port: _a Function: add Instance: 2 Address: 01000000000000 (1000)
4 # Grant given at cycle: 25 RAM: memory_controller Port: _a Function: add Instance: 4 Address:
   000000011000000000000000110000 (01800030)
5 # Grant given at cycle: 25 RAM: memory_controller Port: _b Function: add Instance: 4 Address:
   000000011000000000000000110100 (01800034)
6 # Contention at cycle: 25 arbiter: round_robin_arbiter_inst_arbiter_input_var0_a ram: input_var0 request: 00000011
7 # Grant given at cycle: 25 RAM: input_var0 Port: _a Function: add Instance: 0 Address: 00000000000011 (0003)
8 # Grant denied at cycle: 25 RAM: input_var0 Port: _a Function: add Instance: 1 Address: 00100000000010 (0802)
9 # Contention at cycle: 26 arbiter: round_robin_arbiter_inst_arbiter_input_var0_a ram: input_var0 request: 00000110
10 # Grant given at cycle: 26 RAM: input_var0 Port: _a Function: add Instance: 1 Address: 00100000000010 (0802)
11 # Grant denied at cycle: 26 RAM: input_var0 Port: _a Function: add Instance: 2 Address: 01000000000001 (1001)

```

4.1.2 Memory Simulator

The memory simulator is a program that models the arbitration of requests to various block RAMs. The program is written in Python. The simulator will, based on any partitioning scheme and an input memory trace collected during the unpartitioned execution, determine the number of port contentions that will occur, as well as the cycle number in which the last memory request will be completed. The steps required to determine the number of cycles needed to complete all accesses of the program under a particular partitioning configuration is as follows:

1. Create memory access chain(s) (defined below) to represent the input memory trace.
2. Create an internal representation of the arbiters needed for the particular partitioning scheme.
3. Iterate through the memory access chain(s) to determine the cycle of last memory access.

Memory Access Chain Generation

Figure 4.1: Memory access schedule of 2-threaded execution to memory access chain.

The memory simulator first creates a representation of the unpartitioned execution memory accesses as a chain of nodes, in which each node represents a memory access and the edge weight between the nodes denotes the *distance*, i.e. the intrinsic number

of cycles between accesses. The *distance* may not be the same as the number of cycles between accesses in an execution trace since the trace would capture cycles lost to contention. Figure 4.1 shows a series of scheduled accesses for a program running 2 threads, as well as the memory access chains that are created from the memory trace. The boxed accesses represent the requests granted in that cycle, while the unboxed accesses represent the accesses which stalled due to contention. As seen here, the cycle difference between when *access 0* is granted and when *access 1* is granted is 2. However, the *distance* will measure 1 cycle, because it captures the number of cycles between accesses in the absence of contention. The *distance* therefore, represents the inherent cycle differences between accesses in the schedule without considering run-time contentions.

When creating the memory access chain, a list of potential requester modules are input by the user to the memory simulator. These requesters are identified by their hardware module and instance numbers. A memory access chain is created for each requester since we know that the accesses within each instance of a module (thread) must be carried out in sequence.

Fundamentally, this method extracts the dynamic execution schedule of all memory accesses. Given this dynamic execution schedule, and given that partitioning does not alter the schedule, then we can accurately predict the dynamic schedules of these same memory accesses under various array partitioning configurations. In Section 4.2.3, we will discuss when the schedules of array-partitioned designs may differ from the unpartitioned case.

Scheduling Memory Access Chains

Once the memory access chain is gathered, the simulator will “execute” the trace. Given that we know the order of all memory accesses by each requester, as well as the specific addresses being accessed, we can then know which partition each access belongs to for a particular partitioning scheme “under test”. The simulator has an

in-memory model of the round-robin arbiter instantiated by LegUp. It is therefore able to assess cycles lost to contention, i.e. a requester waiting for a “grant” from the arbiter.

Listing 4.2: Algorithm for simulating a memory partitioning scheme

```

1  setQ: current pending accesses for each arbiter
2  T: memory trace
3
4  for each Q in setQ {
5      Q = {}
6  }
7  cycle = first access cycle in T
8  while (!done) {
9      done = True
10     for each Q in setQ {
11         populateQ(Q, partitionConfig, cycle, T)
12         executeCycle(Q)
13     }
14     cycle++
15     for each Q in setQ {
16         if (Q !empty || accesses remain in T) {
17             done = False
18         }
19     }
20 }
```

Listing 4.2 shows pseudocode for simulating the scheduling of a memory access trace for a particular array partitioning configuration, `partitionConfig`. Our approach bears similarity to event-driven logic simulation. Here, `setQ` is a set of queues: one for each arbiter in the partitioning scheme being simulated. Each queue stores the pending memory requests for a port on a memory bank. Initially, all arbiter queues are initialized to empty.

We commence simulation at the cycle of the first memory access (line 7) and enter the `while` loop (line 8). For each arbiter in the design (line 10), as specified by `partitionConfig`, we use the current cycle, as well as the trace `T` to populate the queue with active requests (line 11). The `populateQ` function (not shown) takes accesses from the trace `T`, which occur at time `cycle`, and for each such access, uses the partitioning scheme to create a request in a specific queue. Then, the `executeCycle` function (line 12) uses an internal arbiter model by calling the `get_next_grant` function, shown in Listing 4.3. The granted requester in each queue is found, and this

request is then removed from the queue. This process continues until there are no more requests outstanding in any arbiter-request queues and when the trace is empty (line 16). Upon termination, the `cycle` variable holds the cycle index of the last-served memory request. Partitioning schemes which reduce this quantity are generally desirable.

Internal Arbiter Representation

We created a software model of the round-robin arbiter which functions as follows:

Listing 4.3: Round robin arbiter software model

```

1  init_arbiter(arbiter):
2      (arbiter.requesterList, arbiter.numRequesters) = get_all_requesters()
3      arbiter.reqList = []
4      arbiter.prevGrant = arbiter.numRequesters - 1
5
6  get_next_grant(arbiter):
7      if (outstanding request exists):
8          contention = False
9          if (number outstanding requests > 1):
10             contention = True
11
12         # do not need to wrap around
13         if ((arbiter.prevGrant + 1) < arbiter.numRequesters):
14             for (pos = (arbiter.prevGrant + 1) : arbiter.numRequesters):
15                 if (arbiter.reqList[pos] is active request):
16                     arbiter.prevGrant = pos
17                     return get_requester_name(arbiter, pos)
18
19         # need to wrap around
20         for (pos = 0 : arbiter.prevGrant):
21             if (arbiter.reqList[pos] is active request):
22                 arbiter.prevGrant = pos
23                 return get_requester_name(arbiter, pos)

```

Each arbiter is connected to a set of requester modules. Requester modules are uniquely specified by their function name and instance number. The arbiter satisfies requests in a round-robin fashion, from the smallest requester ID to the largest requester ID. At initialization, the `requesterList`, which contains a list of the function and instance numbers associated with each of the requester modules to the resource arbitrated by each arbiter, will be populated by the `get_all_requesters` function (line 2). The initialization function also sets the `numRequesters` variable, used for tracking the number of requesters connected to the arbiter, accordingly. The `reqList`,

which contains a list of outstanding requests to the arbiter will also be cleared (line 3). The `prevGrant`, which saves information about the last requester to be given the grant is set to the last requester number (line 4). This allows the first grant to be given to the first requester since the “last seen” request was the last requester in the list.

Throughout the execution of the simulator, we insert requests into the `requestList` as we iterate through the memory trace. After a request is satisfied, it is removed from the `requestList`. The `get_next_grant` function (line 6) determines which requester will be given the grant next. The function first checks to see if there are any outstanding requests on the `reqList`. If there is more than one outstanding request, we set the `contention` flag to “true” (line 10) since each resource connected to the arbiter can only service one request at a time. We first begin by looking at the request list from the requester index after the previously granted requester, let us call this `prevGrant`. Line 13 checks that the index after the last granted requester, `prevGrant + 1`, is valid. If so, the request list is examined to see if any outstanding requests exist from `prevGrant + 1` to $n - 1$, the index of the last requesters in the requester list. The grant will be given to the first request seen in order in the list. The `prevGrant` variable will be modified accordingly (line 16). If no requests exists from position `prevGrant + 1` to $n - 1$, then it must exist between 0 to `prevGrant`, this is shown in line 20. As before, the grant is given to the first request seen in this interval and the `prevGrant` variable is updated in line 22.

4.2 Evaluation

4.2.1 Exploration Space

We take a brute-force approach to finding the optimal partitioning solution. This means we iterate through all possible partitioning schemes, while predicting the number of cycles needed in order to complete all the memory accesses collected in the trace. Section 3.1.6 quantifies the size of the partitioning exploration space. For the

Table 4.1: Size of the partitioning configuration search space.

benchmark	arrays to partition	total partitioning possibilities	actual partitioning possibilities
matrixadd	input (128×128)	57	45
histogram	input_array (32768)	121	55
matrixmult	matrixA (32×32), matrixB (32×32)	961	45
matrixmult (cyclic)	matrixA (32×32), matrixB (32×32)	961	45
matrixtrans	input_array (128×128), output (128×128)	3249	45
matrixtrans (block cyclic)	input_array (128×128), output (128×128)	3249	45
substring	str (16384)	106	51
los	obstacle (64×64)	43	37

purposes of checking the accuracy of the memory simulator, we only consider partition configurations with 16 partitions or less, since we have the ModelSim results for these. Table 4.1 presents the number of partitioning possibilities for each of our 8 benchmarks. Note that the option to not perform partitioning (null option) is also included as a possible partitioning configuration. Column 2 of Table 4.1 lists the arrays in the benchmark that are considered for partitioning and their dimensions; column 3 shows the total unrestricted partitioning configurations according to the equations presented in Section 3.1.6, and, column 4 shows the total actual number of partitioning configurations we performed the search on when we followed the restriction of allowing at most 16 partitions and when we pruned the search space according to Section 3.2.2.

While we do not perform a comparison of the speed at which our automatic approach is able to make a final prediction with the exhaustive ModelSim approach, we note that we can take a sampling approach which can significantly shorten the size of the input memory trace, and thus reduce the execution time of the algorithm. Moreover, the trace-based automatic approach uses our memory simulator, which

is only concerned with the memory requests of the system unlike ModelSim, which needs to perform full gate-level simulation.

4.2.2 Memory Simulator Accuracy

Table 4.2: Memory simulator evaluation for 16 partitions and under.

