Floating-Point Unit Reuse in an FPGA Implementation of a Ray-Triangle Intersection Algorithm

Craig Ulmer
Sandia National Laboratories*
Livermore, California USA
cdulmer@sandia.gov

Adrian Javelo
UCLA EE Department
Los Angeles, California USA
ajavelo@ucla.edu

ABSTRACT

The recent emergence of high-quality floating-point libraries for FPGAs has sparked a renewed interest in accelerating scientific applications through Reconfigurable Computing (RC) techniques. Unfortunately, the sheer size of these floating-point units makes it difficult to house a large number of units in a single FPGA. In order to support the adaptation of non-trivial algorithms to hardware, it is therefore necessary to consider methods by which a set of floating-point units can be reused to perform different operations in an algorithm.

In this paper we discuss a "recycling architecture" that reuses a fixed number of floating-point units to implement an algorithm. We customize the hardware data path for this architecture at compile time based on a static computational schedule that is generated for an algorithm. As a means of illustrating tradeoffs, we step through the adaptation process with an example application that computes ray-triangle intersection points. By reusing hardware, we are able to halve resource requirements while maintaining acceptable performance. As a means of motivating future work, we also discuss our experiences constructing tools that translate an algorithm's equations into a synthesizable netlist.

Keywords: FPGA, Floating-Point Reuse

1. Introduction

1.1 Reconfigurable Computing

Reconfigurable Computing (RC) [1] refers to the practice of utilizing reconfigurable hardware devices to accelerate the computational performance of a system. In RC, an application's performance-critical computational kernels are adapted to function as digital circuitry that can be emulated by a reconfigurable hardware device. By implementing the kernels in efficient hardware, it is possible to obtain significant performance speedups over solutions that implement an algorithm entirely in software. A number of RC researchers have reported improvements in many application domains, including cryptography [2], signal processing [3], and pattern matching [4].

Field-Programmable Gate Arrays (FPGAs) are the dominant form of reconfigurable hardware utilized by RC researchers, due to the wide availability of commercial products and the broad flexibility of the hardware. Modern FPGAs such as the Xilinx Virtex II/Pro (V2P) [5] have the capacity to emulate hardware designs that are comprised of millions of logic gates. Additionally, these "platform FPGAs" feature large amounts of internal memory, dedicated integer multiplication units, and embedded processors that can be utilized in a flexible manner.

1.2 Floating-Point Operations in FPGAs

While commercial FPGAs have improved significantly over the last decade, they still lack native support for floating-point calculations. As a result, RC researchers that use FPGAs in scientific work have either (1) adjusted their applications to use fixed-point calculations, or (2) implemented their own floating-point units using FPGA logic. Implementing a floating-point unit is non-trivial and requires a sizable amount of FPGA resources to even be feasible. However, the availability of high-capacity FPGAs has renewed interest in building floating-point libraries for FPGAs [6,7].

One such library for the Xilinx V2P FPGA has been developed [8] by Keith Underwood and K. Scott Hemmert at Sandia National Laboratories. This library includes a diverse set of floating-point operations that are available in both single- and double-precision. The units are deeply pipelined (10 to 20 stages) and operate at high clock speeds (up to 200 MHz). For this

paper, we make use of single-precision adders and multipliers available in the library. However, this work is applicable for use with other floating-point libraries.

1.3 Algorithm Adaptation: Full Pipeline

The simplest means of adapting an algorithm to hardware is to allocate a floating-point unit for each computation in the algorithm. This adaptation results in a long, continuous pipeline that we refer to as a full pipeline. For example, consider the trivial case where an algorithm computes y=(a+b)+(c+d). A full-pipeline implementation would employ two adders to compute (a+b) and (c+d) in parallel, and then route the results to a third adder to generate y. In addition to being straightforward to implement, this approach offers excellent performance because all floating-point units operate in parallel once the pipeline fills. Users can issue a new algorithmic operation every clock cycle. Results are produced after a delay that is equivalent to the longest path through the system (2*N clock cycles in this example, where N is the number of pipeline stages in an adder).

