

THE EFFECTS OF HARDWARE ACCELERATION ON POWER USAGE IN BASIC
HIGH-PERFORMANCE COMPUTING

by

CHRISTOPHER AMSLER

B.S., Kansas State University, 2010

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2012

Approved by:

Major Professor
Dr. Dwight Day

Copyright

CHRISTOPHER AMSLER

2012

Abstract

Power consumption has become a large concern in many systems including portable electronics and supercomputers. Creating efficient hardware that can do more computation with less power is highly desirable. This project proposes a possible avenue to complete this goal by hardware accelerating a conjugate gradient solve using a Field Programmable Gate Array (FPGA). This method uses three basic operations frequently: dot product, weighted vector addition, and sparse matrix vector multiply. Each operation was accelerated on the FPGA. A power monitor was also implemented to measure the power consumption of the FPGA during each operation with several different implementations. Results showed that a decrease in time can be achieved with the dot product being hardware accelerated in relation to a software only approach. However, the more memory intensive operations were slowed using the current architecture for hardware acceleration.

Table of Contents

List of Figures	vi
List of Equations	vii
List of Tables	viii
Acknowledgements	ix
Chapter 1 - Introduction.....	1
FPGA	2
Overview.....	3
Previous Work	4
Fine-Grained Component-Level Power Measurement	4
FPGA-based Hardware Acceleration.....	5
Chapter 2 - Power Monitor	6
Microsemi Power Measurement	7
Current Work	9
System Monitor.....	10
Chapter 3 - Benchmark	13
LINPACK	13
Mantevo	14
HPCCG	14
Matrix Formats	15
Chapter 4 - Hardware Acceleration	17
Architecture	17
Dot Product.....	20

WAXPBY	23
SPARSEMV	24
Chapter 5 - Results.....	25
Setup	26
Performance Metric	26
Initial Results	27
Tuning.....	29
Accumulator Stages	30
HardwareStep Size.....	32
Chapter 6 - Conclusion	34
Future Work	35
Chapter 7 - Bibliography	37

List of Figures

Figure 1 - Power of components during 401.bzip2 benchmark.....	5
Figure 2 - Speedup achieved Figure 3 - Power reduction.....	6
Figure 4 - IGLOO power consumption.....	8
Figure 5 - Power Measurement.....	10
Figure 6 - System Monitor Block Diagram	11
Figure 7 - ADC Transfer Function.....	12
Figure 8 - Embedded Processor Block in Virtex-5 FPGA.....	18
Figure 9 - Block Diagram of Accelerator	19
Figure 10 - Cascaded Dot Product.....	22
Figure 11 - Dot Product ChipScope Timing.....	22
Figure 12 - Hardware Implementation of WAXPBY	23
Figure 13 - ChipScope Timing of WAXPBY.....	24
Figure 14 - ChipScope Timings for Matrix Vector Dot Products	25
Figure 15 - Hardware vs. Software Time/iteration.....	28
Figure 16 - Hardware vs. Software MFLOPS/Watt.....	29
Figure 17 - Accumulator Stages Time/Iteration	30
Figure 18 - Accumulator Stages MFLOPS/Watt.....	31
Figure 19 - HardwareStep Comparison Time/Iteration	32
Figure 20 - HardwareStep Comparison MFLOPS/Watt.....	33

List of Equations

Equation 1 - Conjugate Solve Method.....	15
Equation 2 - Example Sparse Matrix	16
Equation 3 - Representation of A in DIA Format.....	16
Equation 4 - Representation of A in COO Format	17
Equation 5 - Representation of A in CSR Format	17
Equation 6 - Dot Product Equation.....	21
Equation 7 - Dot Product pseudo code	21
Equation 8 - WAXPBY Equation.....	23
Equation 9 - Cross Product Equation.....	24

List of Tables

Table 1- Accelerator Types.....	6
Table 2- IGLOO Specifications [14]	7
Table 3 - System Monitor Signals	11
Table 4 - UDI Instructions	19
Table 5 - Register Map for Accelerator	20
Table 6 - Floating Operations	27

Acknowledgements

I would like to acknowledge Dr. Dwight Day for his assistance in completing this project. Additionally, his previous work created a starting point for this research and I appreciate all of his guidance on this topic. I would also like to thank Sandia National Laboratories for their financial support and for offering me the opportunity to gain knowledge while with Douglas Doerfler and Carter Edwards. I would also like to thank my committee for their involvement and guidance in this research, Dr. Don Gruenbacher and Dr. David Soldan.

Chapter 1 - Introduction

As quickly as new technologies are developing, a major concern is keeping the power consumption on these devices manageable. According to Dally [1], reducing power consumption is one of the major challenges of exascale computing, which is defined as executing a quintillion floating point operations per second, exaflops. In 2007, the United States Defense Advanced Research Projects Agency (DARPA) asked a team of engineers to determine: what kind of technologies would be needed to build supercomputers capable of computing exaflops. In addition to making a computer capable of exaflops, DARPA also limited the power consumption of the computer to 20MW and it must fit into at most 500 conventional server racks [2].

Keeping a power budget of 20MW for exascale computing is an ambitious task. In November 2011, the fastest supercomputer, K computer, could achieve 10 petaflops using 12MW [3], or 833 MFLOP/Watt. This clearly shows that having exascale computing with a power limit of only 20MW, or 50 GFLOP/Watt, is extremely difficult given current technologies. For exascale computing to be possible the energy consumption must dramatically be reduced. During the study by Kogge [2], it was found that the energy used per flop was around 70 pJ, excluding memory accesses. This energy needs to be reduced to around 5 to 10 pJ. For this reduction to occur, dramatic changes must arise in computer architectures. A possible solution to help reduce power consumption could be using hardware accelerators. Hardware accelerators take a normally complicated and time-consuming process to calculate on a typical processor and offloads that process to dedicated hardware capable of performing that single process. The dedicated hardware varies from a dedicated processor such as a floating point unit to specialized hardware that computes an FIR (Finite Impulse Response) filter.