benchmark	predicted cycle of last memory access	actual cycle of last memory access	% error	partitioning schemes
matrixadd	4163	4163	0	1b4, 1b8, 1b16, 0c16, 0bc8_2, 0bc16_2, 1bc4_16, 1bc8_8, 1bc16_4
histogram	23987	23987	0.004	0b8, 0b16
matrixmult	16793	16799	0.03	0b16, 1b4, 1b8, 1b16, 0c8, 0c16, 1c16, 0bc4_2, 0bc8_2, 1bc4_4, 1bc8_2
matrixmult (cyclic)	16789	16789	0.16	0b16, 1b16, 0c8, 0c16, 1c4, 1c8, 1c16, 0bc4_2, 0bc8_2, 1bc4_2, 1bc8_2
matrixtrans	6420	6419	0.01	0b4, 0b8, 0b16, 0c8, 0c16, 0bc4_2, 0bc8_2, 0bc8_4, 0bc8_8, 0bc16_2, 0bc16_4
matrixtrans (block- cyclic)	6345	6344	0.01	0c8, 0c16, 1c16, 1b16, 0bc4_2, 0bc8_2, 0bc16_2, 1bc4_4, 1bc4_8, 1bc8_2, 1bc8_4, 1bc8_8, 1bc16_2, 1bc16_4
substring	4117	4117	16.52	0c8, 0c16, 0b8, 0b16, 0bc8_2, 0bc8_1024, 0bc16_2, 0bc16_512 (missing: 0c4)
los	79537	79537	0	1bc16_2

The goal of the memory simulator is to accurately predict the schedule of memory accesses of the circuit under various partitioning schemes. The memory access trace is gathered from an unpartitioned implementation of the circuit under a representative input. The size of the exploration space is described in Section 3.1.6. In this evaluation, we predict the execution cycle counts under all possible partitioning schemes where the number of partitions does not exceed 16 and compare the results to the results from Table 3.2 in Section 3.2.2. We intend to assess whether: *i*) the optimal partitioning schemes were detected and *ii*) the predicted cycle of the last memory access reported by the memory simulator matches the actual cycle from ModelSim simulation.

Table 4.2 shows the predicted and actual cycle of last memory access of the partitioned arrays leading to the lowest cycle count in columns 2 and 3. The partition configurations leading to the best cycle counts is listed in column 5. Column 4 shows the absolute percentage error for all variants of partitioning, where the number of partitions is equal to or less than 16. For our 8 benchmarks, the memory simulator had an average absolute percentage error of 2.09%. Figure 4.2 through 4.9 plots the accuracy of the memory simulator for each of the 8 benchmarks, where the horizontal axis plots the actual cycle of the last memory access to the arrays of interest and the vertical axis plots the predicted cycle of last memory access. The closer to the red line the data point falls, the more accurate the prediction is to the actual result.

The memory simulator is able to predict within 0.2% error for 7 of the 8 benchmarks. Figure 4.8 shows the accuracy of the memory simulator for the *substring* benchmark. The absolute percentage error for the *substring* benchmark is 16.52%, the highest percentage error of all benchmarks. The source of error was determined to be differences in scheduling due to different port bindings for memory accesses between the original unpartitioned case, from which we extract the memory access chain, and the memory port binding during the HLS of the partitioned case. Port binding refers to the process by which each memory access instruction is bound to a particular memory port of the RAM that it accesses. For example, LegUp employs dual-ported RAMs, which means there are two ports (a and b). When a certain LLVM IR memory instruction is converted to a hardware load or store to a RAM, the port through which it is allowed to perform the load or store is selected during the binding stage. Binding is performed using a bipartite weighted matching algorithm, where instructions and hardware resources belong to two disjoint sets of vertices to be matched. The weight between vertices of the instruction set and hardware resources set is determined based on a weighted sum of the existing number of input muxes to the hardware resource, the number of new input muxes required for this instruction, and whether the output register is sharable (i.e., the lifetimes of the instructions as-

Figure 4.2: Memory simulator *matrixadd* predicted versus actual cycle of last memory access

Figure 4.3: Memory simulator *histogram* predicted versus actual cycle of last memory access

Figure 4.4: Memory simulator *matrixmult* predicted versus actual cycle of last memory access

Figure 4.5: Memory simulator *matrixmult (cyclic)* predicted versus actual cycle of last memory access

Figure 4.6: Memory simulator *matrixtrans* predicted versus actual cycle of last memory access

Figure 4.7: Memory simulator *matrixtrans (block cyclic)* predicted versus actual cycle of last memory access

Figure 4.8: Memory simulator *substring* predicted versus actual cycle of last memory access

Figure 4.9: Memory simulator *los* predicted versus actual cycle of last memory access

signed to the same hardware resource do not overlap). Different port bindings occur as a result of differences in the weight construction between different designs.

For 7 of the 8 test cases, the memory simulator is also able to accurately predict the partitioning schemes which lead to the lowest execution cycle counts. In the case of *substring*, the memory simulator missed 1 of the 9 best partitioning schemes. The results are encouraging since looking at the variety of best partitioning schemes, we see that many of these schemes are non-intuitive. For examples, we believe it would be challenging for a designer to realize that the partitioning scheme *1bc4_16* for the benchmark *matrixadd* leads to one of the best solutions. As seen from the area comparison in Table 3.3, this solution actually leads to the lowest normalized area-delay product. This is where the memory simulator-driven automatic memory partitioning tool becomes a useful tool for detecting multiple good solutions to aid in pruning the exploration space for the hardware/software designer.

4.2.3 Sources of Error

The sources of error for the memory simulator are listed below:

1. Differences in port bindings.
2. Differences in LegUp scheduling.
3. Interactions between multiple threads.
4. Differences in input data.

When we “execute” the memory trace under different partitioning schemes to determine the number of cycles required to complete the memory instructions in a memory trace collected from an unpartitioned design, we assume the same port bindings of instructions but we prescribe them to their corresponding sub-array partitions based on their address. In reality, the port bindings for memory access memory instructions may change from an unpartitioned design to a partitioned design. The

memory simulator is not able to predict these changes, and this can cause dramatic differences in the predicted execution scheduled versus the actual execution schedule.

Listing 4.4: LLVM IR pre- array partitioning pass

```

1 ; <label>:9                                ; preds = %9, %.preheader
2 %sum.12 = phi i32 [ %sum.04, %.preheader ], [ %11, %9 ]
3 %j.01 = phi i32 [ 0, %.preheader ], [ %12, %9 ]
4 %scevgep = getelementptr [128 x [128 x i32]]* @input, i32 0, i32 %8, i32 %j.01
5 %10 = load i32* %scevgep, align 4, !tbaa !7
6 %11 = add nsw i32 %10, %sum.12
7 %12 = add nsw i32 %j.01, 1
8 %exitcond1 = icmp eq i32 %12, 128
9 br i1 %exitcond1, label %13, label %9

```

Listing 4.5: LLVM IR post- array partitioning pass

```

1 ; <label>:9                                ; preds = %9, %.preheader
2 %sum.12 = phi i32 [ %sum.04, %.preheader ], [ %14, %9 ]
3 %j.01 = phi i32 [ 0, %.preheader ], [ %15, %9 ]
4 %and.1 = and i32 %8, 63
5 %GEP_sub_input_0_2 = getelementptr [64 x [128 x i32]]* @sub_input_0, i32 0, i32 %
   and_1, i32 %j.01
6 %and.3 = and i32 %8, 63
7 %GEP_sub_input_1_4 = getelementptr [64 x [128 x i32]]* @sub_input_1, i32 0, i32 %
   and_3, i32 %j.01
8 %rshift.5 = ashr i32 %8, 6
9 %10 = icmp eq i32 %rshift.5, 0, !legup.ap_pred_id !7
10 call void @__legup_preserve_value_0_1(i1 %10), !legup.ap_pred_id !7
11 %Load_sub_input_0_7 = load i32* %GEP_sub_input_0_2, !legup.ap_pred !7
12 %11 = select i1 %10, i32 %Load_sub_input_0_7, i32 0
13 %12 = icmp eq i32 %rshift.5, 1, !legup.ap_pred_id !8
14 call void @__legup_preserve_value_0_1(i1 %12), !legup.ap_pred_id !8
15 %Load_sub_input_1_9 = load i32* %GEP_sub_input_1_4, !legup.ap_pred !8
16 %13 = select i1 %12, i32 %Load_sub_input_1_9, i32 0
17 %Or.10 = or i32 %11, %13
18 %14 = add nsw i32 %Or.10, %sum.12
19 %15 = add nsw i32 %j.01, 1
20 %exitcond1 = icmp eq i32 %15, 128
21 br i1 %exitcond1, label %16, label %9

```

When we perform LegUp compilation on an unpartitioned design and a partitioned design, the schedules may differ for various reasons. Firstly, the schedule of the partitioned case may be longer due to the size of the `or` tree, described in Section 3.1.3. Let us again consider the transformation pass that performs array partitioning, Listings 4.4 and 4.5 show the LLVM IR before and after the pass. The instructions on lines 4, 6, 8, 9, 10, 12, 13, 14, 16, and 17 in listing 4.5 are the additional instructions that need to be scheduled.

(a) Before array partitioning pass.

(b) After array partitioning pass.

(c) After array partitioning pass with tight clock period constraint.

Figure 4.10: Comparison of inner loop basic block schedule for *matrixadd* as seen through the *scheduleviewer*. Blue blocks denote the cycles the instructions to the left of the block is scheduled into; yellow blocks denote instructions under analysis; red boxed instructions are dependencies of the yellow block instructions; orange boxed instructions depend on the yellow block instruction output.

Figures 4.10a and 4.10b visually show the schedule of the inner loop basic block using LegUp’s built-in tool `scheduleviewer`. Notice in Figure 4.10b that the instructions to compute the new sub-array indices and predicates on lines 4, 8, 9, 10, 13, and 14 have all been scheduled into the same cycle as the initial cycle of the `GEP`s and loads on lines 5, 7, 11, and 15. Line 6 in Listing 4.5 is not scheduled as it was a redundant instruction and was thus eliminated. The `select` and `or` instructions for implementing the `or` tree to determine the valid read value from the predicated reads depend on the previous load instruction to complete, therefore, they are scheduled in the next state. Referring to Figure 4.10b, the yellow box shows the instruction we are focused on (`%Or_10 = or i32 %11, %13`), the red box shows the instructions which the current instruction depends on, and the orange box shows the instruction that depends on the output of the current instruction. Here we see that the `select`, `or`, and the final `add` instructions were all scheduled into the same cycle. This ensured that the schedule of the unpartitioned and partitioned designs are identical, despite the addition of multiple instructions. However, Figure 4.10c shows a contrived case when the target clock period for LegUp was constrained to 3ns. In this case, the LegUp scheduler, in its attempt to satisfy the criteria of a 3ns clock period, scheduled the `select`, `or`, and the `add` instructions in 3 different cycles. This contrived example demonstrates the effect on schedule length in the event when a certain chain of operations with high combinational delay exceeds the clock period constraint.

