Lost in Space! Quantifying the Elements of FPGA Speedup

Scott Sirowy and Alessandro Forin
Microsoft Research, Redmond, WA, USA

Abstract

*Where do all the cycles go when microprocessor applications are implemented spatially as circuits on an FPGA? It is well established that certain sequential applications can be captured spatially and achieve breathtaking speedups when run on an FPGA, but **why**? Despite running at clock speeds orders of magnitude slower compared to their embedded processor equivalents, FPGA applications can "lose" enough cycles to create exceptionally fast spatially-oriented circuits. We profile and analyze three canonical applications amenable to FPGA speedup to quantify exactly where FPGAs gain that speedup. We compare the FPGA implementations to several idealized software platforms. The idealized software platforms give insight as to how FPGA implementations attain such dramatic speedups. We quantify the effects of parallelizing and pipelining instructions, streaming data, and eliminating the instruction fetch, showing exactly where the cycles are lost in an FPGA implementation. The results, though not surprising, provide a clearer and more intuitive understanding of the performance FPGAs can achieve, offering researchers and engineers alike a new angle to attack the task of parallelizing applications.*

1. Introduction

"…Therefore, we were able to attain 10,000X speedup over the fastest software implementation using our novel FPGA implementation …" And so concludes a typical researcher who ported a software application to a field-programmable gate-array (FPGA). Invented in the 1980s, FPGAs are custom computing elements that allow designers to create highly efficient, custom circuit accelerators. In fact, certain application domains have been known to achieve extraordinary speedups when implemented on an FPGA, the results being extensively reported [3][18][19], and with several conferences dedicating forums to FPGA application speedup[9][10].

Qualitatively, the reasons for FPGA speedups are clear. FPGAs can expose parallelism at many different levels, from the bit and instruction level all the way to the loop and task level. A typical von-Neumann computer must fetch an instruction from memory, execute the instruction, and store the result. An FPGA implementation instead thrives on executing multiple (often 10s of) instructions in one clock cycle, and ridding itself of fetching instructions since they are built into the FPGA data path itself. FPGAs implementations can also implement deep pipelines, enabling highly efficient, high throughput circuits that output results every clock cycle. Finally, FPGAs use specialized custom interfaces to memory banks that make the best use of the memory bus for that application.

Quantitatively, the reasons for FPGA have not been completely unfolded. *Why* is it that despite a clock speed often orders of magnitude slower than the microprocessor an FPGA is able to achieve orders of magnitude speedup on the application in both latency and throughput compared to the software implementation? How much speedup can we gain by unrolling a software loop once? Twice? What about pipelining the circuit? What effect does the memory interface have on that pipeline? On the loop unrolling? Quantifying the effects of such optimizations can provide a deeper understanding for applying the same optimizations to other FPGA applications.

In [12], the authors perform an extensive quantitative analysis of FPGA speedups on several simple image processing applications, comparing to baseline MIPS, Pentium, and VLIW platforms. The authors generalize the speedup factors accounted for into a cohesive speedup model, while Dehon [7] develops the concept of computational density to quantitatively analyze the differences between CPUs and FPGAs (as well as ASICs).

We quantify three applications known to achieve high performance when implemented on an FPGA, including an N-body simulation, a JPEG compressor, and an AES decryption algorithm. We do not plan on reporting on their speedup on an FPGA for the sake of speedup, but rather to quantitatively and intuitively show exactly where the speedup comes from when applying well-known FPGA optimizations. Our goal is to provide a baseline and quantitative intuition as to where the speedups FPGAs are known to achieve come from, and a list of practical recipes for attacking the problem of parallelizing an application. The techniques and tools should be valuable with the advent of a more parallel-computing aware generation of researchers and engineers.

The rest of this paper is organized as follows. Section 2 discusses the study methodology. Sections 3, 1, and 4 quantitatively analyze three separate applications and their FPGA implementations. Section 5 concludes.