One of the more popular hardware accelerators is the Graphics Processing Unit (GPU). These devices have hundreds of small processors that work well with massively parallel applications. These small processors are ideal for computing graphics, because each pixel can be calculated somewhat independently. GPUs also allow highly parallel applications to run much more efficiently when compared with a typical multi-core processor. Because of this many High Performance Computing (HPC) centers have implemented these devices in supercomputers. As of November 2011, three of the top 10 supercomputers implemented systems with GPUs [3]. A NVIDIA Tesla C2075 can perform double precision 515 GFLOPS and only requires 225W thermal design power. [4] This out-performs an Intel Core i7, which only can perform 187 GFLOPS, [5] using a maximum thermal design power of 130W [6]. GPUs are now being added into supercomputer system because of their performance.

FPGA

Another option is creating specialized hardware to compute a problem. For this, a Field Programmable Gate Array (FPGA) can be used. FPGAs contain tens to hundreds of thousands of logic cells. Logic cells normally contain flip-flops and lookup tables (LUTs), along with some other logic, such as multiplexers and arithmetic circuitry, but these vary between FPGA manufacturers. With such a large number of logic cells many different problems can be accomplished, such as digital signal processing (DSP), cryptography, and the creation of glue logic between sensors or devices. These devices also contain several built-in hardware components, such as dedicated memory blocks, integrated Phase Lock Loops (PLL), DSP blocks and more. These dedicated blocks help improve performance and reduce the number of gates needed to build a design. Creating the hardware on FPGAs is done through hardware description language (HDL). Verilog and VHDL are the two most common languages used.

FPGAs can be physically built in several ways, but Random Access Memory (RAM) and flash based memory are the most common. RAM blocks can be used to connect and route logic together. Because RAM is volatile memory, power to the FPGA must be kept on for designs to work. However, RAM based FPGAs can normally run at higher speeds. Altera and Xilinx are two of the larger RAM-based FPGAs manufactures. Flash-based devices have the beneficial ability to keep configurations even after power is off, but they also have an added benefit of keeping the static power consumption down to a minimum because of the flash technology. Static power is defined as the power needed to keep the device powered-up but not performing any operations [7]. Dynamic power is the power consumed during operation. Flash-based FPGAs have a much higher dynamic power than static power. However, in RAM-based FPGAs this is reversed and static power is higher than dynamic power. This is because the RAM must be refreshed frequently.

FPGAs are inherently parallel in nature. The lack of system structure allows for this parallelism, unlike in a single processor where data must flow from instruction decode to execute. This makes FPGAs great for parallel computing applications. Applications like DSP, and cryptography are great examples of parallel computing. Because they manage different sensor and device signals simultaneously, much better than a standard processor, which would require interrupt handling.

Overview

The goal of this research is to use an FPGA to hardware accelerate a HPC application and see the power consumption used during the process. The FPGA chosen was a Xilinx Virtex-5 XC5VFX130T, which is on the ML510 development board [8]. This board was chosen because

it is equipped with the equipment needed to complete this project, such as serial communications ports and power monitoring capabilities.

The main objectives are as follows:

1. Create a power measurement system that can be used during program execution.
2. Implement hardware accelerator created by Dr. Dwight Day for the HPCCG application on updated Xilinx device.
3. Use a performance metric to compare hardware and software implementations.

Previous Work

Fine-Grained Component-Level Power Measurement

In [9], the approach was to monitor the most important devices in a personal computer (PC), the hard drive, RAM memory, network card, video card, and CPU. The power from the hard drive was monitored through the SATA wires. To measure the components plugged into specific slots in the motherboard, memory network card, and video card, a wrapper card was made to measure power, which is transparent to the computer system. The CPU power was monitored through the voltage regulators. Although this is not the most ideal way to monitor CPU power, it is the most economical because creating a wrapper card for the CPU is impractical for physical and monetary reasons.

To ensure that the measurements recorded were valid, several components were put through strenuous tests. The memory was tested using the STREAM benchmark [10]. STREAM measures "sustainable memory bandwidth and computation rate for a vector kernel." [10] Running this benchmark should make the memory run at maximum power dissipation because it

is very memory intensive. STREAM yielded a maximum power dissipation of 2.029W, the maximum power of the memory is 2.016W, which validates the measurements.

The CPU benchmark chosen was Prime95. This program finds new Mersenne prime numbers [11]. Prime95 has a setting labeled "Torture Test," that can make the CPU dissipate the most power. The power for this test was at 64.272W, while the maximum thermal design power is 65W, again validating the monitoring setup for the CPU.

After validation of measurements was completed, the CPU, memory and disk power consumption were recorded while running 401.bzip2, which is part of the SPEC CPU2006 benchmark [12]. This benchmark compresses and decompresses different types of files formats. The results from the experiment are shown below in Figure 1. This experiment showed that power consumption is dominated by the CPU.

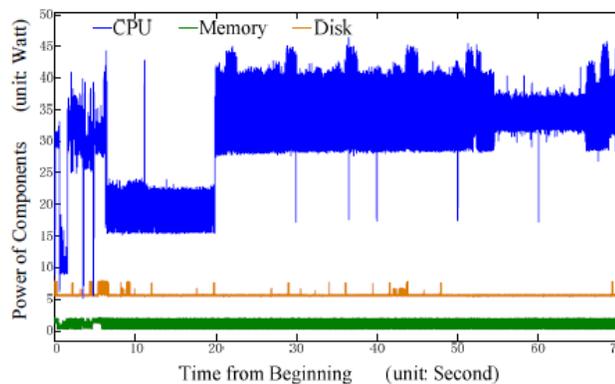


Figure 1 - Power of components during 401.bzip2 benchmark

FPGA-based Hardware Acceleration

Possa [13] addressed the tradeoffs between power and performance in embedded systems. They used a FPGA to offload the task of implementing a 15th order FIR filter function. The FPGA of choice was an Altera EP3C120, which is on the Cyclone III development board.