2. Reusing Floating-Point Units

While full-pipeline implementations provide excellent performance, they are impractical for general use due to two reasons. First, current generation FPGAs only have enough capacity to house a small number of floating-point units (i.e., less than 100). Therefore, full-pipeline implementations are generally only possible for algorithms with a small number of operations. Second, practical RC systems have finite memory resources from which input values are supplied. As such, it is possible that an FPGA would have the capacity to house a full pipeline, but insufficient bandwidth to supply new input data values to the pipeline every clock cycle.

In order for RC to move forward in the scientific community, it is necessary to develop alternative design techniques that enable large-scale algorithms to be mapped to resource-constrained hardware. Our solution to this problem is to construct a hardware architecture that reuses a set of floating-point units over time to perform different operations that are required by the algorithm. This hardware is customized to the algorithm's data flow, and is referred to as a *recycling architecture* in this paper.

2.1 The Recycling Architecture

As depicted in Figure 1, the recycling architecture is comprised of four components: (1) an array of floating-point units, (2) an input selection unit to route data values to the right units at the right times, (3) an intermediate buffering unit to store results for later use, and (4) a control unit for managing the architecture on a cycle-by-cycle basis.

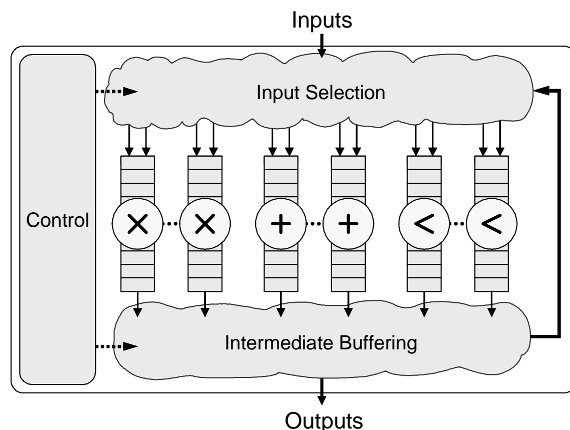

Figure 1: The Recycling Architecture

The concept of reusing floating-point units through a recycling architecture is attractive for multiple reasons. First, this approach provides us with a means of executing large algorithms on FPGAs that have limited floating-point and/or memory bandwidth resources. Second, the recycling architecture shares similarities with the architecture of a general-purpose CPU, and therefore it is possible to leverage existing techniques for maximizing the hardware's utilization. Finally, unlike CPUs, the recycling architecture's hardware data path can be customized to the characteristics of the algorithm. This customization can include adjustments to the number of floating-point units, the amount of on-chip buffering, and the manner in which resources are interconnected.

2.2 Ray-Triangle Intersection Example

The tradeoffs associated with developing a recycling architecture are best illustrated with a practical example. In this paper we focus on the adaptation of a computational kernel that computes the intersection point between a ray and a triangle. We selected this kernel because it is frequently required in visualization applications

such as photon mapping [9]. Photon mapping applications inject millions of photons into a scene and then track the photons' paths as they reflect off objects. A ray-triangle intersection algorithm is used to determine which object a photon will hit next and where the collision will take place. The sheer number of photons and objects in a scene highlight the importance of performing the intersection computation as rapidly as possible.

The Moller-Trumbore [10] algorithm is a well-known method for efficiently computing the ray-triangle intersection point. Given a ray and a triangle, the algorithm produces a TUV intersection point, where T is the distance from the ray origin to the triangle and UV is the offset of the intersection within the triangle. The algorithm exclusively utilizes additions, multiplications, and comparisons, except for one division that is used to scale the final TUV values.

For our work we assume that the ray-triangle intersection algorithm will be used in a photon mapping context, where it is necessary to compute the intersection point of each photon with each triangle. In order to minimize the work performed in the inner loop, we have modified the algorithm to defer the final division and scaling operation until all intersections have been computed. This optimization requires comparisons to be performed using a numerator/denominator notation, but removes two stages of floating-point computation from the critical path.