Another factor causing schedule length differences are the other LLVM optimization passes. These optimization passes were written to optimize code for general-purpose processor architectures, but not for our FPGA/hardware HLS paradigm and can therefore cause unwanted effects to our IR code, affecting the downstream hardware generation. Specifically, the `-indvars2` pass will modify the `GEP` instruction in an undesirable way. The `GEP` instruction dereferences a pointer to a specific element in an aggregate data structure based on the indices of a multi-dimensional array to give the address of that element. When the `GEP` instruction is within an inner loop body,

where one index does not change within the inner loop iterations, the **GEP** instruction will be split into multiple **GEP** instructions. Comparing Figure 4.11 and Figure 4.12 we observe the difference when the **-indvars2** pass performs the change versus when the pass is suppressed. In Figure 4.11, the **-indvars2** pass is applied unrestricted, and the **GEP** instruction is seen to be split into two stages. The first stage, in basic block **.preheader**, computes an intermediate address which results in the address of the first element in the row indicated by the variable **%8**. In the second stage, in basic block **%9**, the final address of access is computed using the precomputed row address from the previous step. In Figure 4.12, the **-indvars2** pass is applied with the **-disable-iv-rewrite** option enabled. This prevents the **GEP** instruction from being split in basic block **%9**. The reason the **GEP** split is undesirable is because it leads to an increase in the schedule length. This is because the **GEP** instruction is a “high combinational delay” instruction since it potentially contains a combination of multiply and addition instructions and therefore the delay is unknown at compile time. As a simplification, LegUp currently pessimistically expects that the delay is half of the clock period we are compiling for, which is set as an input parameter to LegUp. This limits the number of instructions that can fit into the same clock period and therefore the schedule length may grow when there are more **GEP** instructions. In this particular example, we may see that when the **-indvars2** pass is applied, the **.preheader** basic block schedule length may be increased by 1 cycle. To disable this transformation, we invoke the option **-disable-iv-rewrite**. Further, the delay estimation in LegUp can be augmented to provide a more precise model of the **GEP** instruction delay.

The unpredictability of schedule lengths of basic blocks may have dramatic effects on the interaction of threads and their accesses. The variability in the arbiter grant patterns to each thread can be quite large and is heavily dependent on the work load, scheduling, and port binding. Without examining into excessive detail, we present a short description of when scheduling led to dramatic differences in execution cycle

Figure 4.11: *Matrixadd* final optimized IR without `-disable-iv-rewrite` option.

Figure 4.12: *Matrixadd* final optimized IR with `-disable-iv-rewrite` option enabled.

counts due to interaction of threads accessing partitions. Two executions of the *matrixadd* benchmark are compared, **exec1** and **exec2**, each running with 8 threads. The partition scheme applied is such that there are two sub-arrays created. In both runs, all 8 threads will first try to access the first partition, then all 8 threads will migrate to access the second partition. This back and forth pattern repeats until all rows of the input matrix are accessed. The only difference between **exec1** and **exec2** is that **exec1** is compiled with the `-disable-iv-rewrite` option turned on, so the inner loop basic block has a schedule length of 2 cycles, while **exec2** is compiled without the `-disable-iv-rewrite` option and the inner loop basic block has a schedule length of 3 cycles. Through further exploration of the memory trace, it was observed that in both **exec1** and **exec2**, each of the 8 threads begin their memory accesses in the first partition in a staggered fashion due to the staggering of the **start** signal for each thread module. In steady state, **exec1** will have 7 in flight reads where 1 read will be granted access. In steady state, **exec2** will have 6 in flight reads where 1 read will be granted access. This means that the steady state memory load throughput of each execution is the same. While the throughput at steady state is the same, we eventually start to detect differences in the performance progression between **exec1** and **exec2** much further into the runs. The pattern seen in **exec1** is that the workload was split in a way where a group of threads worked exclusively on partition 1 while another group of threads worked exclusively on partition 2. The pattern seen in **exec2** is that the threads did not split in the same way as in **exec1**, but progressed slowly together from partition 1 to partition 2 collectively. Therefore, the threads' accesses of partitions in **exec1** created less contentions. This effect is non-trivial and can significantly affect the prediction accuracy. Thread level interaction is beyond the scope of this thesis.

Lastly, our trace-based approach relies on a representative input data set. For benchmarks which are input-dependent, the predictions will also be affected by input data.

4.3 Summary

In this chapter, a trace-based approach for automatic memory partitioning selection is presented, which used our memory simulator to predict the execution cycle counts under various partition schemes. This automatic approach requires a single LegUp compilation and execution in ModelSim from which to extract a memory access trace. The memory trace is then evaluated under all possible partitioning schemes to find the best candidates. This automatic approach relies on an accurate memory simulator. We provided the evaluation of the memory simulator in this chapter which showed an average absolute percentage error across 8 benchmarks of 2.09%. Finally, we concluded with a list of potential sources of errors that can be considered in future work.

Chapter 5

Conclusion

Memory access is often a performance bottleneck in computer hardware. In this work, we considered memory architecture synthesis in the HLS context, specifically in the synthesis of parallel hardware from multi-threaded software. We implemented flexible array partitioning as an LLVM compiler pass within the LegUp HLS framework, where, partitions do not need to be perfect (i.e. a memory access instruction may access any partition and each partition may have multiple requesters). The array partitions are implemented as separate logical RAMs (banks) in the hardware, with arbiters inserted to manage concurrent accesses among threads. With the banked architecture synthesis, the geomean speed up of the parallel execution cycles is $2.21 \times$ across 8 benchmarks.

As it is onerous for designers to select an array partitioning scheme and specify it using specialized compiler pragmas, we devised an automatic memory partitioning tool that uses a cycle accurate memory simulator, which, with an accurate model of memory access arbitration can predict with an average absolute percent error of 2.09% the last cycle of memory access of our 8 multi-threaded benchmarks. The simulator can be applied to automatically select the best partitioning for an array.

The work presented differs from past works in memory partitioning in that previous works all focused on improving memory bandwidth to allow for loop-pipelining,

in particular, to allow for an Π of 1. However, our work focuses on multi-threaded functions and the contention that arise as a result of their interaction as each thread requests for the common memory resource. These are fundamentally different problems, which require very different approaches. In the loop-pipelining work, the schedule is pre-determined, meaning that once the partition is created, the schedule is deterministic and arbitration need not be considered. Secondly, our work involves predicated memory instruction, which makes it possible to perform “fast” loads and stores to multiple memory banks by one memory access instruction without creating additional control flow. It is unclear in the documentation of other tools that support memory partitioning whether the partitioning is necessarily perfect.

5.1 Future Work

As mentioned in Section 3.1.5, we only consider partitioning configurations which lead to minimal partition numbers and index calculation overhead. In particular, we reject a partitioning configuration if it leads to a division or modulo operation by a non-power-of-2 integer. However, as suggested by [46], we can employ the techniques from [42] to perform division and modulo of integers (our array index) by a constant (the block size or partition number) efficiently. This way, we can support non-power-of-2 partitions as the performance would no longer be as negatively impacted by the high latency division and modulo instructions.

Currently, the array partitioning pass only supports partitioning in one dimension. In the future, this can be extended also to multiple dimensions to observe whether further improvements can be achieved.

In our performance analysis of the array partitioning mechanism, we so far provide a set of 8 benchmarks of which 2 exhibit irregular, data dependent memory accesses. In future work, we can extend this set of benchmarks to include more tests with irregular accesses such as graph algorithms including triangle counting and the PageRank algorithm used for sorting relevant webpages.

The automatic memory partitioning algorithm currently takes a brute force approach to finding the best partitioning solution. However, as mentioned in Section 3.1.6, the exploration space of array partitioning, especially when working with multiple arrays can grow quite large. Instead, we can employ a pruning approach, which is able to quickly eliminate the partitioning solutions that are not likely to improve the performance relative to other solutions.

Another improvement to the automatic memory partitioning approach is to create a better cost function for the automatic memory partitioning algorithm. The current approach strictly considers cycles improvement and uses the number of partitions as a ranking system, where less partitions is favored. However, as we noticed in Section 3.2.3, when the gain in cycle count is relatively small, further partitioning will lead to decrease in frequency, which will in turn result in longer wall-clock time. Therefore, it is necessary to account for the area and frequency impacts of partitioning.

As noted in Section 3.2.2, the current implementation of LegUp and array partitioning does not allow for the opportunity to share the two ports of a dual-ported RAM across threads, i.e. there is no notion of scheduling ports across multiple pthread function instances. For example, if thread 0 and thread 1 both access a different address of the same RAM at a given execution cycle, LegUp is not able to, at the time of scheduling and binding, realize that these two thread instances should be bound to different ports. With this additional port optimization, we should expect to see further performance improvements.

We can also look at extensions of the automatic partitioning portion of this work, currently focusing on on-chip memory partitioning, to target memory architectures containing off-chip DRAM paired with on-chip caching schemes.

Appendices

Appendix A

Array Partitioning Full Results

Full ModelSim and Quartus II v15.0 synthesis results for total of 8 benchmarks. All tables follow the same format, the first column shows the partitioning scheme used, the second column shows the total cycle count of the benchmark including the error checking portion, the third column shows the cycle count of the parallel kernel regions only, the fourth column shows the cycle of the last memory access in the parallel function, column five shows the synthesized circuit frequency, column six shows the wall-clock time computed for the full benchmark, column seven shows the wall-clock time computed for the parallel kernel regions only, column 8-12 show the area usage for ALMs, registers, memory bits, M10K RAMs, and DSPs respectively, column 13 shows the area-delay product, column 14 shows the area-delay product normalized to the unpartitioned case, column 15 shows the number of memory port contentions in the benchmark, column 16 shows the test error checking result. The area-delay product is computed as described in Section 3.2.3.