They implemented several different acceleration techniques, located in Table 1. Tests were done implementing different processor versions, economic (/e), standard (/s), and fast (/f).

FIR Version	Description
Software	Nios II Softprocessor, only
Avalon-MM (Memory Mapped)	Data between processor and accelerator communicated through Altera Avalon interface
Avalon-MM with FIFOs	Same architecture as above, except FIFOs are included between interface
Tightly Couple Memory (TCM)	Low latency memory independent from Avalon-MM
Custom Instructions (CI)	Custom instructions that communicate directly between processor and accelerator
C2H Compiler (C-to-Hardware)	Altera automated tool that can create HDL code directly from ANSI C code

Table 1- Accelerator Types

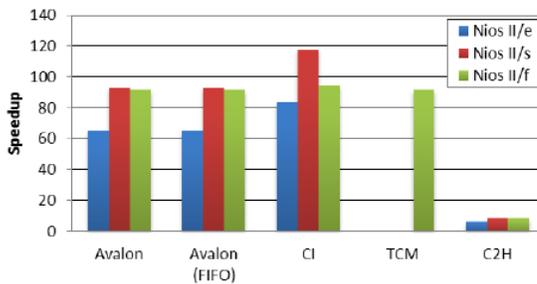


Figure 2 - Speedup achieved

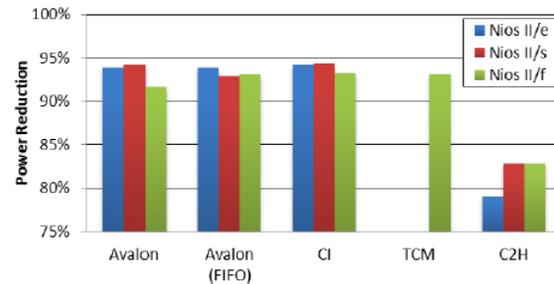


Figure 3 - Power reduction

As shown above, significant speed ups can be accomplished by offloading the FIR filter to the FPGA. Along with the performance speed up, power consumption was also reduced.

Chapter 2 - Power Monitor

Because of increased interest in power consumption many large systems now are created with power consumption in mind. This requires the power of the system to be monitored in some

way. There are a variety of ways this can be done, either through simulation or by accurately monitoring the system in real time. Simulating power consumption of a system can give a quick insight to a system’s performance. However, sometimes a more precise measurement is required because simulation cannot take into account all the variables that occur in actual power performance. This involves introducing power sensors into the system, which can in some cases be costly and even unfeasible.

Many FPGA systems are now being made with power monitoring sensors on the development boards or even on the FPGA itself, making them a good candidate for monitoring power consumption. The way the monitoring is done can vary, some cases have only chip temperature information available while others have current and voltage that can be calculated.

Microsemi Power Measurement

A study was done involving the measurement of power consumption in an FPGA. The device chosen was a Microsemi IGLOO nano AGLN250Z, which comes on the IGLOO nano Starter Kit development board. The IGLOO nano is built using flash memory technology and is designed for low power solutions. Full specifications are located in Table 2. This development board has the ability to measure current of the voltage rails on the board directly through header pins. The rail of interest was the core voltage rail V_{CC} .

Description	Value
System Gates	250,000
VersaTiles	6,144
Flash*Freeze Mode Power Usage	24 μ W
RAM kbits	36
Maximum User I/Os	68

Table 2- IGLOO Specifications [14]

The goal of this project was to categorize power consumption in terms of size and frequency. Size represents the percent of the FPGA taken up by the design used and frequency is the speed at which the FPGA is running. A variety of Microsemi Intellectual Protocols (IP) were chosen: UART, ABC processor, SRAM, and PWM. These IPs were chosen because they are common to many FPGA designs. Each IP has several options that can be chosen: varying size or SRAM, baud rate of UART, or number of outputs for the PWM. Varying the frequency of the FPGA was done through the internal PLL.

Measurement of the FPGA was then taken using a Digital Multi-Meter (DMM), HP 34401A, which can measure μ As. Then power was calculated using voltage times current where the voltage was equal to the rail voltage of the FPGA, 1.2V. The results are shown in Figure 4.