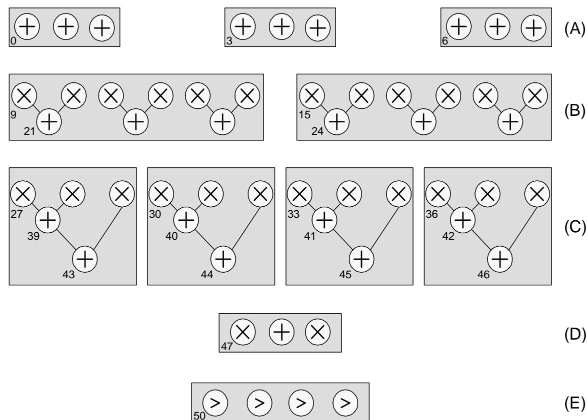

Figure 2: Ray-Triangle Intersection Data Flow

A high-level representation of the modified algorithm implemented in this work is depicted in Figure 2. Data flows from top to bottom in this diagram, with results generated by one row of boxes being used in one or more of the lower boxes. The rows of boxes perform (A) three vector subtractions, (B) two dot products, (C) four cross products, (D) numerator-denominator cross multiplication, and (E) comparisons to determine if the intersection is closer than the current value. We reference individual operations by a numerical identifier that is assigned based on the operation's location in the diagram. In terms of floating-point operations, the algorithm requires 26 multiplies, 24 adds, and 4 compares.

2.3 Hardware Environment

We selected the Xilinx V2P20 FPGA as the target FPGA for this study. This decision was motivate by the availability of a V2P20 reference board for experiments. While the V2P20 is relatively small by current standards, it features enough capacity to house 10-20 single-precision floating-point units. Working with a small part is beneficial in that it motivates the designer to implement the hardware as compactly as possibly. We note that for larger FPGAs, the design can simply be replicated in order to improve performance.

An initial investigation was performed to determine how many of each type of floating-point unit should be utilized in the design. From Figure 2 we see that the algorithm performs 12 multiplies, 9 adds, and 4 compares at most in any row. After estimating that only half the chip should be allocated for floating-point units, we decided to constrain the recycle architecture to employ 6 multiply, 5 add, and 4 comparison units.

2.4 Related Work

Our work draws upon the insight provided by a variety of related research projects. In terms of scheduling, the compiler community has developed a number of techniques that are directly applicable in this work, such as modulo scheduling [11] and register coloring [12]. However, it is clear that software compiler efforts have different constraints than hardware developers. Most notably, software compiler efforts typically assume that the hardware is predefined and cannot be changed. Hardware developers on the other hand have the ability to

adjust both the schedule of operations and the hardware data path that executes those operations.

Hardware compilers have been discussed a great deal in the literature. Prominent efforts such as StreamsC [13] and Handel-C [14] have produced tools that convert C-like languages into FPGA hardware. Hardware compilers typically exploit parallelism by instantiating as many computational units as possible to implement an application's operations. This technique works well for integer applications because integer operations can be implemented compactly in an FPGA. However, the sheer size of floating-point hardware limits the effectiveness of this technique for floating-point applications. As such, most high-level compilers for FPGAs do not support floating-point operations. While recent efforts such as Trident [15] seek to remedy this shortcoming, we believe that floating-point will be a central challenge in RC for several years.

We differentiate our work from high-level compiler efforts based on our focus. Rather than build a general-purpose hardware compiler, we are focused on techniques and tools for creating complex computational kernels that can be used as components in other hardware designs. While an optimizing compiler is desirable, we expect that there we will always need to perform a moderate amount of design and optimization by hand in hardware development.

2.5 Paper Organization

The remainder of this paper steps through the process of adapting the ray-triangle intersection algorithm to a recycling architecture implementation. Sections 3-5 describe the construction of a computational schedule for the algorithm, the mapping of operations to specific units, and the design of hardware to buffer intermediate values. Section 6 provides performance estimates for the implementation, while Section 7 outlines opportunities for automating the design process. Finally, we conclude the paper with a brief summary of our experiences.