2. Study Methodology

The goal of this study is to identify exactly why FPGA implementations are so much faster than their software counterparts. For this reason, we chose three applications that are particularly amenable to FPGA speedup. The examples include an N-body simulation, a JPEG compressor, and an AES decryption unit. While each example comes from a different application domain, each one can be characterized as a data-driven, computationally intensive application, enabling efficient FPGA implementations that are suitable and convenient for such an analysis.

We first run the software on a base MIPS architecture based on the eMIPS extensible processor [16]. Each example is run on a highly configurable simulator [11] that features real time hardware and software integrations, and allows the integration of a detailed SRAM and DDR-2 memory model for streaming data to the FPGA. The base MIPS platform (***BASE***) accesses memory through an SRAM interface, requiring five cycles to fetch both instructions and data. The base architecture is not pipelined, and has no instruction or data cache. We used this particular MIPS platform because the platform was simple to analyze, and is

characteristic of a platform often used for real-time embedded system applications. To make our study more representative of the general-purpose processor class, we clocked the ***BASE*** at a nominal clock speed of 2 GHz.

To better understand and more intuitively depict why FPGAs attain a much higher performance over their software counterparts, we investigate a spectrum of *idealized* optimizations which augment the base MIPS processor for better performance, shown in Figure 1. While not entirely realistic, the optimizations do give some insight as to why FPGAs are able to attain orders of magnitude speedups on certain applications.

	Key	Implementation
SW	BASE	Base MIPS Platform
	S1	BASE + Superscalar Commit*2
	S2	BASE + Perfect Instruction Cache
	S3	BASE + No Instruction Fetch
	S4	BASE+ No I-fetch+ Perfect Data Cache
	S5	BASE+ Superscalar*4 & No I-fetch
	S6	BASE + Perfect Pipeline
FPGA	C1	FPGA data path, No Unrolling
	C2	C1, Loop Unrolled Once
	C3	C1, Loop Unrolled Twice
	C4	C1, Loop Unrolled Four Times
	C5	C1, Pipeline Memory and Computation
	C6	C2, Pipeline Memory and Computation
	C7	C3, Pipeline Memory and Computation
	C8	C4, Pipeline Memory and Computation
	C9	C1, Fully Pipelined
	C10	C2, Fully Pipelined
	C11	C3, Fully Pipelined
	C12	C4, Fully Pipelined

Figure 1: Analyzed Implementations. The key will serve as a reference for subsequent discussion.

Each optimization represents an aspect of execution that FPGAs accomplish very well. The superscalar optimization (***S1***, ***S5***) allows the processor to execute *N* instructions in a single cycle, modeling the ability of an FPGA to exploit instruction level parallelism. We model a perfect instruction cache (***S2***) as an intermediate step in showing the impact of eliminating the execution time required for fetching instructions. Similarly, we model a platform that has no instruction fetch (***S3, S4, and S5***) to more closely model the FPGAs mode of execution, where there is no concept of an instruction fetch. We model a perfect data cache (***S4***) to simulate that an FPGA will probably be reading a large contiguous memory section and streaming data as fast as the memory interface allows. Finally, we model a perfect pipeline (***S6***), allowing the ***BASE*** to complete one instruction per cycle, regardless of actual instruction dependencies or hazards. The perfect pipeline closely models the ability of FPGA applications to complete one data element per cycle, a hallmark of efficient FPGA design for dataflow applications.

We compare the software implementations of each example to the FPGA implementations, also shown in Figure 1. For each FPGA implementation, we apply varying levels of well-known FPGA optimizations [6], including loop unrolling (***C1-C12***), pipelining (***C5-C12***), and streaming constructs (***C9-C12***). Where applicable, we manually implemented each of the data paths for the examples in question and synthesized them using Xilinx ISE 9.2 targeting the Virtex4 Ml401 chip [20]. We simulated the circuits using Giano and ModelSim. For the JPEG compression circuit, we utilized extensive data published by [1].