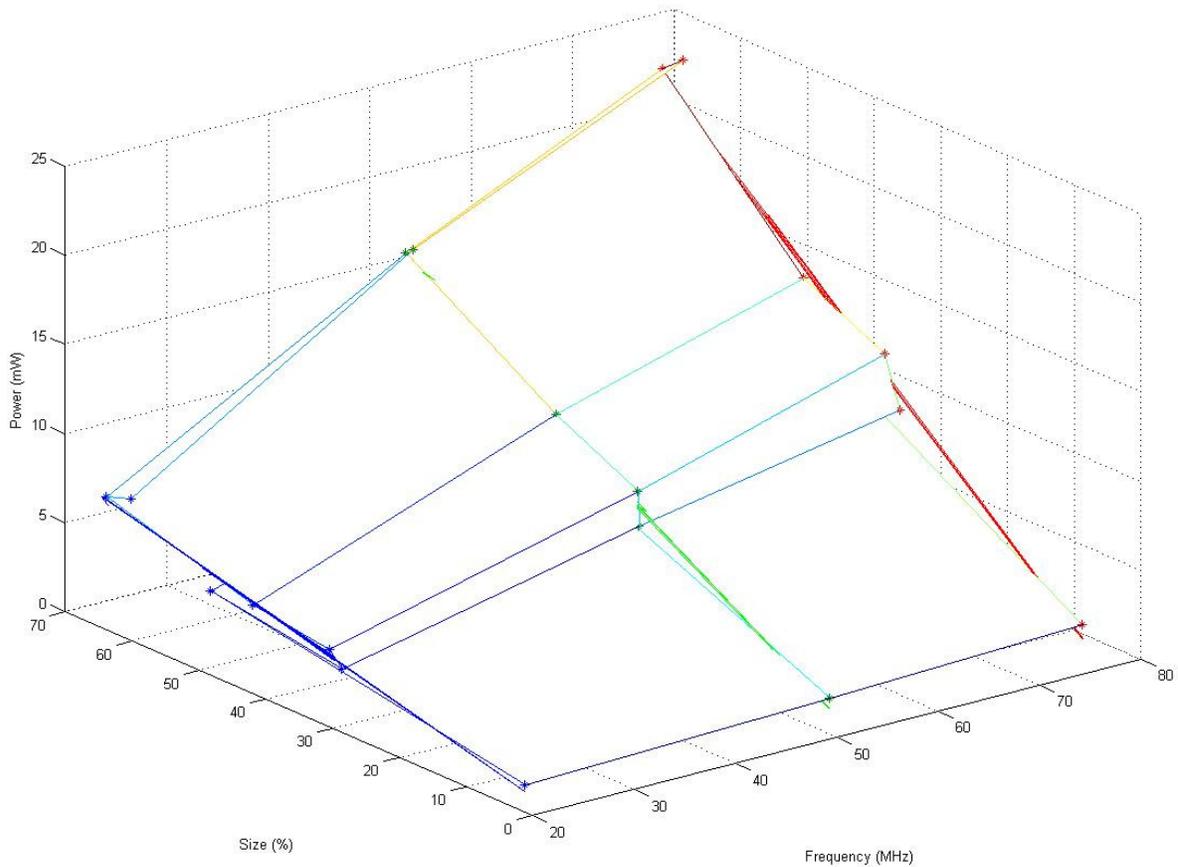


Figure 4 - IGLOO power consumption

This plot shows that power is dependent on both size and frequency. It may look as though power is more dependent on frequency than size; however, these variables are somewhat dependent upon each other. The more of the device that is utilized, the more the clock will need to be routed around the chip. However, when a small amount of the device is used the change in frequency causes less of a adjustment in the power consumption.

Current Work

For our case, measuring the power that the FPGA chip consumes can be done quite easily using the Xilinx System Monitor, which is implemented in every Virtex-5 device. The ML510 development board implements this monitor according to Xilinx System Monitor specifications, according to [15]. The System Monitor allows the user to measure several voltage rails, temperature readings of the chip, and external signals. The only signals of interest in our experiment are the dedicated analog input pins, V_P and V_N . On the ML510 board, these signals can be connected directly to a current sense resistor on the board via jumpers. The sense resistor is connected directly between the voltage regulator and the FPGA. Connecting V_P and V_N to the output of the sense resistor returns the voltage drop across the resistor. (Figure 5) This voltage can then be used to find the current the FPGA is drawing from Ohms law. Finally, power can be calculated using current and voltage relationships.

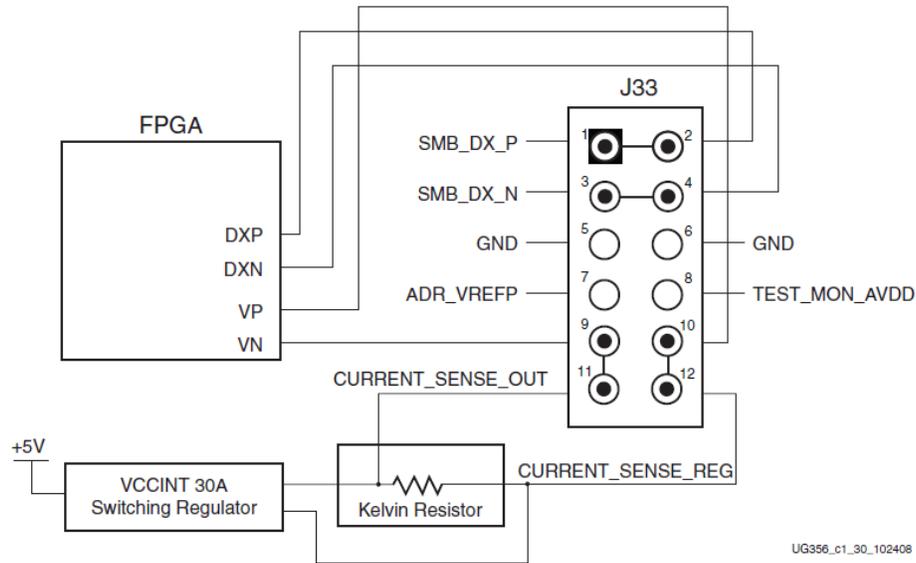


Figure 5 - Power Measurement

System Monitor

On Virtex 5 FPGAs, the system monitor is located in the center of the die. The monitor uses a 10-bit, 200kSPS(kilo-samples per second) Analog-to-Digital Converter (ADC). The block diagram for the monitor is shown on Figure 6. Setting up the system monitor requires writing to the control registers, through the dynamic reconfiguration port (DRP). Getting ADC readings out is done by reading the status registers. The system monitor can be set to measure internal supply voltages, temperature sensor, or auxiliary inputs. The DRP also allows for the monitor to be changed during run time through custom logic or using Xilinx ChipScope Pro. In fact, ChipScope has the ability to set all signals to allow for reading of all sensors without manually setting them. ChipScope also allows logging of all data that is being measured. Although using ChipScope allows for easy measurement of temperature and internal supply sensors, it only allows for measurement periods of one second, which may not be desirable. For our study, a different approach was taken because more accurate measurements would yield better results.