3. Operation Scheduling

At the heart of the recycling architecture is an array of floating-point units. These units are used to perform the computations needed by an algorithm at different points in time. As such, the first task in mapping an algorithm to a recycling architecture is to generate a computational schedule that sequences the flow of execution for an algorithm on the hardware.

3.1 Pipeline Challenges for Scheduling

There are two characteristics of floating-point unit pipelines that make scheduling challenging. First, different floating-point operations have different pipeline latencies (e.g., the adder used in this work is 10 stages long while the multiplier is 11). As a result, the designer must perform temporal alignment when different types of operations are executed in parallel. Second, there is a sizable amount of delay between when an operation is issued and when the result is produced. This delay makes it challenging for the scheduler to find enough work to keep the units utilized unless the algorithm exhibits a significant amount of parallelism.

In order to simplify the task of scheduling, we (1) add buffering to the floating-point units so that all operations have the same number of pipeline stages (i.e., 11) and (2) employ a strip-mining technique to maximize resource utilization. With this technique, we sequentially process a strip of N independent iterations of an algorithm at a time, where N is equivalent to the number of stages in the pipeline. This technique enables us to perform scheduling on a strip-by-strip basis that hides the fact that the floating-point units are deeply pipelined

3.2 A Single-Strip Schedule

As a first step in scheduling the ray-triangle intersection algorithm, we constructed a computational schedule that processes a single strip of 11 iterations at a time. We refer to the amount of time required to perform a single operation for a strip of values as a single *stage* of execution in the schedule. Modulo scheduling was applied to enable the back-to-back processing of multiple strips of data to take place more efficiently. The schedule is listed in Table 1. Each numerical identifier in the table refers to a particular operation in Figure 2.

Table 1: Single-Strip Schedule

Stage	Add	Multiply	Compare
0	0-4		
1	5-8		
2		9-14	
3	21-23	15-20	
4	24-26		
5		30-32, 36-38	
6	40,42	27-29, 33-35	
7	39,41, 44,46		
8	43,45,48		
0		47,49	
1			50-53
2			(output)

The single-strip schedule requires 12 stages of execution to produce all the output values for a single strip of 11 iterations, resulting in a total execution time of 132 clock cycles. Utilization for the 5 adders and 6 multipliers is 40% and 36% respectively. Should multiple single-strips of data be processed back-to-back, the overlap provided by modulo scheduling increases utilization to 53% and 48%.

3.3 A Double-Strip Schedule

The low resource utilization of the single-strip schedule motivated us to consider a modified schedule that processes two strips of data (i.e., 22 iterations) in a single pass. In order to include the second strip of operations, the first strip's schedule had to be relaxed and the overall schedule lengthened. The resulting double-strip schedule is presented in Table 2. Each identifier in the table refers to a particular operation in Figure 2. Operations that are for the second strip of data are listed in brackets.

While this schedule requires three extra stages of execution, it processes twice as many values as the single-strip schedule. Utilization for processing a double-strip of computations is 64% for the adders and 57% for the multipliers. However, if multiple double-strip computations are processed back-to-back, this utilization increases to 80% and 72%.

Table 2: Double-Strip Schedule

Stage	Add	Multiply	Compare
0	0-4		
1	5-8		
2	[0-4]	9-14	
3	[5-8]	15-20	
4	21-23	[9-14]	
5	24-26	[15-20]	
6	[21-23]	30-32, 36-38	
7	40,42 [24-26]	27-29, 33-35	
8	39,41, 44,46	[30-32, 36-38]	
9	43,45,48 [40,42]	[27-29, 33-35]	
10	[39,41, 44,46]	47,49	
11	[43,45,48]		50-53
0		[47,49]	(output)
1			[50-53]
2			[(output)]

3.4 Schedules with Additional Strips

It is possible to continue unrolling the algorithm further in order to process more strips of data in a single pass of work. As the double-strip schedule illustrates, the advantage of increasing the number of iterations processed in a pass enables the designer to backfill empty slots in the schedule and achieve higher utilization of the computational hardware.