Since the goal of this work was not to report speedups for any given circuit, but rather explain how and why those speedups were attained, we observe that the execution time of each implementation can be broken down into four main elements:

1. Instruction Fetch
2. Support/Control Instructions
3. Direct Computation Instructions
4. Memory Instructions

The first element is the instruction fetch, which accounts for the time required to fetch instructions from memory. The support/control instructions, first noted by [12], are the instructions that do not directly contribute the real computation, but rather update loop counters, branching code, updating the stack pointer, etc. The direct computations account for all the instructions directly responsible for computing the desired results. Finally, the last category is the time taken to execute memory instructions, including loads and stores. Separating and correlating these four elements among the different implementations leads to a more specific breakdown of how the FPGA is able to attain often dramatic speedups.

3. N-body Simulation

One class of applications that are particularly amenable to FPGA speedup are N-body applications. As a broad definition, N-body applications compute the forces and movements for a (often very large) set of interacting particles/bodies using classical mechanics models. N-body applications usually work on a large set of data, and are computationally intensive. For each element/body in the data set, the N-body algorithm computes the force contributed by each of the other elements/bodies in the data set. The algorithm executes one time step, and then updates each body's position according to the bodies' velocity and the forces previously computed. The naïve particle-particle algorithm runs in $O(n^2)$, although more advanced algorithms can compute a time step in $O(n\log(n))$, using approximation for bodies beyond a certain threshold.

Our particular implementation of N-body operated on 500 distinct bodies, but the analysis would have been similar for both larger and smaller data sets. We first ran the resulting N-body application on a number of software platforms in Figure 1. The distributions are shown in Figure 2. Each column reports the execution time as a percentage of the base platform (***BASE***).

We broke down the execution time for each application run into four separate elements: instruction fetch cycles, direct computation cycles, memory cycles, and support instruction cycles. The base platform (***BASE***) spends 87% of its time fetching instructions, and another 9% accessing data memory. The last 4% of the ***BASE***'s execution time is split between direct computation instructions and support instructions. As shown in Figure 3, the ***BASE*** requires 900 cycles to compute the force for one element in the space. A 2-way superscalar platform (***S1***) is not much better compared to the ***BASE***. Since instruction fetches dominate the execution time, only 13% of the entire time is amenable to speedup via a superscalar platform. ***S1*** did achieve a 7% speedup compared to the base platform. When we supplement the ***BASE*** with a perfect instruction cache (***S2***), we

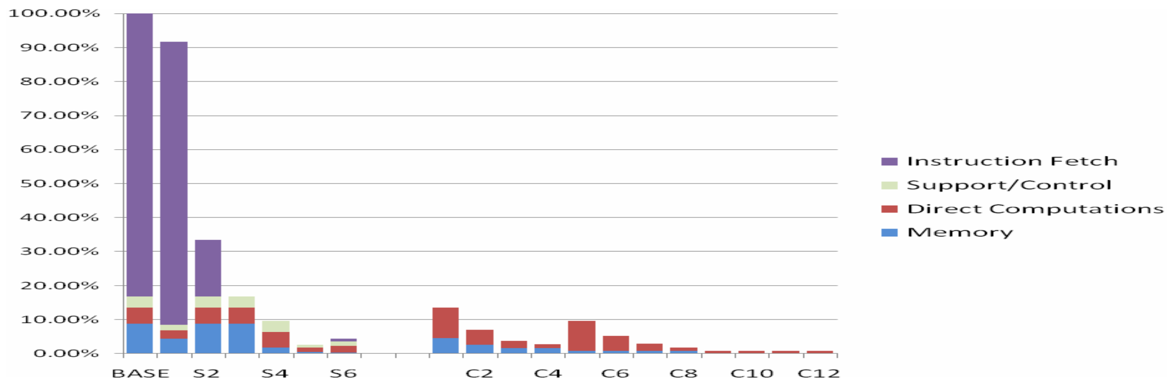


Figure 2: N-body running on a number of different software platforms and with varying levels of FPGA optimizations.