The synthesis results for *matrixadd* and *histogram* is the average of five runs, each with a different seed. Due to limited time constraints, the other benchmarks show only the result of one synthesis run.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsps	adprod	nadprod	contentions	pass/fail
1bc4.16	4180	4157	4163	89.61	46.65	46.39	3754.2	4931.2	525312	68	0	237247.74	0.33	4	pass
0c4	4195	4172	4178	87.18	48.12	47.86	3818.8	4901.4	525312	68	0	247857.37	0.35	120	pass
1b4	4180	4157	4163	82.65	50.57	50.3	3850.2	5141.6	525312	68	0	262073.06	0.37	0	pass
1bc8.8	4180	4157	4163	87.11	47.99	47.72	5138.6	6135.4	525312	60	0	302477.99	0.43	4	pass
0bc4.4	5162	5138	5144	90.19	57.24	56.97	3820.2	4905.2	525312	68	0	295115.99	0.41	4713	pass
0bc4.2	5142	5119	5125	89.62	57.37	57.12	3822	4914.4	525312	68	0	295995.84	0.42	7681	pass
0c8	4182	4159	4165	86.14	48.55	48.28	5257.8	6107.6	525312	60	0	311782.58	0.44	3	pass
0bc8.2	4180	4157	4163	85.45	48.92	48.65	5252.4	6110.2	525312	60	0	313909.26	0.44	0	pass
1b8	4180	4157	4163	82.57	50.62	50.35	5283	6349.8	525312	60	0	326419.05	0.46	0	pass
0bc8.4	4689	4665	4671	87.29	53.72	53.44	5255.4	6113.6	525312	60	0	344976.58	0.49	3603	pass
0bc4.16	5927	5903	5909	88.91	66.66	66.39	3817.6	4910	525312	68	0	343740.86	0.48	8068	pass
0b8	5209	5185	5191	87.79	59.33	59.06	4934.2	6110.8	525312	60	0	362285.85	0.51	4140	pass
1bc2.16	8234	8211	8217	94.33	87.29	87.05	2814.6	4325.4	525312	68	0	363398.93	0.51	16350	pass
1bc2.32	8234	8211	8217	92.5	89.01	88.77	2810	4323	525312	68	0	370170.90	0.52	16342	pass
0bc2.2	8241	8214	8220	92.39	89.2	88.91	2840.6	4320	525312	68	0	373475.35	0.53	14720	pass
0bc2.4	8343	8318	8324	93.05	89.66	89.39	2841.4	4320.4	525312	68	0	375563.15	0.53	14565	pass
0bc2.8	8338	8314	8320	92.66	89.98	89.72	2838.6	4320.8	525312	68	0	376698.39	0.53	14832	pass
0c2	8235	8212	8218	90.61	90.88	90.63	2836.6	4320	525312	68	0	380337.86	0.53	8178	pass
0bc2.16	8539	8515	8521	91.79	93.03	92.77	2846.2	4318.4	525312	68	0	390209.17	0.55	14894	pass
1b2	8240	8217	8223	85.49	96.39	96.12	3033	4543.2	525312	68	0	422255.16	0.59	16366	pass
1bc16.4	4180	4157	4163	82	50.97	50.69	8011.2	8503.8	525312	68	0	475026.13	0.67	4	pass
1bc4.8	8223	8200	8206	89.93	91.44	91.18	3751.4	4933.2	525312	68	0	466057.45	0.66	16310	pass
0c16	4180	4157	4163	81.46	51.31	51.03	8479.4	8467.4	525312	68	0	502104.58	0.71	0	pass
0bc16.2	4180	4157	4163	80.83	51.71	51.43	8470.33	8462	525312	68	0	505573.87	0.71	0	pass
1b16	4180	4157	4163	79.12	52.83	52.54	8354	8713.6	525312	68	0	510373.56	0.72	0	pass
0bc4.8	8306	8280	8286	86.71	95.79	95.49	3825.4	4906	525312	68	0	495153.85	0.70	14871	pass
1bc8.4	8197	8174	8180	88.08	93.06	92.48	5138	6145.6	525312	60	0	588166.40	0.83	16224	pass
0bc2.32	12954	12928	12934	92.02	140.77	140.49	2843	4315.4	525312	68	0	590479.47	0.83	15891	pass
0bc8.8	8428	8402	8408	85.5	98.58	98.27	5253.4	6111.4	525312	60	0	634175.62	0.89	14644	pass
0bc16.4	5601	5577	5583	80.89	69.24	68.94	8482.8	8474.6	525312	68	0	678562.63	0.95	7961	pass
1bc2.2	14440	14414	14420	93.16	155	154.72	2816.6	4323.8	525312	68	0	646203.55	0.91	14489	pass
1c2	14768	14742	14748	93.59	157.8	157.52	2816.2	4325.6	525312	68	0	657835.02	0.93	14974	pass
0b2	15018	14992	14998	92.36	162.61	162.33	2887.6	4326	525312	68	0	689512.91	0.97	15635	pass
nopartition	16420	16396	16402	93.35	175.9	175.64	2688.8	4025.6	525312	68	0	711131.23	1.00	16376	pass
1bc2.4	16233	16207	16213	93.24	174.1	173.82	2805.8	4325.6	525312	68	0	724099.36	1.02	16241	pass
1c4	13322	13296	13302	89.26	149.24	148.95	3757.2	4928.8	525312	68	0	762206.94	1.07	13476	pass
1bc2.8	16396	16372	16378	91.83	178.54	178.28	2805.2	4327.2	525312	68	0	742571.86	1.04	16363	pass
1c8	11111	11085	11091	88.16	126.03	125.74	5135	6132.8	525312	60	0	796562.90	1.12	11104	pass
1bc4.2	14046	14020	14026	89.91	156.22	155.93	3751.4	4924.2	525312	68	0	797020.60	1.12	14093	pass
0b4	14314	14288	14294	91.5	156.43	156.15	3753.8	4912	525312	68	0	798519.87	1.12	15500	pass
1bc16.2	8096	8073	8079	82.01	98.72	98.44	8004	8495.8	525312	68	0	921792.16	1.30	15956	pass
1bc4.4	16233	16207	16213	90.75	178.88	178.6	3756	4927.4	525312	68	0	913717.60	1.28	16241	pass
1bc8.2	14046	14020	14026	88.88	158.03	157.74	5143.6	6151.4	525312	60	0	1000639.46	1.41	14093	pass
1c16	11077	11051	11057	83.44	132.75	132.44	8015.2	8485.6	525312	68	0	1241651.49	1.75	11066	pass
0b16	10921	10895	10901	80.55	135.58	135.26	8053.5	8466.5	525312	68	0	1273270.01	1.79	14186	pass

Table A.1: *Matrixadd* benchmark full performance and area results (average of 5 synthesis runs).

partition	f0yc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	nbits	rms	dsp	adprod	nadprod	contentions	pass/fail
0bc2.2	25638	25623	25597	92.46	277.3	277.14	3639.8	5692	1048576	128	0	1718212.57	0.79	11281	pass
0bc2.4	25758	25745	25671	92.33	278.98	278.84	3651.6	5690.2	1048576	128	0	1732042.54	0.80	11672	pass
0c2	25483	25474	25448	90.65	281.13	281.03	3635.2	5689.8	1048576	128	0	1741037.06	0.80	10422	pass
0bc2.8192	25226	25211	25175	89.1	283.13	282.96	3659.2	5699.4	1048576	128	0	1759784.83	0.81	9023	pass
0bc2.8	25987	25974	25944	91.52	283.96	283.81	3640.2	5690.8	1048576	128	0	1759678.76	0.81	12660	pass
0bc2.4096	25224	25215	25189	88.9	283.72	283.62	3655.4	5693	1048576	128	0	1762811.75	0.81	8827	pass
0bc2.16	26281	26266	26240	91.86	286.11	285.95	3646.4	5691.2	1048576	128	0	1774720.08	0.82	12780	pass
0c4	24622	24613	24579	93.25	264.03	263.94	4303.8	6287.2	1048576	128	0	1811631.37	0.83	4590	pass
0bc4.4	24719	24710	24684	90.82	272.18	272.08	4309.6	6283.6	1048576	128	0	1869080.77	0.86	5164	pass
0bc4.2	24728	24719	24693	90.01	274.72	274.62	4300.4	6291.2	1048576	128	0	1884003.05	0.87	5293	pass
0bc4.4096	24316	24307	24281	87.45	278.07	277.97	4292	6279	1048576	128	0	1904650.44	0.88	2541	pass
0bc4.8	24921	24906	24880	89.19	279.43	279.26	4319.6	6285	1048576	128	0	1921197.10	0.88	5823	pass
0bc4.16	25480	25465	25439	90.68	281	280.84	4301.4	6283.8	1048576	128	0	1926955.58	0.89	8553	pass
0bc2.32	27674	27665	27639	90.23	306.69	306.59	3639.2	5694	1048576	128	0	1900612.73	0.87	17751	pass
0b2	25296	25287	25261	80.97	312.41	312.3	3803.2	5686	1048576	128	0	1987227.36	0.91	9006	pass
0bc2.64	29729	29710	29648	92.27	322.2	322	3636.4	5694.2	1048576	128	0	1995240.80	0.92	24049	pass
0bc4.2048	25283	25274	25248	85.28	296.48	296.38	4331.2	6279.4	1048576	128	0	2042413.86	0.94	8879	pass
0bc4.32	27318	27309	27283	89.84	304.08	303.98	4319.6	6287.2	1048576	128	0	2091260.81	0.96	15004	pass
0bc4.64	29106	29091	29065	91.23	319.05	318.89	4317	6285.4	1048576	128	0	2193006.53	1.01	21363	pass
0b4	24341	24332	24306	76.88	316.59	316.48	4406.6	6269.4	1048576	128	0	2204789.57	1.01	2542	pass
0bc2.128	32113	32106	32073	91.34	351.57	351.49	3659.8	5695.2	1048576	128	0	2186197.50	1.01	30616	pass
nopartition	33084	33077	32784	91.14	362.99	362.92	3430.8	5401.6	1048576	128	0	2174181.14	1.00	32752	pass
0bc2.512	33053	33032	32753	92.13	358.76	358.54	3642.2	5690.8	1048576	128	0	2223736.79	1.02	32685	pass
0c8	24342	24333	24307	84.56	287.85	287.75	5567.8	7469	1048576	128	0	2338774.45	1.08	2357	pass
0bc2.2048	33079	33072	32779	91.93	359.81	359.74	3651.6	5694.6	1048576	128	0	2234560.98	1.03	32738	pass
0bc2.256	32885	32864	32689	91.15	360.78	360.55	3647	5691.8	1048576	128	0	2237933.85	1.03	32487	pass
0bc2.1024	33090	33069	32773	91.55	361.46	361.23	3643	5693	1048576	128	0	2240709.69	1.03	32705	pass
0bc8.4	24503	24494	24468	85.24	287.47	287.37	5662.2	7478.8	1048576	128	0	2362813.61	1.09	2883	pass
0bc8.2048	24314	24305	24279	82.92	293.24	293.13	5500	7465.8	1048576	128	0	2362627.80	1.09	2543	pass
0bc8.2	24396	24387	24361	84.19	289.79	289.68	5648.4	7471.2	1048576	128	0	2377809.31	1.09	2707	pass
0bc8.8	24764	24751	24723	83.93	295.06	294.91	5630	7468.8	1048576	128	0	2415312.90	1.11	4157	pass
0bc8.16	25421	25406	25380	84.45	301.02	300.84	5651	7470.6	1048576	128	0	2470197.24	1.14	7229	pass
0bc4.128	32113	32106	32073	91.03	352.79	352.71	4309.4	6281.8	1048576	128	0	2422906.07	1.11	30616	pass
0bc8.1024	25203	25194	25168	81.4	309.61	309.5	5485.6	7454.8	1048576	128	0	2490113.20	1.15	8902	pass
0bc8.32	26899	26890	26864	85.22	315.63	315.52	5634.2	7471.4	1048576	128	0	2585433.98	1.19	12438	pass
0bc4.1024	33090	33069	32773	90.07	367.36	367.13	4304	6286.4	1048576	128	0	2519980.32	1.16	32705	pass
0bc4.512	33053	33032	32753	89.43	369.6	369.36	4307.8	6288	1048576	128	0	2536690.61	1.17	32685	pass
0bc4.256	32885	32864	32689	89.04	369.34	369.1	4315	6283.8	1048576	128	0	2537562.50	1.17	32487	pass
0b8	24022	24013	23987	73.56	326.57	326.45	5548.4	7420.6	1048576	128	0	2646987.18	1.22	0	pass
0bc8.64	28943	28928	28894	84.08	344.24	344.06	5635	7466.6	1048576	128	0	2819571.70	1.30	20442	pass
0bc8.128	32113	32106	32073	85.41	375.97	375.89	5643	7476.4	1048576	128	0	3083425.67	1.42	30616	pass
0bc16.2	24243	24234	24208	79.19	306.14	306.03	8582.8	9825.8	1048576	112	0	3312101.48	1.52	1566	pass
0bc16.4	24323	24314	24288	79.54	305.79	305.67	8611.6	9832.6	1048576	112	0	3317008.57	1.53	2007	pass
0bc8.256	32885	32864	32689	85.03	386.74	386.49	5622.4	7469.2	1048576	128	0	3162415.78	1.45	32487	pass
0bc16.1024	24314	24305	24279	78.02	311.64	311.52	8550.6	9820.2	1048576	112	0	3361487.71	1.55	2535	pass
0bc16.8	24815	24803	24769	79.32	312.68	312.68	8588	9822.6	1048576	112	0	3385699.04	1.56	4053	pass
0bc8.512	33053	33032	32753	83.95	393.71	393.46	5633.4	7469.8	1048576	128	0	3223775.16	1.48	32685	pass
0c16	24160	24151	24125	76.57	315.54	315.43	8608	9804.6	1048576	112	0	3421784.64	1.57	1264	pass
0bc16.16	25312	25297	25271	79.01	320.38	320.19	8600.8	9840.2	1048576	112	0	3471115.75	1.60	6258	pass
0b16	24022	24013	23987	71.62	335.43	335.3	8231.8	9755.4	1048576	112	0	3511194.54	1.61	0	pass
0bc16.512	25231	25222	25196	76.57	329.51	329.39	8565	9830.6	1048576	112	0	3559058.95	1.64	9005	pass
0bc16.32	26854	26845	26813	79.73	336.83	336.72	8602	9851.8	1048576	112	0	3650718.24	1.68	12139	pass
0bc16.64	28894	28828	28894	79.44	364.32	364.13	8598	9809.4	1048576	112	0	3946440.94	1.82	20442	pass
0bc16.128	32113	32106	32073	78.84	407.3	407.21	8597.6	9835.6	1048576	112	0	4413179.10	2.03	30616	pass
0bc16.256	32885	32864	32689	79.47	413.79	413.53	8598.8	9820.4	1048576	112	0	4482168.96	2.06	32487	pass

Table A.2: *Histogram* benchmark full performance and area results (average of 5 synthesis runs).