However, increasing the number of strips processed in a single pass also increases the complexity of the hardware that must be generated to implement the schedule. This complexity can have a negative impact on the maximum clock rate of the system (e.g., see Table 4). Given that the double-strip schedule achieves respectable utilization, we did not unroll the loop beyond two strips (22 iterations).

4. Mapping Operations to Units

Once a designer has created a computational schedule, the next step in building a recycling architecture is to map individual operations to specific floating-point units. This mapping has a direct effect on the input selection unit's hardware, which is responsible for routing data values to the input ports of the floating-point units. Therefore, it is beneficial to examine how mapping optimizations can be applied to yield hardware that is more efficient.

4.1 Input Selection Unit

Our approach to implementing an input selection unit for a particular algorithm is to generate application-specific hardware that routes data values to the input ports of the floating-point units. As illustrated in Figure 3, this routing can be accomplished through the use of data buses and multiplexers. Each input port for a floating-point unit is equipped with a multiplexer that is connected to a subset of the available data buses. At runtime, configuration data is supplied to each multiplexer in order to perform the necessary routing. Multiplexer configuration data is defined at compile time based on the data flow of a mapped schedule.

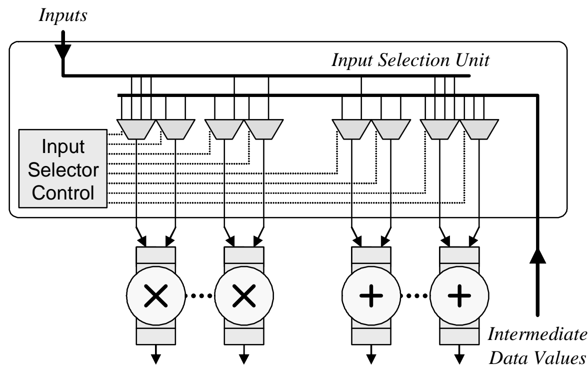

Figure 3: The Input Selection Unit

The mapping of a schedule's operations to specific floating-point units affects the routing that the multiplexers must provide, which in turn affects the input selection unit's overall performance. Therefore it is beneficial to map operations in a way that balances the workload and minimizes the size of the largest multiplexer in the input selection unit. Our strategy for creating an acceptable mapping is based on a two-part heuristic. First, we step through a schedule and assign operations to units based on input similarities. Second, an operation's inputs are swapped if doing so reduces the total number of unique sources that are required by the unit's input ports. In the case of a subtraction operation, the sign bits are swapped accordingly.

4.2 Balancing the Double-Strip Schedule

The double-strip schedule's operations were assigned to floating-point units using our mapping strategy. For comparison, a second mapping was generated using a first-come-first-serve (FCFS) approach that simply flood-fills the units. The distribution of multiplexers in the input selection unit for each mapping strategy is presented in Table 3. The heuristic approach decreases the largest multiplexer size from 7 in FCFS to 5 in our heuristic. Additionally, the heuristic approach's distribution favors smaller multiplexer sizes. These experiments demonstrate that it is possible to adjust the hardware requirements for the input selection unit simply by assigning computations to floating-point units in a more logical manner.

Table 3: Multiplexer Distribution

Multiplexer Size	Number of Multiplexers	
	FCFS	Heuristic
7	2	0
6	3	0
5	10	4
4	5	10
3	2	8

5. Intermediate Values

The final task in converting an algorithm to a recycling architecture is to implement a unit that is capable of buffering intermediate data values. This unit captures results generated by the floating-point unit and supplies the values back to the input selection unit as needed by the schedule. At first glance it would appear as though it would be best to buffer intermediate values using on-chip block RAM (BRAM). However, while BRAMs provide a convenient means of storing large amounts of data, a single BRAM only provides two access ports for exchanging data. As such, BRAM bandwidth is likely to be insufficient for schedules that fetch a moderate number of independent intermediate values during execution.