begin to see a more balanced distribution and larger speedups. ***S2*** only spends 57% of its time fetching instructions, since fetching only takes one cycle. Data memory accesses now account for 29% of the execution time, direct computations take 8%, and support/control instructions account for 6%. The fourth platform models an idealization of a processor that does not have an instruction fetch (***S3***). While not realistic, the modeling technique does give some insight as to where speedup from an FPGA implementation comes from since an FPGA has no concept of an instruction fetch. ***S3*** runs 7.5X faster than the ***BASE***. Memory accesses dominate, accounting for 69% of the execution time. The computations that relate directly to the force computation take 18% of the time, and the support/control instructions account for the final 13% of the execution time. We can improve the execution time by supplementing the previous ideal platform with a perfect data cache (***S4***), allowing our platform to effectively read data from memory in one cycle. The resulting platform attains ~17X speedup. Direct computations now account for 41% of the execution time, support/control instructions take 28% of the time, and memory accesses take 31%. The resulting execution profile is beginning to resemble an FPGA implementation. A perfect four-way superscalar platform (***S5***) achieves 62X speedup over the ***BASE*** because it is able to execute four computations in parallel. This results in 39% of the execution time coming from direct computation execution, 27% of the execution from support/control instructions, and memory accesses occupying 35% of the execution time. Finally, we run the N-body application on a model of a perfect pipeline (***S6***). ***S6*** runs ~44X faster than the base platform. Memory computations take 8% of the execution time; direct computations take 42% of the time; support/control instructions take 30%, and the instruction fetch takes 20% of the execution time.

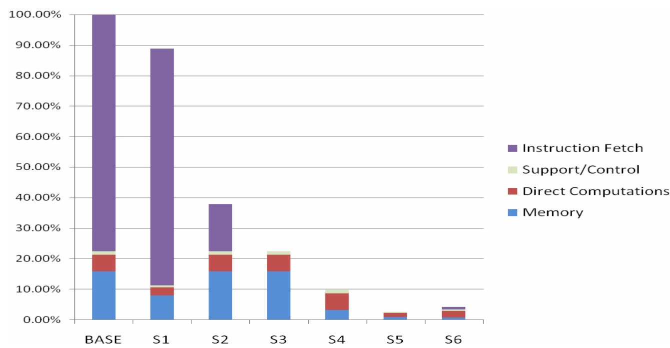
By executing the N-body application on a number of different idealized platforms, we gain insight into how the FPGA gains performance for the equivalent software platform. The superscalar models showed they can attain performance gains by parallelizing at the instruction level. Platforms without an instruction fetch immediately showed FPGAs gain performance because they have no concept of an instruction fetch. Similarly, FPGA implementations gain speedup by eliminating or hiding the support/control instructions, including branch instructions, updating loops, and moving data because of a lack of registers. Finally, FPGAs can gain by pipelining operations.

We now analyze several FPGA implementations for the N-body inner loop, which calculates the force between two bodies in the simulation. The right hand side of Figure 2 shows the distributions for FPGA circuit implementations with various levels of loop unrolling and pipelining. Figure 3 shows some of the FPGA implementations and how many total cycles they require to compute one force calculation. All of the circuit implementations were able to eliminate the cost of fetching instructions *and* the cost of the control instructions. The N-body circuit allows controlling software to push data as fast as it can through the N-body circuit, eliminating the need for any control flow within the circuit. The leftmost bar (***C1***) in Figure 2 shows a single FPGA data path implementing the N-body force calculation. The circuit fetches data from DDR memory, and accounts for 34% of the circuit's execution time. Therefore, 66% of the time is spent calculating the force. Figure 3 shows that an FPGA data path is able to complete one force calculation in 94 cycles. The computational latency of the data path is only 61 cycles, but accessing the DDR-2 memory requires another 33 cycles. Still, the FPGA data path is ~7X more efficient than the base MIPS processor. Part of the FPGAs efficiency is due to the elimination of the instruction fetch and control cycles (which account for 89% of the base platform's time), but the FPGA is also able to schedule multiple instructions in parallel, which accounts for ~2X efficiency after the instruction fetch and control instructions are eliminated. Factoring in a slower clock speed of 250 MHz (~8X slower than the MIPS platform), the FPGA data path is able to achieve ~7X compared to the ***BASE***.