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsp	adprod	nadprod	contentions	pass/fail
0bc4.2	18919	16799	16793	93.2	202.99	180.25	6318	10991	131072	24	16	1311859.50	0.78	8	pass
1bc8.8	18919	16799	16793	95.29	198.54	176.29	6671	11180	131072	24	16	1345268.99	0.80	8	pass
1b4	18919	16799	16793	91.63	206.47	183.34	6511	11195	131072	24	16	1369733.14	0.82	8	pass
1bc4.4	18919	16799	16793	93.05	203.32	180.54	6658	11134	131072	24	16	1375353.72	0.82	8	pass
1bc4.8	18919	16799	16793	89.85	210.56	186.97	6681	11157	131072	24	16	1428637.77	0.85	8	pass
1b8	18919	16799	16793	91.63	206.47	183.34	8559	13607	131072	40	16	1803882.26	1.08	0	pass
1bc8.4	18919	16799	16793	88.51	213.75	189.8	8705	13592	131072	40	16	1895153.00	1.13	0	pass
1bc16.4	18919	16799	16793	87.93	215.16	191.05	8670	13564	131072	40	16	1900947.50	1.14	0	pass
1bc8.2	18919	16799	16793	87.47	216.29	192.05	8655	13567	131072	40	16	1908016.75	1.14	8	pass
0bc8.2	18919	16799	16793	89.36	211.72	187.99	9391	13213	131072	40	16	2006041.29	1.20	8	pass
0c8	18919	16799	16793	85.62	220.96	196.2	9369	13227	131072	40	16	2089333.80	1.25	0	pass
1b16	18919	16799	16793	82.58	229.1	203.43	14880	18304	131072	72	16	3417624.00	2.04	0	pass
0bc16.2	18919	16799	16793	80.97	233.65	207.47	14765	17648	131072	72	16	3461636.95	2.07	8	pass
1bc16.2	18919	16799	16793	81.41	232.39	206.35	15103	18339	131072	72	16	3512696.05	2.10	0	pass
0c16	18919	16799	16793	79.72	237.32	210.73	14714	17656	131072	72	16	3505282.82	2.09	0	pass
0b16	18919	16799	16793	79.13	239.09	212.3	14608	17626	131072	72	16	3508894.40	2.10	8	pass
1c16	18919	16799	16793	79.7	237.38	210.78	15001	18313	131072	72	16	3566608.38	2.13	8	pass
0c4	18920	16800	16794	94.21	200.83	178.33	6333	10990	131072	24	16	1300560.69	0.78	8	pass
0bc16.4	20025	17905	17899	87.19	229.67	205.36	9349	13216	131072	40	16	2182771.44	1.30	16270	pass
0b8	20025	17905	17899	84.01	238.36	213.13	9332	13215	131072	40	16	2261735.56	1.35	16270	pass
0bc8.4	20025	17905	17899	81.1	246.92	220.78	9423	13186	131072	40	16	2363008.34	1.41	16270	pass
0bc4.8	20244	18122	18116	93.37	216.81	194.09	6337	10991	131072	24	16	1416274.73	0.85	10592	pass
0bc8.8	20244	18122	18116	91.23	221.9	198.64	6407	10998	131072	24	16	1463380.88	0.87	10592	pass
0b4	20244	18122	18116	89.48	226.24	202.53	6331	10991	131072	24	16	1476646.23	0.88	10592	pass
0bc4.4	20523	18403	18397	93.67	219.1	196.47	6311	10986	131072	24	16	1428533.37	0.85	24620	pass
0bc2.16	21955	19833	19827	98.81	222.19	200.72	4953	9858	131072	16	16	1154742.16	0.69	20274	pass
0bc4.16	21955	19833	19827	95.46	229.99	207.76	5091	9873	131072	16	16	1223914.16	0.73	20274	pass
0b2	21955	19833	19827	92.4	237.61	214.64	5355	9874	131072	16	16	1321109.20	0.79	20274	pass
0c2	22248	20128	20122	94.95	234.31	211.99	4957	9872	131072	16	16	1220426.43	0.73	27660	pass
1bc2.4	22249	20129	20123	94.18	236.24	213.73	4964	9952	131072	16	16	1231939.72	0.74	53268	pass
1bc4.2	22249	20129	20123	93.98	236.74	214.18	6664	11165	131072	24	16	1632908.32	0.98	53268	pass
1c8	22249	20129	20123	83.06	267.87	242.34	8707	13560	131072	40	16	2420249.58	1.45	53268	pass
1bc2.8	22409	20289	20283	95.96	233.52	211.43	4981	9939	131072	16	16	1222276.83	0.73	45300	pass
0bc2.2	22509	20386	20380	94.11	239.18	216.62	4936	9863	131072	16	16	1242532.32	0.74	43792	pass
1b2	22646	20512	20506	102.19	221.61	200.72	5197	9979	131072	16	16	1203717.84	0.72	35721	pass
1bc2.16	22646	20512	20506	97.41	232.48	210.57	4979	9945	131072	16	16	1216884.03	0.73	35721	pass
1bc4.16	22646	20512	20506	95.54	237.03	214.7	5122	9984	131072	16	16	1271453.40	0.76	35721	pass
0bc2.8	23483	21361	21355	95.37	246.23	223.98	4955	9871	131072	16	16	1289004.90	0.77	31732	pass
0bc2.4	23532	21412	21406	91.45	257.32	234.14	4950	9858	131072	16	16	1346305.00	0.80	45780	pass
1c2	34838	32716	32710	93.34	373.24	350.5	4949	9937	131072	16	16	2015024.50	1.20	42831	pass
1c4	34838	32716	32710	94.65	368.07	345.65	6654	11185	131072	24	16	2631779.10	1.57	42831	pass
1bc2.2	34922	32800	32794	94.59	369.19	346.76	4976	9942	131072	16	16	2002885.76	1.20	51523	pass
nopartition	34961	32839	32833	99.45	351.54	330.21	4267	9301	131072	16	16	1673174.07	1.00	65164	pass
1bc2.32	34961	32839	32833	98.59	354.61	333.09	4548	9354	131072	16	16	1781365.32	1.06	65164	pass
0bc2.32	34961	32839	32833	96.39	362.7	340.69	4549	9319	131072	16	16	1822350.81	1.09	65164	pass

Table A.3: *Matrixmult* benchmark full performance and area results.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsp	adprod	nadprod	contentions	pass/fail
1c4	18914	16794	16789	87.79	215.45	191.3	8614	12617	131072	24	64	2106978.20	0.75	8	pass
1bc4.2	18914	16794	16789	85.45	221.35	196.54	8563	12598	131072	24	64	2154668.02	0.76	8	pass
0bc4.2	18914	16794	16789	83.93	225.35	200.1	8815	12599	131072	24	64	2244121.50	0.80	8	pass
1bc8.2	18914	16794	16789	81.47	232.16	206.14	14319	15167	131072	40	64	3512419.46	1.25	8	pass
1c8	18914	16794	16789	80.71	234.35	208.08	17066	15345	131072	40	64	4117070.88	1.46	0	pass
0c8	18914	16794	16789	74.74	253.06	224.7	16853	14632	131072	40	64	4398053.10	1.56	0	pass
0bc8.2	18914	16794	16789	74.4	254.22	225.73	16903	14633	131072	40	64	4429499.79	1.57	8	pass
1c16	18914	16794	16789	71.66	263.94	234.36	20036	20377	131072	72	64	5483086.56	1.94	0	pass
1b16	18914	16794	16789	77.45	244.21	216.84	23341	20244	131072	72	64	5789844.84	2.05	8	pass
1bc16.2	18914	16794	16789	73.55	257.16	228.33	23884	20211	131072	72	64	6220622.52	2.20	8	pass
0bc16.2	18914	16794	16789	70.72	267.45	237.47	23005	19250	131072	72	64	6260896.55	2.22	8	pass
0c16	18914	16794	16789	69.7	271.36	240.95	22874	19256	131072	72	64	6321082.30	2.24	0	pass
0b16	18914	16794	16789	63.66	297.11	263.81	30320	19310	131072	72	64	8885120.80	3.15	8	pass
0c4	18915	16795	16790	84.65	223.45	198.41	10299	12300	131072	24	64	2519608.59	0.89	8	pass
0bc8.4	20020	17900	17895	77.3	258.99	231.57	16894	14623	131072	40	64	4542013.98	1.61	16270	pass
0b8	20020	17900	17895	68.88	290.65	259.87	17055	14587	131072	40	64	5138929.25	1.82	16270	pass
0bc4.8	20239	18117	18112	83.7	241.8	216.45	8784	12310	131072	24	64	2420776.80	0.86	10600	pass
0b4	20239	18117	18112	75.55	267.89	239.8	10393	12259	131072	24	64	3067761.40	1.09	10600	pass
0bc4.4	20518	18398	18393	84.25	243.54	218.37	8815	12291	131072	24	64	2449019.55	0.87	24620	pass
0bc2.16	21965	19843	19838	85.97	255.5	230.81	6483	11066	131072	16	64	2013355.63	0.71	20346	pass
0b2	21965	19843	19838	83.04	264.51	238.96	6679	11093	131072	16	64	2131284.24	0.76	20346	pass
0c2	22243	20123	20118	80.93	274.84	248.65	6463	11054	131072	16	64	2164000.95	0.77	27660	pass
1c2	22244	20124	20119	87.23	255	230.7	6620	11170	131072	16	64	2044002.00	0.72	53268	pass
1bc2.2	22498	20378	20373	87.39	257.44	233.18	6475	11176	131072	16	64	2032163.70	0.72	49556	pass
0bc2.2	22504	20381	20376	88.72	253.65	229.72	6421	11062	131072	16	64	1989604.92	0.71	43792	pass
1bc2.4	22945	20811	20806	89.18	257.29	233.36	6488	11202	131072	16	64	2036766.08	0.72	31624	pass
1bc4.4	23159	21031	21026	87.71	264.04	239.78	8760	12474	131072	24	64	2675944.80	0.95	28867	pass
1b8	23195	21067	21062	79.54	291.61	264.86	12130	15026	131072	40	64	3933171.00	1.39	29219	pass