Our approach is to instead implement all intermediate buffering with registers. Modern FPGAs can easily support hundreds of registers that can be accessed independently by the input selection unit. In order to handle the fact that the recycling architecture processes a strip of 11 data values per execution stage, we place 11 registers in series and refer to the memory as a *delay block*. Delay blocks are implemented compactly in the V2P architecture through the SRL16 primitive. For this work, we consider two

strategies for utilizing the delay blocks: one where the delay blocks are independently writable and another where a chain of delay blocks is associated with a specific floating-point unit.

5.1 Two Buffering Strategies

One approach to buffering intermediate data values is to allocate a number of independent delay blocks that can be written to by different floating-point units. At runtime a floating-point unit writes a strip of results to a pre-specified delay block, which in turn preserves the data until it is required later in the schedule. As illustrated in Figure 4, routing data between the floating-point units and the delay blocks can be performed by equipping each delay block with a multiplexer that is connected to relevant floating-point units. The advantages of this buffering strategy are that delay blocks can be reused and that register coloring techniques can be applied to minimize the total number of delay blocks in the system.

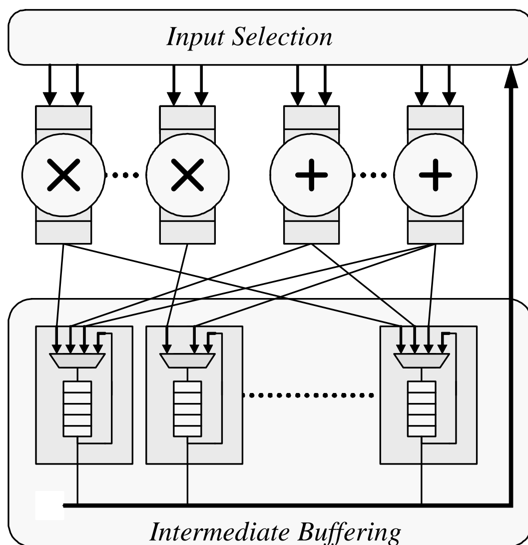

Figure 4: Independently-Writeable Delay Blocks

An alternate approach to buffering data values is to assign a chain of delay blocks to each floating-point unit, as Figure 5 depicts. The number of delay blocks in a chain is equivalent to the largest number of execution stages that a strip of results must be buffered for the floating-point unit. While this approach does not permit delay blocks to be shared between floating-point units, there are multiple benefits that arise from its simplicity. First, the delay blocks do not require input multiplexers and are therefore smaller and faster than the previous approach. Second, a chain's delay blocks do not require explicit management at runtime, which simplifies the control hardware for the system. Finally, the pipelined nature of the delay chain implies that delay units are automatically reused temporally.

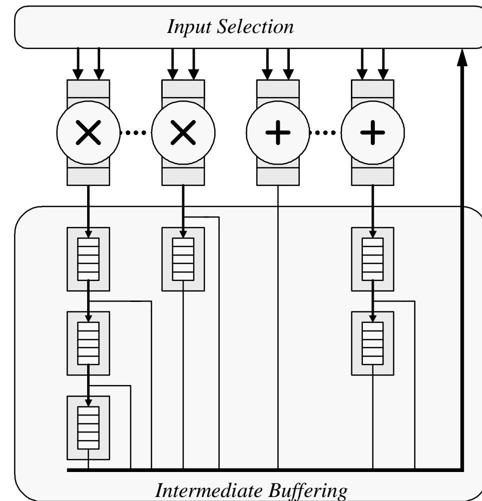

Figure 5: Chaining Delay Blocks

5.2 Delay Block Implementation

Two designs were constructed to observe the performance differences between the independently-writable and chaining strategies for buffering intermediate data values. In terms of resources, the independently-writable strategy required 40 delay blocks while the chaining strategy required 81. The longest chain of delay blocks required in the chaining approach was 6 for the adders and 1 for the multipliers. While the chaining approach requires twice as many delay blocks, synthesis results indicated that the overall design was 6% faster and 19% smaller than the design that employed the independently-writable strategy. These benefits indicate that a simple but brute-force approach can yield better results than a complex but intelligent approach. As such, we employ chaining for buffering intermediate values in the remainder of this paper.