	Implementation	Cycles per force element
SW	Base MIPS (BASE)	900
	Perfect I-Cache (S2)	408
	Superscalar*4 & No I-fetch (S5)	60
	Perfect Pipeline (S6)	123
FPGA	FPGA data path (C1)	94
	Loop Unroll 1 (C2)	49
	Full Pipeline (C9)	1

Figure 3: N-body. Cycles to compute one force element.

The FPGA implementation can be further improved by unrolling the inner loop a number of times, exhibited by the next few bars (***C2-C4***) in Figure 2. Loop unrolling once, twice, and four times results in speedups of 13X, 25X, and 35X respectively, compared to the ***BASE***. As shown in Figure 3, loop unrolling once reduces the number of clock cycles to compute one force element to 49 cycles. The latency of the data path is still the same, but the FPGA can compute two forces at the same time. The distributions in Figure 2 show us that the FPGA increases the amount of computations performed in one time step by loop unrolling. The percentage of time spent on computation reduces from 66% with the base FPGA data path to 63% when



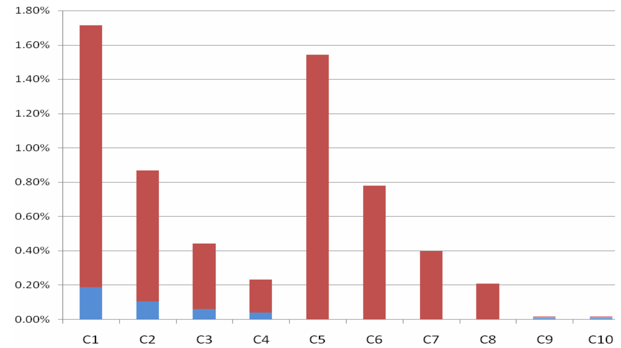


Figure 6: (a) AES running on different software platforms. Eliminating instruction fetch speeds up AES by ~5X. (b) Different FPGA implementations.

we unroll the force loop once, 58% when the loop is unrolled twice and 40% when the loop is unrolled four times. Unrolling the calculation shifts the bottleneck from the calculation to the data fetch. In ***C4***, the memory accesses account for 60% of the execution time.

We can improve upon the naïve versions of loop unrolling by pipelining the accesses to memory while the previous calculation is taking place such that the data is ready for the next execution. Pipelining the memory accesses (***C5-C8***) for each of the four loop unrolling examples results in a ~1.2-1.4X speedup compared to the original loop unrolling implementations (***C1-C4***). The majority of the execution time is now concentrated on the computation, 92% for ***C5*** and 60% for ***C8.***

We fully pipeline the N-body calculation into a 61 stage pipeline (***C9***), allowing the circuit to achieve speedups of ~127X compared to the ***BASE***. A full pipeline can complete one force calculation per cycle, 900X more efficient than the ***BASE***. Extra time must be spent paging and refreshing the DDR-2 memory. Those extra cycles actually cause the pipeline to stall several times during execution. Unrolling the pipelined circuit (***C10, C11, and C12***) does not offer any additional opportunities for speedup since the 32-bit bus is completely occupied supplying just one pipeline with data. The N-body circuit could have certainly also performed better with a more custom memory interface to allow for larger bandwidth, but the current analysis was fixed to a development board that only had a 32-bit interface to memory.