1bc8.4	23195	21067	21062	79.3	292.5	265.66	12529	14987	131072	40	64	4051049.34	1.44	29219	pass
0bc2.4	23527	21407	21402	86.81	271.02	246.6	6475	11054	131072	16	64	2149119.00	0.76	45780	pass
0bc2.8	23618	21496	21491	87.35	270.38	246.09	6544	11051	131072	16	64	2161654.56	0.77	32054	pass
1bc2.8	34947	32825	32820	91.12	383.53	360.24	6127	11132	131072	16	64	3014128.08	1.07	41048	pass
1b4	34947	32825	32820	88.71	393.95	370.03	8136	12443	131072	24	64	3898636.08	1.38	41048	pass
1bc4.8	34947	32825	32820	88.07	396.81	372.71	8290	12453	131072	24	64	3984269.90	1.41	41048	pass
1b2	34955	32833	32828	95.01	367.91	345.57	6161	11199	131072	16	64	2903133.57	1.03	50277	pass
1bc2.16	34955	32833	32828	89.09	392.36	368.54	6095	11119	131072	16	64	3071780.90	1.09	50277	pass
nopartition	34959	32837	32832	87.96	397.44	373.32	5317	10462	131072	16	64	2821179.24	1.00	65164	pass

Table A.4: *Matrixmult (cyclic)* benchmark full performance and area results.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsp	adprod	nadprod	contentions	pass/fail
0bc4.2	39455	6423	6419	84.14	468.92	76.34	2896	2443	1572864	192	0	514226.24	0.53	4	pass
0b4	39455	6423	6419	74.44	530.02	86.28	3086	2419	1572864	192	0	597575.28	0.61	4	pass
0c8	39455	6423	6419	83.78	470.94	76.67	5034	3787	1572864	176	0	655835.18	0.67	0	pass
0bc8.8	39455	6423	6419	78.46	502.87	81.86	5034	3794	1572864	176	0	700557.18	0.72	4	pass
0bc8.4	39455	6423	6419	76.92	512.94	83.5	5019	3792	1572864	176	0	713006.50	0.73	4	pass
0bc8.2	39455	6423	6419	75.84	520.24	84.69	5052	3795	1572864	176	0	725962.68	0.75	4	pass
0b8	39455	6423	6419	75.52	522.44	85.05	5311	3778	1572864	176	0	751076.55	0.77	0	pass
0bc16.4	39455	6423	6419	71.94	548.44	89.28	9974	6614	1572864	192	0	1233313.92	1.27	0	pass
0bc16.2	39455	6423	6419	71.28	553.52	90.11	9843	6554	1572864	192	0	1232975.13	1.27	0	pass
0c16	39455	6423	6419	71.45	552.2	89.9	9904	6616	1572864	192	0	1235585.60	1.27	0	pass
0b16	39455	6423	6419	71.9	548.75	89.33	10117	6518	1572864	192	0	1246778.81	1.28	0	pass
0c4	39456	6424	6420	80.33	491.17	79.97	2906	2438	1572864	192	0	539477.62	0.55	4	pass
0bc4.16	39582	6550	6546	79.5	497.89	82.39	2954	2428	1572864	192	0	559757.66	0.58	1020	pass
0bc4.8	39969	6937	6933	78.21	511.05	88.7	2923	2429	1572864	192	0	598878.10	0.62	4106	pass
0bc4.4	40490	7458	7454	82.65	489.9	90.24	2924	2435	1572864	192	0	610383.36	0.63	5569	pass
0c2	41244	8212	8208	86.78	475.27	94.63	1567	1717	1572864	192	0	511664.41	0.53	8051	pass
0bc2.16	41245	8213	8209	79.81	516.79	102.91	1587	1726	1572864	192	0	558492.57	0.57	14314	pass
1c4	41272	8238	8234	85.17	484.58	96.72	2887	2446	1572864	192	0	650635.44	0.67	5752	pass
0bc2.2	41375	8343	8339	84.14	491.74	99.16	1574	1725	1572864	192	0	536852.24	0.55	11634	pass
0bc2.32	41385	8352	8348	81.23	509.48	102.82	1587	1726	1572864	192	0	558004.14	0.57	15272	pass
0b2	41389	8356	8352	72.07	574.29	115.94	1824	1729	1572864	192	0	656684.16	0.67	15276	pass
0bc2.4	41518	8486	8482	85.02	488.33	99.81	1561	1730	1572864	192	0	539073.81	0.55	13367	pass
1c8	41677	8643	8639	80.37	518.56	107.54	5004	3799	1572864	176	0	916670.96	0.94	6860	pass
0bc2.8	41800	8766	8762	86.7	482.12	101.11	1570	1725	1572864	192	0	547005.10	0.56	8994	pass
1c2	42046	9012	9008	86.42	486.53	104.28	1569	1719	1572864	192	0	564050.52	0.58	8761	pass
1c16	42071	9037	9033	70.97	592.8	127.34	9700	6552	1572864	192	0	1724183.60	1.77	7926	pass
1bc4.2	42986	9952	9948	83.51	514.74	119.17	2894	2440	1572864	192	0	802490.78	0.82	9017	pass
1bc8.2	43008	9974	9970	75.11	572.6	132.79	4971	3787	1572864	176	0	1127519.89	1.16	9459	pass
1bc16.2	43213	10179	10175	72.87	593.01	139.69	9717	6562	1572864	192	0	1893777.33	1.95	8202	pass
1bc2.2	43535	10501	10497	88.07	494.32	119.23	1557	1723	1572864	192	0	643484.31	0.66	10387	pass
1bc8.4	44026	10992	10988	81.2	542.19	135.37	4979	3816	1572864	176	0	1150509.63	1.18	9206	pass
1bc16.4	44160	11126	11122	70.53	626.12	137.75	9737	6591	1572864	192	0	2141771.75	2.20	10707	pass
1bc4.4	44563	11529	11525	83.75	532.1	137.66	2886	2424	1572864	192	0	925901.16	0.95	11293	pass
1bc8.8	45620	12586	12582	81.97	556.55	153.54	4985	3791	1572864	176	0	1305857.70	1.34	12351	pass
1b16	45620	12586	12582	75.86	601.37	165.91	9675	6517	1572864	192	0	2242273.65	2.30	12351	pass
1bc2.4	45700	12666	12662	80.59	567.07	157.17	1560	1725	1572864	192	0	848718.00	0.87	12250	pass
1bc4.8	45988	12954	12950	82.55	557.09	156.92	2880	2452	1572864	192	0	1054502.40	1.08	12714	pass
1bc2.8	46860	13826	13822	86.76	540.11	159.36	1555	1719	1572864	192	0	859747.20	0.88	13039	pass
1bc2.16	48582	15548	15544	88.04	551.82	176.6	1567	1727	1572864	192	0	954876.20	0.98	15533	pass
1bc4.16	48582	15548	15544	82.76	587.02	187.87	2871	2429	1572864	192	0	1260795.57	1.30	15533	pass
1b8	48582	15548	15544	76.83	632.33	202.37	4951	3766	1572864	176	0	1714276.27	1.76	15533	pass
1bc2.32	49345	16311	16307	84.73	582.38	192.51	1551	1718	1572864	192	0	1037821.41	1.07	16291	pass
1b4	49345	16311	16307	78.25	630.61	208.45	2833	2433	1572864	192	0	1390986.85	1.43	16291	pass
1b2	49411	16377	16373	83.79	589.7	195.45	1623	1735	1572864	192	0	1067743.35	1.10	16360	pass
nopartition	49427	16395	16391	82.45	599.48	198.85	1055	1392	1572864	192	0	973370.75	1.00	16376	pass

Table A.5: *Matrixtrans* benchmark full performance and area results.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsps	adprod	nadprod	contentions	pass/fail
0bc4.2	39380	6348	6344	78.41	502.23	80.96	8338	9189	1572864	192	0	985930.88	0.66	4	pass
1bc4.4	39380	6348	6344	77.85	505.84	81.54	8314	9590	1572864	192	0	991037.16	0.66	4	pass
1bc4.8	39380	6348	6344	77.3	509.44	82.12	8342	9575	1572864	192	0	1000385.84	0.67	4	pass
1bc8.4	39380	6348	6344	74.06	531.73	85.71	11838	11066	1572864	176	0	1316334.18	0.88	0	pass
1bc8.2	39380	6348	6344	73.28	537.39	86.63	11882	11079	1572864	176	0	1334275.26	0.89	4	pass
1bc8.8	39380	6348	6344	71.65	549.62	88.6	11876	11067	1572864	176	0	1364085.60	0.91	4	pass
0c8	39380	6348	6344	70.64	557.47	89.86	11886	10517	1572864	176	0	1384383.16	0.93	0	pass
0bc8.2	39380	6348	6344	70.04	562.25	90.63	11857	10528	1572864	176	0	1393617.51	0.93	4	pass
1b16	39380	6348	6344	61.32	642.2	103.52	18331	16537	1573968	216	0	2344831.52	1.57	4	pass
0c16	39380	6348	6344	65.13	604.64	97.47	21569	13373	1572864	192	0	2476615.23	1.66	0	pass
1bc16.4	39380	6348	6344	65.75	598.94	96.55	22048	14114	1572864	192	0	2499486.40	1.67	0	pass
1c16	39380	6348	6344	64.91	606.69	97.8	22069	14125	1572864	192	0	2533900.20	1.69	4	pass
0bc16.2	39380	6348	6344	61.11	644.41	103.88	21619	13368	1572864	192	0	2644680.92	1.77	4	pass
1bc16.2	39380	6348	6344	61.5	640.33	103.22	21972	14018	1572864	192	0	2664314.64	1.78	0	pass
0c4	39382	6350	6346	76.03	517.98	83.52	8287	9169	1572864	192	0	1012847.04	0.68	6	pass
0bc8.4	39484	6452	6448	70.54	559.74	91.47	11854	10541	1572864	176	0	1406259.78	0.94	394	pass
0bc16.4	39579	6547	6543	61.72	641.27	106.08	21602	13327	1572864	192	0	2698887.36	1.80	725	pass
0bc4.8	39877	6843	6839	77.06	517.48	88.8	8305	9188	1572864	192	0	1078476.00	0.72	1827	pass
0bc8.8	40045	7011	7007	73.09	547.89	95.92	11854	10536	1572864	176	0	1474674.08	0.99	2101	pass
0bc4.4	40400	7366	7362	76.25	529.84	96.6	8333	9191	1572864	192	0	1175911.80	0.79	5098	pass