6. Performance

We implemented complete recycling architecture hardware designs for both the single- and double-strip versions of the ray-triangle intersection algorithm. Following simulation experiments to confirm that these designs were functionally correct, we compiled the designs to hardware and tested them on an Avnet Virtex II/Pro Development Kit [16]. This board features a Xilinx V2P20 FPGA that has enough capacity to house small floating-point designs. The designs were compiled using the Xilinx ISE 6.3 tool chain, which performs synthesis through the Xilinx Synthesis Tool (XST).

For comparison purposes, we also implemented the ray-triangle intersection algorithm as a full pipeline of 54 floating-point operations. This pipeline is 74 stages deep and produces an output every clock cycle once the pipeline is filled with data. Due to the capacity constraints of the V2P20, the full-pipeline design was built targeting the larger V2P50 FPGA. Area estimates for this design are scaled to be in terms of V2P20 capacity for relevancy.

6.1 Build Results

The build results for the three designs are presented in Table 4. Area estimates are based on the percentage of a V2P20's available slices that are utilized by a design. Speed refers to the maximum clock rate at which the design can operate in a part with a -7 speed grade. The performance estimate is determined by multiplying the average number of floating-point operations that are performed per clock period by the design's maximum clock rate.

Several observations can be made from the build results. First, the recycling architecture enables a large design to be implemented in an FPGA that does not have the capacity to house the full pipeline in its entirety. Second, a significant performance drop is incurred when moving away from the full pipeline. This drop is to be expected because the full pipeline has 100% utilization and allows more computations to be in flight at the same time since it has nearly *five times* as many floating-point units as the recycling architecture. However, the performance of the recycling architectures is still roughly equivalent to performance observed on a host CPU. Finally, the recycling architecture designs are capable of operating at higher speeds than the full pipeline. While it is likely that the full pipeline's clock rate could be improved, doing so involves a fair amount of fine tuning through floor-planning tools.

Table 4: Design Performance Measurements

Design	Area (V2P20)	Speed (MHz)	Performance (GFLOPS)
Single-Strip	70%	155	0.9
Double-Strip	79%	148	1.2
Full Pipeline	199%	142	7.1

6.2 Memory Bandwidth

In addition to area and performance, another metric by which the full pipeline and recycling architectures can be compared is memory bandwidth requirements. Each computation performed by the ray-triangle intersection algorithm requires a total of 17 floating-point inputs (i.e., 68 bytes). In order to keep the full-pipeline design saturated, this input data must be supplied *every* clock cycle. While many FPGA platforms support multiple banks of memory, few are able to sustain this performance. In comparison, the double-strip schedule requires 2x17x11 floating-point values to be fetched every 12x11 clock cycles. As such, this recycling architecture must read 3 floating-point values (12 bytes) on average every clock cycle. This data rate is much more manageable and can easily be implemented on many FPGA platforms.

7. Future Work: Automation

Having explored several design options for adapting a single floating-point algorithm to hardware, the next logical step in this work is to automate the development process. As a means of working towards this goal, we have constructed an initial set of data-flow graph tools. These tools (1) translate a set of equations to a computational graph, (2) determine an optimal computational schedule for a specified number of units, (3) locate an optimal mapping of operations to units, and (4) generate a synthesizable VHDL netlist for the hardware.

7.1 Design-Space Tradeoffs

The data-flow graph tools were used to conduct a broader study of design tradeoffs for the ray-triangle intersection algorithm. In this experiment we removed the strip-mining constraint and varied the number of floating-point units available in the architecture. We then measured the total number of clock cycles required to process 128 iterations of the ray-triangle intersection algorithm.