4. Other Examples

We also quantify several other examples known to achieve much FPGA speedup, including a JPEG compressor[4][15], and an AES encryption unit[5]. The quantitative analysis of those can be found in [17].

5. Conclusion

We analyzed how FPGAs attain so much speedup over their sequential software counterparts. We presented an extensive quantitative analysis a baseline N-body simulation By first showing how each application ran on both a baseline MIPS processor and several ideal optimized software architectures, *and* breaking up the execution time into several elements, we created a spectrum that visually and intuitively showed how an FPGA gathers speedup. The superscalar models showed how FPGAs can exploit instruction level parallelism. Platforms without an instruction fetch closely modeled the fact that FPGAs have no instruction fetch phase. We then compared those ideal optimized platform executions to several FPGA implementations, varying levels of loop unrolling and pipelining. We showed how FPGA implementations lose cycles by eliminating the software instruction fetch, hiding the control instructions, executing multiple instructions in parallel, and pipelining those instructions. Our goal is to provide researchers and engineers a quantitative intuition how to attack the problem of parallelizing certain applications, both for software and FPGA platforms.

References

[1] AGOSTINI, L., B. SERGIO., AND SILVA, I. High Throughput Architecture of JPEG Compressor for Color Images Targeting FPGAs. ICECS 2006.

[2] ALTERA CORPORATION. http://www.altera.com

[3] BEECKLER, J. S. AND GROSS, W. J. 2005. FPGA Particle Graphics Hardware. FCCM 2005

[4] BHASKARAN, V. AND KONSTANTINIDES, K. Image and Video Compression Standards Algorithms and Architectures Second Edition. Kluwer Academic Publishers, USA, 1999.

[5] DAEMEN, J. AND RIJMEN, V. The Design of Rijndael. AES- The Advanced Encryption Standard. Springer 2002.

[6] DE MICHELI, G. Synthesis and Optimization of Digital Circuits. McGraw Hill Higher Education. 1994

[7] DEHOHN, A. The Density Advantage of Configurable Computing. IEEE Computer, vol. 33, No.4, April 2000

[8] ELBIRT, A.J., YIP W., CHETWYND, B. AND PAAR C. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm. AES Candidate Conference. 2000.

[9] FCCM. Field-Programmable Custom Computing Machines Conference. http://www.fccm.org

[10] FPGA. International Symposium on Field-Programmable Gate Arrays. http://www.ece.wisc.edu/~kati/isfpga/

[11] FORIN, A. , NEEKZAD, B. ,AND LYNCH, N. Giano: The Two-Headed System Simulator. Microsoft Technical Report?

[12] GUO, Z., NAJJAR, W., VAHID, F., AND VISSERS, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. FPGA '04

[13] LIENHART, G., KUGEL, A. AND MANNER, R. Using Floating Point Arithmetic on FPGAs to Accelerate Scientific N-body Simulations. FCCM 2002.

[14] MEYER, K. AND HALL, G. Introduction to Hamiltonian Dynamical Systems and the N-body Problem. Springer Publishing 2002.

[15] PENNEBAKER, W. AND MITCHELL, J. JPEG Still Image Data Compression Standard. Van Nostrand, 1992.

[16] PITTMAN, R.N, LYNCH, N., AND FORIN, A. eMIPS, a Dynamically Extensible Processor. Microsoft Technical Report. October 2006.

[17] SIROWY, S. AND FORIN, A. Wheres the Beef? Why FPGAs Are So Fast. Technical Report MSR-TR-2008-130

[18] TSOI, K. H., LEE, K. H., AND LEONG, P. H. 2002. A Massively Parallel RC4 Key Search Engine. FCCM

[19] WHITTON, K., HU, X. S., YI, C. X., AND CHEN, D. Z. 2006. An FPGA Solution for Radiation Dose Calculation.FCCM

[20] XILINX, INC. http://www.xilinx.com