0b16	40849	7815	7811	62.87	649.74	124.3	21841	13323	1572864	192	0	3192148.30	2.13	4668	pass
0bc4.16	41026	7992	7988	81.7	502.15	97.82	8345	9189	1572864	192	0	1191936.70	0.80	5120	pass
1c8	41334	8302	8298	71.29	579.8	116.45	11877	11060	1572864	176	0	1792980.65	1.20	15626	pass
1bc4.2	41374	8342	8338	77.6	533.17	107.5	8347	9572	1572864	192	0	1310102.50	0.88	15946	pass
0c2	41393	8361	8357	80.21	516.06	104.24	5163	8552	1572864	192	0	938472.72	0.63	8118	pass
1bc2.4	41394	8362	8358	81.86	505.67	102.15	5242	8776	1572864	192	0	927726.30	0.62	16106	pass
1bc2.8	41398	8366	8362	81.79	506.15	102.29	5236	8759	1572864	192	0	928384.04	0.62	16138	pass
1bc2.16	41406	8374	8370	83.55	495.58	100.23	5223	8752	1572864	192	0	908384.49	0.61	16202	pass
1bc4.16	41406	8374	8370	79.13	523.27	105.83	8314	9584	1572864	192	0	1286257.32	0.86	16202	pass
1b8	41406	8374	8370	70.61	586.4	118.6	12088	13427	1573968	200	0	1908036.80	1.28	16202	pass
0bc2.2	41414	8379	8375	82.32	503.09	101.79	5173	8568	1572864	192	0	917433.27	0.61	12140	pass
0b8	41479	8445	8441	71.27	582	118.49	11991	10499	1572864	176	0	1837898.39	1.23	6553	pass
0bc2.4	41618	8586	8582	82.74	503	103.77	5209	8550	1572864	192	0	939014.73	0.63	10428	pass
0bc2.8	41930	8896	8892	81.01	517.59	109.81	5171	8552	1572864	192	0	989497.91	0.66	9375	pass
0bc2.16	42247	9213	9209	78.68	536.95	117.09	5165	8550	1572864	192	0	1054395.45	0.70	9205	pass
0b4	42715	9681	9677	79.68	536.08	121.5	8360	9158	1572864	192	0	1482300.00	0.99	8650	pass
0bc2.32	43208	10174	10170	81.17	532.31	125.34	5158	8561	1572864	192	0	1127809.32	0.75	9956	pass
0b2	45435	12401	12397	82.07	553.61	151.1	5627	8585	1572864	192	0	1430463.70	0.96	12168	pass
1c4	46136	13102	13098	75.67	609.7	173.15	8348	9588	1572864	192	0	2110352.20	1.41	12807	pass
1c2	47052	14018	14014	81.92	574.37	171.12	5219	8747	1572864	192	0	1550176.08	1.04	13079	pass
1bc2.2	48634	15600	15596	83.91	579.6	185.91	5208	8759	1572864	192	0	1682113.68	1.12	15421	pass
1bc2.32	49506	16472	16468	83.38	593.74	197.55	5219	8750	1572864	192	0	1789605.45	1.20	16292	pass
1b4	49506	16472	16468	69.94	707.84	235.52	8643	11917	1573968	216	0	3053045.76	2.04	16292	pass
nopartition	49516	16482	16478	91.53	540.98	180.07	4470	8389	1572864	192	0	1496381.70	1.00	16310	pass
1b2	49516	16482	16478	78.25	632.79	210.63	6688	11184	1573968	216	0	2318615.04	1.55	16310	pass

Table A.6: *Matratrans (block cyclic)* benchmark full performance and area results.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsps	adprod	nadprod	contentions	pass/fail
0c4	4129	4120	4117	93.46	44.18	44.08	2356	3392	131072	16	0	117958.02	0.41	0	pass
0bc8.2	4129	4120	4117	83.61	49.38	49.28	3639	4285	131072	16	0	195099.52	0.68	0	pass
0c8	4129	4120	4117	82.43	50.09	49.98	3625	4250	131072	16	0	197171.10	0.69	0	pass
0bc8.1024	4129	4120	4117	79.76	51.77	51.65	3761	4267	131072	16	0	210783.65	0.73	4	pass
0b8	4129	4120	4117	73.19	56.41	56.29	4161	4265	131072	16	0	252235.49	0.88	0	pass
0c16	4129	4120	4117	77.54	53.25	53.13	6398	5813	131072	16	0	356927.34	1.24	0	pass
0bc16.512	4129	4120	4117	74.9	55.13	55.01	6275	5826	131072	16	0	362790.95	1.26	4	pass
0bc16.2	4129	4120	4117	75.15	54.94	54.82	6475	5789	131072	16	0	372501.90	1.29	0	pass
0b16	4129	4120	4117	66.36	62.22	62.09	6976	5740	131072	16	0	453008.64	1.57	0	pass
0bc4.2048	4131	4122	4119	91.59	45.1	45	2504	3412	131072	16	0	127080.00	0.44	16	pass
0b4	4131	4122	4119	76.98	53.66	53.55	2814	3390	131072	16	0	167825.70	0.58	15	pass
0bc8.4	4725	4716	4713	81.49	57.98	57.87	3546	4291	131072	16	0	223725.42	0.78	4566	pass
0bc4.2	5149	5140	5137	92.29	55.79	55.69	2317	3402	131072	16	0	146854.53	0.51	8164	pass
0bc4.8	5262	5253	5250	90.13	58.38	58.28	2359	3379	131072	16	0	156132.12	0.54	5267	pass
0bc16.8	5506	5497	5494	75.92	72.52	72.41	6336	5789	131072	16	0	481960.96	1.67	5931	pass
0bc16.16	5703	5694	5691	76.32	74.72	74.61	6539	5790	131072	16	0	511749.99	1.78	6274	pass
0bc8.8	5784	5775	5772	79.73	72.54	72.43	3592	4273	131072	16	0	283346.16	0.98	7439	pass
0bc8.16	5870	5861	5858	83.98	69.9	69.79	3637	4298	131072	16	0	276159.03	0.96	6360	pass
0bc16.4	6639	6630	6627	74.12	89.57	89.45	6608	5793	131072	16	0	619709.60	2.15	22495	pass
0bc4.4	6683	6674	6671	90.9	73.52	73.42	2350	3373	131072	16	0	196031.40	0.68	23856	pass
0bc4.16	6865	6856	6853	92.18	74.47	74.38	2364	3379	131072	16	0	199635.92	0.69	9690	pass
0bc16.32	6948	6939	6936	75.76	91.71	91.59	6410	5755	131072	16	0	616400.70	2.14	9861	pass
0bc8.32	7678	7669	7666	83.87	91.55	91.44	3630	4248	131072	16	0	361188.00	1.25	9833	pass
0bc16.256	8087	8078	8075	73.35	110.25	110.13	6121	5808	131072	16	0	709347.33	2.46	16012	pass
0bc8.512	8174	8165	8162	82.27	99.36	99.27	3576	4293	131072	16	0	386678.00	1.34	16238	pass
0bc4.1024	8201	8192	8189	87.29	93.95	93.85	2380	3384	131072	16	0	253395.00	0.88	16318	pass
0bc2.2048	8217	8206	8203	93.41	87.97	87.85	1729	2962	131072	16	0	180004.65	0.63	16369	pass
0bc2.4096	8217	8207	8204	99.05	82.96	82.86	1727	2957	131072	16	0	169614.42	0.59	8186	pass
0bc2.8	8217	8208	8205	94.76	86.71	86.62	1716	2962	131072	16	0	176358.32	0.61	16367	pass
0bc2.2	8219	8209	8206	98	83.87	83.77	1690	2955	131072	16	0	168377.70	0.59	13807	pass
0b2	8223	8213	8210	81.95	100.34	100.22	2029	2961	131072	16	0	235416.78	0.82	16381	pass
0bc2.4	8228	8219	8216	94.59	86.99	86.89	1696	2953	131072	16	0	175170.24	0.61	31639	pass
0bc2.8	8276	8267	8264	95.44	86.71	86.62	1691	2954	131072	16	0	174192.82	0.61	31140	pass
0bc4.32	8497	8488	8485	92.63	91.73	91.63	2379	3386	131072	16	0	247309.37	0.86	10094	pass
0bc2.16	8823	8814	8811	98.02	90.01	89.92	1746	2958	131072	16	0	185774.72	0.65	14561	pass
0bc8.64	9051	9042	9039	82.67	109.48	109.37	3642	4280	131072	16	0	433323.94	1.51	10544	pass
0bc16.64	9171	9162	9159	74.64	122.87	122.75	6427	5767	131072	16	0	828194.25	2.88	10165	pass
0bc2.32	9492	9483	9480	96.27	98.6	98.5	1681	2953	131072	16	0	197098.50	0.68	14752	pass
0bc4.64	10261	10252	10249	93.67	109.54	109.45	2394	3367	131072	16	0	297047.30	1.03	12388	pass
0bc8.128	11669	11660	11657	85.2	136.96	136.85	3698	4266	131072	16	0	549863.30	1.91	17895	pass
0bc2.64	12045	12036	12033	98.15	122.72	122.63	1718	2958	131072	16	0	249919.94	0.87	16514	pass
0bc16.128	12098	12086	12086	73.82	163.89	163.76	6350	5773	131072	16	0	1092279.20	3.80	23194	pass
0bc4.128	12190	12181	12178	90.97	134	133.9	2388	3386	131072	16	0	362601.20	1.26	15935	pass
0bc2.128	13907	13898	13895	98.55	141.12	141.02	1685	2956	131072	16	0	282745.10	0.98	21477	pass
0bc4.256	14360	14351	14348	89.09	161.19	161.08	2383	3369	131072	16	0	435399.24	1.51	26715	pass
0bc8.256	14360	14351	14348	80.87	177.57	177.46	3569	4290	131072	16	0	690141.94	2.40	26715	pass
0bc2.256	14839	14830	14827	93.4	158.88	158.78	1741	2955	131072	16	0	327245.58	1.14	24864	pass
0bc2.512	16229	16220	16217	96.93	167.43	167.34	1742	2950	131072	16	0	345055.08	1.20	28375	pass
0bc4.512	16229	16220	16217	89.23	181.88	181.78	2371	3384	131072	16	0	489169.98	1.70	28375	pass
0bc2.1024	16376	16367	16364	99.32	164.88	164.79	1728	2959	131072	16	0	337489.92	1.17	24538	pass
nopartition	16405	16396	16393	96.96	169.19	169.1	1382	2758	131072	16	0	287808.20	1.00	16382	pass

Table A.7: *Substring* benchmark full performance and area results.