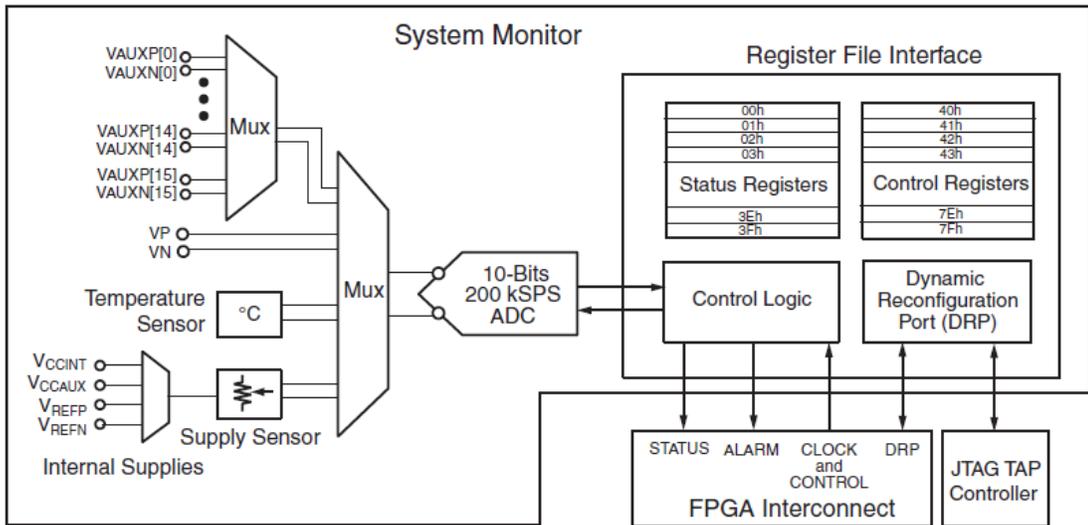


Figure 6 - System Monitor Block Diagram

Setting up the system monitor manually can be done in a variety of ways, such as using a built in IP that is governed by the processor, or using Xilinx CORE Generator (CoreGen). The creation of the System Monitor was done through CoreGen to ensure power monitoring affected the system minimally. Setup requires instantiating the monitor and it's required control registers, located in Table 3. This allows monitoring of the system without intense involvement with the processor. All that is required from the processor is a start and stop signal.

Signal Name	Value	Description
INIT_40	0x0003	Enable V_P and V_N
INIT_41	0x20F0	Continuous sequence, calibrate all sensors
INIT_42	0x0500	Divide input clock to 10 MHz
INIT_48	0x0F01	Select ADC Channel V_P and V_N , enable calibration

Table 3 - System Monitor Signals

After the monitor is setup, acquiring and interpreting data is the next step. The system monitor has six dedicated ADC pins, two of which are V_P and V_N . The transfer function for their operations are located on Figure 7. However, reading those outputs from the system monitor returns a 16-bit number. The 10 Most Significant Bits (MSBs) represent the output from the ADC, which then represent the figure below.

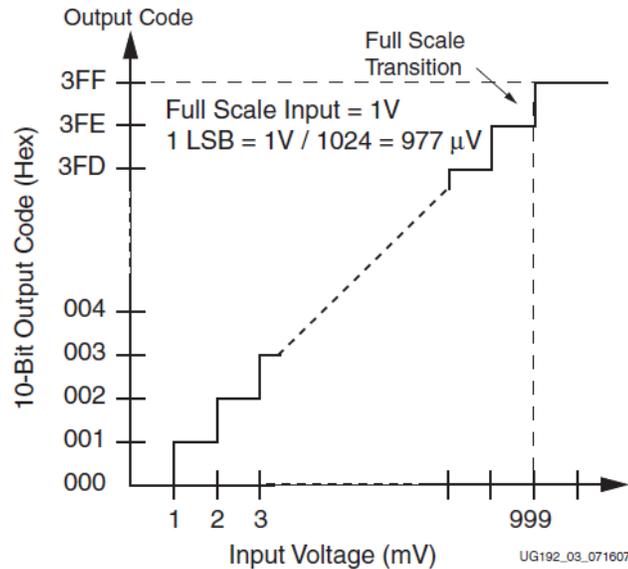


Figure 7 - ADC Transfer Function

Calculating power consumed is done by transforming the ADC measurement into a voltage, the drop across the resistor V_R . This is done using the transfer function above, divide 10 MSBs of ADC measurement by 1024. To calculate the power the FPGA chip is consuming, divide V_R by the sense resistor value, which is $2.2\text{m}\Omega$ and multiply by the rail voltage for the FPGA, $V_{CC_{int}}$, 1V nominal.

With the system monitor initialized, the next step is to get the data back to the PC. A separate module was created to ensure minimal interaction with the processor. This module sends the system monitor data to the PC via RS232. A sampling frequency of 10 Hz was chosen

to get accurate power readings. The only extra data sent to the PC is start and stop signals, which signify the start and stop of the power measurement. The measurement of interest is during the computation of the conjugate gradient method.

Chapter 3 - Benchmark

In HPC, large powerful systems are used to solve large and complex problems, such as weather prediction or molecular dynamic simulation. Most of these complex problems require the same set of algorithms to solve problems. This allows benchmarks to be made that characterize system performance in real application. A very popular benchmark is the LINear Algebra PACKage (LINPACK) which measures a system's floating point computing power. Sandia National Laboratories has also created a test suite named Mantevo according to [16]. This has several mini-applications that can be run in variety of different ways including, parallel and serial computation on GPUs and CPUs using commercial off the shelf computers, such as typical personal computers, or even supercomputers.

LINPACK

LINPACK is a collection of routines that can solve linear equations. A benchmark was created later and solves a system of linear equations, $Ax = b$, where A is a dense matrix. One of the standard tests used to be called the LINPACK 1000, which solved a dense matrix with a size of 1000 by 1000 [17]. However, when LINPACK was commonly used in the 1970s and 1980s, supercomputers were very different than those found today. So, the High-Performance LINPACK (HPL) benchmark was created to be used with distributed-memory computers of today.

Mantevo

Mantevo is an application performance project containing mini-applications, mini-drivers, and application proxies. Mini-applications are small programs that contain performance characteristics of larger applications. Mini-drivers are used in conjunction with Trilinos, a Sandia library created to develop algorithms to be used in large-scale engineering problems [18].