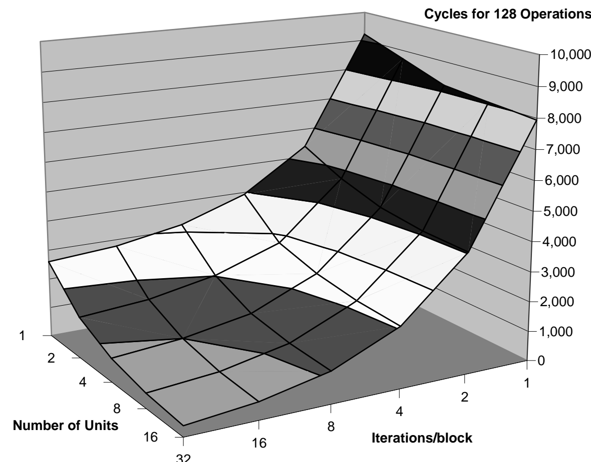

Figure 6: Performance Tradeoffs

The results of the experiment are presented in Figure 6. From these results, we observe that performance improves as we increase the number of floating-point units in the system or the number of iterations processed in a schedule. Of the two, increasing the number of iterations has more of an effect, due to the fact that the floating-point units are deeply pipelined. Another important observation is that it is possible to achieve a performance goal through a combination of these two options. This characteristic is particularly useful when the FPGA can only house a small number of floating-point units.

8. Summary

Floating-point units consume significant resources in modern FPGAs. Therefore it is beneficial to develop methodologies by which floating-point units can be reused to perform different computations required by an algorithm. In this paper we have described a design technique for mapping large floating-point algorithms to FPGAs that is based on the notion of a recycling architecture. By customizing the data path of this architecture and optimizing the computational schedule, it is possible to implement non-trivial algorithms on resource-constrained FPGA platforms.

The authors thank K. Scott Hemmert and Keith Underwood for allowing us to use their floating-point libraries in this work.

References

[1] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," in ACM Computing Surveys, Vol. 34, No. 2, June 2002.

[2] T. Wollinger, J. Guajardo, and C. Paar, "Security on FPGAs: State-of-the-Art Implementations and Attacks," in ACM Transactions on Embedded Computing Systems, Vol. 3, No. 3, August 2004.

[3] A. Shoa and S. Shirani, "Run-Time Reconfiguration Systems for Digital Signal Processing Applications: A Survey," in Journal of VLSI Signal Processing, Vol. 39, No. 3, 2005.

[4] C. Clark and D. Schimmel, "Modeling the Data-Dependent Performance of Pattern-Matching Architectures," in proceedings of International Symposium on Field-Programmable Gate Arrays, 2006.

[5] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X Platform FPGAs Data Sheet," 2005.

[6] B. Catanzaro and B. Nelson, "Higher Radix Floating-Point Representations for FPGA-Based Arithmetic", in Field-Programmable Custom Computing Machines 2005.

[7] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. Hemmert, "A Comparison of Floating Point and Logarythmic Number Systems for FPGAs," in Field-Programmable Custom Computing Machines 2005.

[8] M. Beauchamp, S. Hauck, K. Underwood, and K. Hemmert, "Embedded Floating-Point Units in FPGAs," in proceedings of International Symposium on Field-Programmable Gate Arrays, 2006.

[9] H. Jensen, *Realistic Image Synthesis Using Photon Mapping*, A.K. Peters, Ltd., Massachusetts, 2001.

[10] T. Moller and B. Trumbore, “Fast, minimum storage ray-triangle intersection,” Journal on Graphics Tools, Vol.2, No.1, 1997.

[11] B. Rau, “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops,” in International Symposium on Microarchitecture, 1994.

[12] G. Chaltin, “Register Allocation and Spilling via Graph Coloring,” in Proceedings of Symposium on Compiler Construction, 1982.

[13] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, “Stream-Oriented FPGA Computing in the Streams-C High-Level Language,” in proceedings of Field-Programmable Custom Computing Machines 2000.

[14] Celoxica Inc., “Handel-C Language Overview”

[15] J. Tripp, K. Peterson, C. Ahrens, J. Poznanovic, and M. Gokhale, “Trident: An FPGA Compiler Framework for Floating-Point Algorithms,” in Field Programmable Logic and Applications, 2005.

[16] Avnet Design Services whitepaper, “Xilinx Virtex-II Pro Development Kit,” 2003.