partition	fcyc	pcyc	mcyc	fmax	fwct	pwct	alms	regs	mbits	rams	dsps	adprod	nadprod	contentions	pass/fail
1bc16.2	79552	79543	79537	84.9	937.01	936.9	10165	12589	131072	32	16	10572916.50	1.78	787	pass
0c16	79566	79557	79551	81.13	980.72	980.61	11084	12692	131072	32	16	11967364.44	2.01	853	pass
1c16	79587	79578	79572	84.32	943.87	943.76	10178	12580	131072	32	16	10662600.48	1.79	990	pass
1b16	79591	79582	79576	73.82	1078.18	1078.05	10398	12752	131072	32	16	12416979.90	2.09	957	pass
0bc16.2	79602	79593	79587	78.17	1018.32	1018.2	11063	12677	131072	32	16	12404730.60	2.08	921	pass
1bc8.4	79661	79652	79646	91.73	868.43	868.33	8213	10246	131072	16	16	7826258.29	1.31	1618	pass
0b16	79681	79672	79666	71.84	1109.15	1109.02	11229	12650	131072	32	16	13695287.98	2.30	1427	pass
1b8	79726	79717	79711	84.05	948.55	948.45	8287	10451	131072	16	16	8618565.15	1.45	2001	pass
1bc8.2	79728	79719	79713	91.58	870.58	870.48	8212	10253	131072	16	16	7844765.76	1.32	1918	pass
0bc8.2	79737	79728	79722	87.81	908.06	907.96	8442	10316	131072	16	16	8391366.32	1.41	2009	pass
1c8	79746	79737	79731	90.86	877.68	877.58	8186	10263	131072	16	16	7885933.88	1.32	2117	pass
0bc8.4	79750	79741	79735	84	949.4	949.3	8434	10296	131072	16	16	8765836.20	1.47	1973	pass
0c8	79758	79749	79743	88.75	898.68	898.58	8485	10293	131072	16	16	8343315.30	1.40	1950	pass
0b8	79844	79835	79829	76.84	1039.09	1038.98	8914	10307	131072	16	16	10092651.72	1.70	2919	pass
1bc4.8	79929	79920	79914	93.41	855.68	855.58	6930	9089	131072	16	16	6613633.40	1.11	3419	pass
1bc4.4	79955	79946	79940	90.41	884.36	884.26	6944	9100	131072	16	16	6847709.44	1.15	3879	pass
1b4	80000	79991	79985	84.41	947.76	947.65	6967	9274	131072	16	16	7360397.55	1.24	4097	pass
1bc4.2	80004	79995	79989	99.52	803.9	803.81	6933	9101	131072	16	16	6215862.73	1.04	4360	pass
0bc4.4	80012	80003	79997	93.75	853.46	853.37	7124	9096	131072	16	16	6762103.88	1.14	4185	pass
0bc4.2	80016	80007	80001	92.7	863.17	863.07	7112	9098	131072	16	16	6828609.84	1.15	4291	pass
0c4	80024	80015	80009	94.8	844.14	844.04	7116	9106	131072	16	16	6681420.64	1.12	4042	pass
0bc4.8	80062	80053	80047	96.17	832.5	832.41	7125	9087	131072	16	16	6596849.25	1.11	4090	pass
1c4	80085	80076	80070	100.33	798.22	798.13	6936	9117	131072	16	16	6174333.68	1.04	4480	pass
0b4	80243	80234	80228	81.44	985.3	985.19	7245	9081	131072	16	16	7925853.55	1.33	5923	pass
1b2	80286	80277	80271	86.07	932.8	932.69	6402	8696	131072	14	16	6679925.78	1.12	6817	pass
1bc2.16	80536	80527	80521	100.04	805.04	804.95	6389	8486	131072	14	16	5754587.55	0.97	7540	pass
1bc2.8	80604	80595	80589	101.55	793.74	793.65	6405	8487	131072	14	16	5686502.25	0.96	8341	pass
0bc2.8	80681	80672	80666	101.64	793.79	793.7	6441	8499	131072	14	16	5715433.70	0.96	9123	pass
1c2	80704	80695	80689	102.03	790.98	790.89	6388	8485	131072	14	16	5653281.72	0.95	9109	pass
0c2	80722	80713	80707	103.59	779.25	779.16	6434	8500	131072	14	16	5605277.04	0.94	8764	pass
0bc2.16	80731	80722	80716	103.28	781.67	781.58	6431	8502	131072	14	16	5620341.78	0.94	9115	pass
0bc2.2	80763	80754	80748	98.9	816.61	816.52	6428	8503	131072	14	16	5869145.76	0.99	9281	pass
1bc2.4	80765	80756	80750	97.16	831.26	831.17	6357	8476	131072	14	16	5915436.89	0.99	9547	pass
1bc2.2	80786	80777	80771	102.07	791.48	791.39	6398	8493	131072	14	16	5664769.62	0.95	9586	pass
0bc2.4	80800	80791	80785	99.34	813.37	813.28	6442	8486	131072	14	16	5857242.56	0.98	9417	pass
0b2	80810	80801	80795	89.87	899.19	899.09	6489	8484	131072	14	16	6517503.41	1.10	9111	pass
nopartition	83226	83217	83211	98.42	845.62	845.53	6239	8198	131072	16	16	5951685.67	1.00	23397	pass

Table A.8: *Los* benchmark full performance and area results.

Bibliography

- [1] Altium. C-to-Hardware Compiler User Manual. url = <http://valhalla.altium.com/Learning-Guides/GU0122%20C-to-Hardware%20Compiler%20User%20Manual.pdf>, 2013.
- [2] Y. Ben Asher and N. Rotem. Automatic memory partitioning: Increasing memory parallelism via data structure partitioning. In *IEEE/ACM/IFIP CODES+ISSS*, pages 155–161, 2010.
- [3] Bluespec. Bluespec SystemVerilog Reference Guide. url = <http://bluespec.com/forum/download.php?id=157>, 2012.
- [4] C. Pilato and F. Ferrandi. Bambu. url = <https://panda.dei.polimi.it/?page.id=31>, 2017.
- [5] Cadence Design Systems, Inc. Stratus High-Level Synthesis. url = https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html, 2017.
- [6] Calypto. Catapult C Synthesis. url = http://calypto.agranderdesign.com/datasheets/Calytpo_CatapultC_DS.pdf, 2017.
- [7] A. Canis, J. Choi, and et al. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM/SIGDA FPGA*, pages 33–36, 2011.

- [8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2), 2013.
- [9] Y. T. Chen and J. H. Anderson. Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software. International Conference on Field-Programmable Logic, 2017.
- [10] J. Choi, J. Anderson, and S. Brown. From software threads to parallel hardware in FPGA high-level synthesis. In *IEEE FPT*, pages 270–279, 2013.
- [11] Jongsok Choi, Stephen Brown, and Jason Anderson. Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware. In *IEEE FPT*, pages 152–159, 2015.
- [12] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Trans. Archit. Code Optim.*, 11(4), January 2015.
- [13] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst.*, 16(2):15:1–15:25, April 2011.
- [14] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Trans. on CAD*, 30(4):473–491, 2011.
- [15] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *IEEE/ACM DAC*, pages 433–438, 2006.
- [16] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009.

- [17] Coussy, Philippe and Heller, Dominique. GAUT. url = <http://hls-labsticc.univ-ubs.fr>, 2016.
- [18] DJ. Greaves and S. Singh. Kiwi Scientific Acceleration using FPGA. url = <http://www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/manual/kiwic.pdf>, 2017.
- [19] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides. Area implications of memory partitioning for high-level synthesis on FPGAs. In *FPL*, 2014.
- [20] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data path allocation based on bipartite weighted matching. In *Proceedings of the 27th ACM/IEEE Design Automation Conference, DAC '90*, pages 499–504, New York, NY, USA, 1990. ACM.
- [21] Giang Nguyen Thi Huong and Seon Wook Kim. Gcc2verilog compiler toolset for complete translation of c programming language into verilog hdl. *ETRI Journal*, pages 731–740, 2011.
- [22] Impulse Accelerated Technologies. CoDeveloper. url = <http://www.impulseaccelerated.com/tools.html>, 2015.
- [23] Impulse Accelerated Technologies. Impulse C User Guide. url = <http://www.impulseaccelerated.com/ReleaseFiles/Help/ImpulseCUserGuide.pdf>, 2015.
- [24] LLVM Compiler Infrastructure. Llvm language reference manual. url = <http://releases.llvm.org/3.5.0/docs/LangRef.html>, 2014.
- [25] Intel Corporation. Intel High-Level Synthesis (HLS) Compiler. url = <https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html>.

- [26] Jacquard Computing Inc. ROCCC 2.0 User's Manual. url = <http://roccc.cs.ucr.edu/DOCUMENTATION-0.7.6/UserManual-0.7.6.pdf>, 2013.
- [27] N. Kavvadias and K. Masselos. Automated synthesis of fsmd-based accelerators for hardware compilation. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 157–160, July 2012.
- [28] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM CGO*, pages 75–88, 2004.
- [29] Maxeler Technologies. MaxCompiler White Paper. url = <https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf>, 2011.
- [30] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. Efficient memory partitioning for parallel data access in multidimensional arrays. In *IEEE/ACM DAC*, 2015.
- [31] Mentor. DK Design Suite. url = <https://www.mentor.com/products/fpga/handel-c/dk-design-suite>, 2017.
- [32] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. on CAD*, 35(10):1591–1604, 2016.
- [33] R. Nane, V.M. Sima, B Olivier, R.J. Meeuws, Y.D. Yankova, and K.L.M. Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *Proc. 22nd International Conference on Field Programmable Logic and Applications*, Oslo, Norway, August 2012.
- [34] NEC Corp. CyberWorkBench. url = <http://www.nec.com/en/global/prod/cwb>, 2017.

- [35] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *2008 International Conference on Field Programmable Logic and Applications*, pages 173–178, Sept 2008.
- [36] R. Rashid, J. G. Steffan, and V. Betz. Comparing performance, productivity and scalability of the tilt overlay processor to opencl hls. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 20–27, Dec 2014.
- [37] Synopsys, Inc. Symphony C Compiler. url = <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/symphony-c-compiler.html>, 2017.
- [38] Justin L. Tripp, Maya B. Gokhale, and Kristopher D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007.
- [39] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *IEEE Int’l Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, 2010.
- [40] Y. Wang, P. Li, P. Zhang, Chen Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *ACM/IEEE DAC*, 2013.
- [41] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *ACM FPGA*, pages 199–208, 2014.
- [42] H. S. Warren. *Hacker’s Delight*. Pearson Education, 2013.
- [43] Xilinx Inc. Vivado Design Suite User Guide. url = https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf, 2017.

- [44] Xilinx Inc. Vivado HLS. url = <https://www.xilinx.com/products/design-tools/vivado/integration.html>, 2017.
- [45] Y Explorations. eXCite. url = <http://www.yxi.com/products.php>, 2013.
- [46] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. A new approach to automatic memory banking using trace-based address mining. In *ACM FPGA*, pages 179–188, 2017.