Application proxies are applications that can be tuned to copy the performance of a large scale application, which is then used as a proxy to the large application.

There are several mini-applications under the Mantevo project. The mini-app miniFE is the "best approximation to an unstructured implicit finite element or finite volume application, but in 8000 lines or fewer." [16] High Performance Computing for Conjugate Gradient (HPCCG) solves the equation $Ax = b$, where A is a sparse matrix using the conjugate gradient method. The application, phdMesh is a dynamic mesh application that uses parallel computation methods. HPCCG was chosen to be hardware accelerated. With fewer than 800 lines of code, it is a small enough application that accelerating the process should be more tractable on our system.

HPCCG

The HPCCG benchmark creates an $N \times N \times N$ sparse matrix, where N is defined by the user and then solves the $Ax = b$, where A is the sparse matrix, using the Conjugate Gradient (CG) method. The CG method is fairly straightforward as shown below in Equation 1. The A matrix must also be square, symmetric and positive-definite for the CG solve to correctly solve the system. [19]

$$\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$$

$$\alpha_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{p}_i$$

$$\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{A} \mathbf{p}_i$$

$$\beta_{i+1} = \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}$$

$$\mathbf{p}_{i+1} = \mathbf{r}_{i+1} + \beta_{i+1} \mathbf{p}_i$$

Equation 1 - Conjugate Solve Method

The main parts of the CG solve are the dot product, waxpby (weighted vector addition) and sparse matrix vector multiply (sparsemv). For each iteration of the CG solve, two dot products, three waxpby, and one sparsemv need to be computed. They were chosen for hardware acceleration because these are the most computation intensive operations.

Matrix Formats

A sparse matrix contains a large number of zero elements. These matrices appear in many real world applications, which is why HPCCG uses a sparse matrix. Because this matrix has many zeroes storing the data can be done differently than a typical dense matrix to save space. There are several formats that can be chosen, some of which include: diagonal (DIA) format, coordinate (COO) format, and compressed sparse row (CSR), format. DIA is useful when most of the nonzero elements are in the matrix diagonals. Two structures are used to represent the sparse matrix, **data**, a diagonally organized matrix, and **offsets**, array containing list of diagonal indexes. COO is made up of three vectors: **row**, which stores row indices; **col**, which stores column indices; and **data**, which stores values. CSR is similar to COO except that the **row** is compressed to **ptr**, while **col** and **data** stay the same. Instead **ptr**, is compressed to a list of indexes to the nonzero elements. [20] HPCCG uses the CSR format, because it has less of a

memory footprint. Consider the example below, to get a better understanding of each storage scheme.

$$A = \begin{bmatrix} 1.2 & 0 & 3.6 & 0 \\ 0 & 4.2 & 0 & 2.8 \\ 6.0 & 0 & 5.5 & 7.3 \\ 0.9 & 0 & 0 & 8.6 \end{bmatrix}$$

Equation 2 - Example Sparse Matrix

To access a location in the matrix a typical two dimensional call can be done, for example $0.9 = A[3,0]$, using base zero accessing.

$$data = \begin{bmatrix} * & * & 1.2 & * & 3.6 \\ * & * & 4.2 & * & 2.8 \\ * & 6.0 & 5.5 & 7.3 & * \\ 0.9 & * & 8.6 & * & * \end{bmatrix} \quad offsets = [-3 \quad -2 \quad 0 \quad 1 \quad 2]$$

Equation 3 - Representation of A in DIA Format

Asterisks are merely space holders for the **data** matrix because these can be set to any value if needed. The columns of **data** correspond to the diagonals of the sparse matrix. The **offsets** array represent the diagonal offset; zero being the major diagonal; negative numbers stand for sub diagonals; and positive represent super diagonals. As shown above, Equation 3, more space is used to represent the example matrix. However, if the matrix were predominantly banded on the diagonals, a storage benefit would occur. To access data here two structures must be accessed. The value $0.9 = data[3,0]$ and **offsets**[0]. This will represent the actual position in the sparse matrix **A**.

$$\begin{aligned}
\mathbf{row} &= [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3] \\
\mathbf{col} &= [0 \ 2 \ 1 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3] \\
\mathbf{data} &= [1.2 \ 3.6 \ 4.2 \ 2.8 \ 6.0 \ 5.5 \ 7.3 \ 0.9 \ 8.6]
\end{aligned}$$

Equation 4 - Representation of A in COO Format

In COO, both **row** and **col** represent the row and column locations for the corresponding non-zero elements in **data**. This format is beneficial when the matrix is very sparse because then less information needs to be contained in **data**. Here three single dimensional arrays must be accessed. Locating the data 0.9 is done through the non-zero element index, seven in this case.

$$\mathbf{row}[7] = 3, \mathbf{col}[7] = 0, \text{ and } \mathbf{data}[7] = 0.9.$$

$$\begin{aligned}
\mathbf{ptr} &= [0 \ 2 \ 4 \ 7 \ 8] \\
\mathbf{col} &= [0 \ 2 \ 1 \ 3 \ 0 \ 2 \ 3 \ 0 \ 3] \\
\mathbf{data} &= [1.2 \ 3.6 \ 4.2 \ 2.8 \ 6.0 \ 5.5 \ 7.3 \ 0.9 \ 8.6]
\end{aligned}$$

Equation 5 - Representation of A in CSR Format

As shown above, the CSR format is the most compact and efficient representation of the sparse matrix A; however, this is not always the case. Equation 3 can be very efficient if the sparse matrix has most of the nonzero elements in the diagonals and it only requires two separate vectors. CSR is the same as COO except that the row vector is compressed, creating storage savings. Locating 0.9 is done through the non-zero index, seven, and the row location, three.

$$\mathbf{ptr}[3] = 7, \mathbf{col}[7] = 0, \text{ and } \mathbf{data}[7] = 0.9.$$

Chapter 4 - Hardware Acceleration

Architecture

The Xilinx ML510 board has a Virtex 5 FXT FPGA, which has a built in PowerPC 440 processor [8]. For our experiment, we used a built-in PowerPC to run HPCCG. The PowerPC

will offload the HPCCG data to a hardware accelerator through the auxiliary processor unit (APU) port, also called the fast coprocessor bus (FCB). This port allows a fabric coprocessor module (FCM) to communicate with the PowerPC. User custom instructions (UDI) can be created using the APU port. It also has a separate 128-bit load and store buses for fast data transfers, along with other control signals. A Xilinx floating-point unit (FPU) can be attached through this port as well. A diagram of the PowerPC is located in Figure 8.

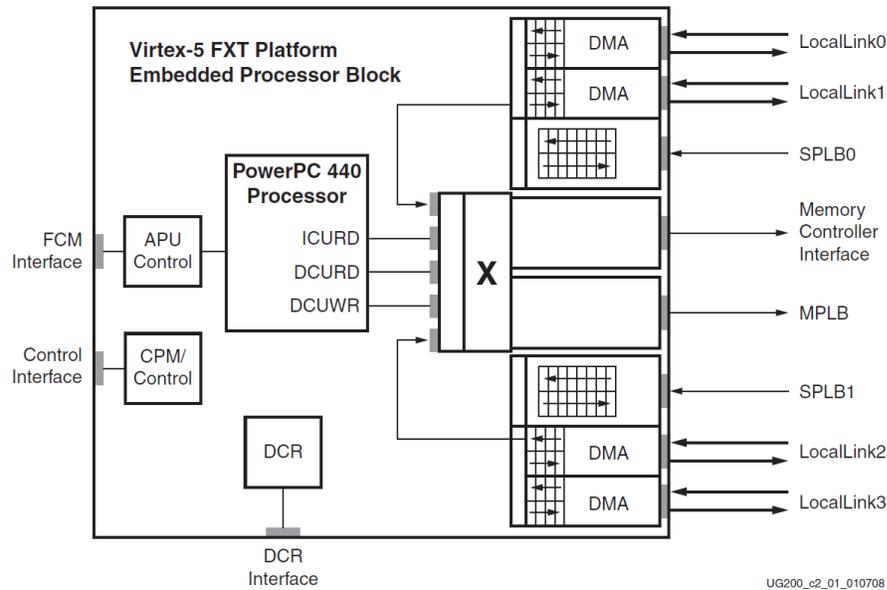


Figure 8 - Embedded Processor Block in Virtex-5 FPGA

Figure 9 represents a block diagram of the hardware accelerator architecture. The FCB wrapper interacts with the FCB and uses signals needed for the accelerator, which are the quadword and other command signals. A first-in-first-out (FIFO) memory was implemented between accelerator and FCB to allow the processor to stream data quickly. This also helps make sure no data is lost in transfers. The command signals specify which operation will take place,

weighted vector add or dot product. They also control which registers the processor needs. The accelerator controller is in charge of processing input and output data, including Vec0 and Vec1.

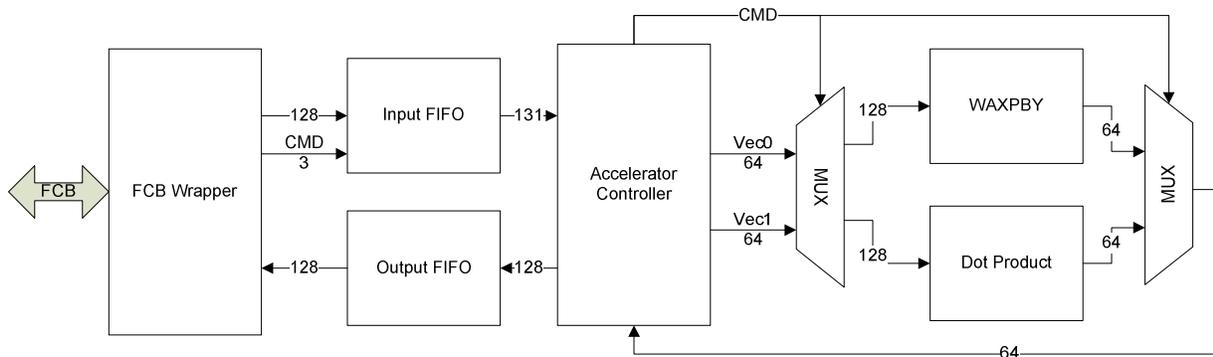


Figure 9 - Block Diagram of Accelerator

The hardware accelerator uses custom instructions to send signals to the hardware. Using these APU instructions requires programming to be done in assembly code. Custom instructions must be configured in hardware through the PowerPC IP. Three instructions were made for the hardware acceleration, located in Table 4.

Name	Description
UDI0	Reset hardware acceleration
UDI1	Start signal for Dot Product
UDI2	Writes into output FIFO

Table 4 - UDI Instructions

Along with custom instructions, the hardware accelerator has several registers that hold data. These are accessed through the 128-bit load and store instructions. Table 5 lists the registers and descriptions when writing to them. Reading can only be from two registers: zero and one. Reading register 0 returns the 128-bit data from the output FIFO. Register 1 returns the status of the accelerator.

Register	Description
0	Start Dot Product
1	Write lower 64-bits to Vector 0
2	Write upper 64-bits to Vector 0
3	Write 128-bits to Vector 0
4	Load 64-bit Alpha and Beta constants and start WXPBY
5	Write lower 64-bits to Vector 1
6	Write lower 64-bits to Vector 1
7	Write 128-bits to Vector 1
8	Starts sum of squares on Vector 0

Table 5 - Register Map for Accelerator

Another important issue to deal with is the amount of data sent to the accelerator at a time. Because only 128-bits can be sent at a time through the FCB, loops must be created to send the vectors during an operations, such as a dot product.. The number of times a loop will run is determined by the parameter HardwareStep. Increasing this value will lead to more 128-bit loads and stores to happen sequentially, while decreasing HardwareStep will break up the amount of sequential loads and stores. The same amount of transfers will still occur, but only the grouping of transfers will be affected. The APU controller is equipped to handle up to three loads or stores at a time [21]. This creates a small buffer between APU and FCM, which allows for pipelining when multiple instructions are sent. However, queuing too many transfers could result in overloading the bus, creating stalling. So an equilibrium must be found. Initially HardwareStep was set to 30. Tuning this value will be discussed later.

Dot Product

The dot product operation is defined by the equation:

$$a \cdot b = \sum_i^n a_i b_i$$

Equation 6 - Dot Product Equation

This operation is normally done with a simple for loop in C code, Equation 7

```
for i = 1 : N
    result += a[i] * b[i]
```

Equation 7 - Dot Product pseudo code

Equation 1 shows that there are two types of dot products: $\mathbf{r}^T \cdot \mathbf{r}$ and $\mathbf{p} \cdot \mathbf{Ap}$. The latter will be explained in more detail in the SPARSEMV Section. The former is simply the sum of squares of the vector \mathbf{r} . Knowing this process helps speed up the dot product process because only one vector needs to be transferred over to the hardware.

Several operations must be occurring simultaneously to achieve a speed up in hardware. To accomplish this, a pipelined approach was taken. As shown in Figure 10, data was received through a special two port memory system. This memory system allows sending two double precision numbers at once because the bus is 128-bits wide. Once both numbers are in memory, the multiplication commences. The output is then latched in a buffer and waits until a second multiplication result is finished. Once both numbers are done addition starts. Each accumulator output is fed into the input of another, creating a chain of accumulators. Moving the data in this fashion creates a pipeline, allowing for simultaneous operations to occur as long as the pipeline keeps busy.

With five accumulators chained together, the processor should send 32, or 2^5 , numbers to the input stream at a time. Increasing this chain can yield higher performance at the cost of size and power consumption of the chip. This will be described in more detail later on.

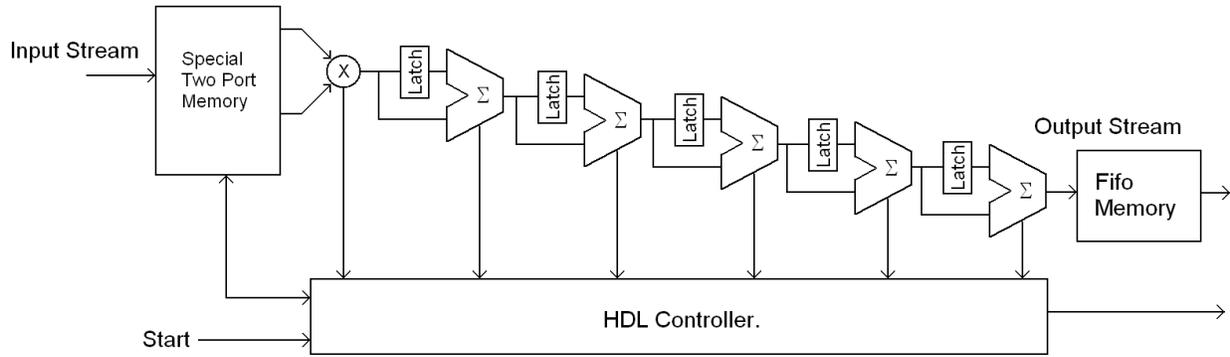


Figure 10 - Cascaded Dot Product

Figure 11 shows a timing diagram of these five accumulators chained together, where Adder0 - 4 represents the done signal of each accumulator. The DataAvail signal is ready after the last adder is finished. The MultiplyStart signal initiates the dot product calculation. Measuring from MultiplyStart to DataAvail gives about 97 clock cycles to compute a dot product between two double precision vectors.

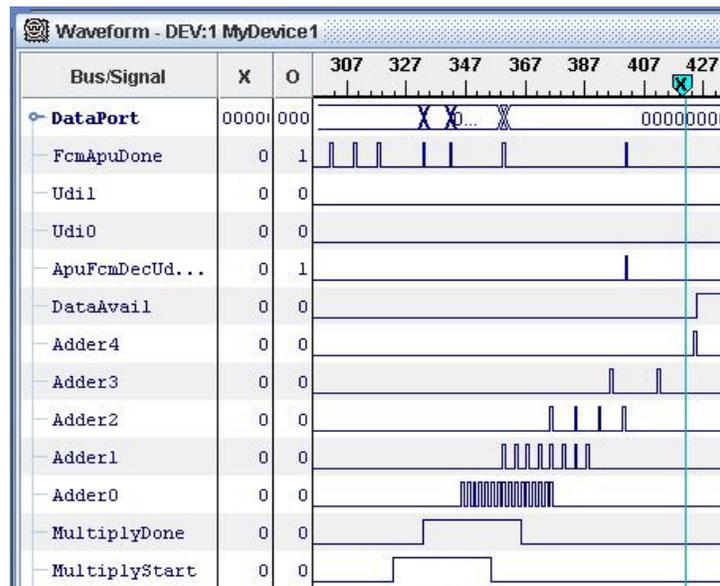


Figure 11 - Dot Product ChipScope Timing

WAXPBY

This operation computes the addition of two vectors, \mathbf{x} and \mathbf{y} , which are multiplied by weights, alpha and beta, as shown in Equation 8.

$$\mathbf{w} = \alpha \cdot \mathbf{x} + \beta \cdot \mathbf{y}$$

Equation 8 - WAXPBY Equation

Using the FPGA to accelerate the weighted vector addition is done by first loading in the two scalar weights alpha and beta. Then computing $\alpha \cdot \mathbf{x}$ and $\beta \cdot \mathbf{y}$ can be done concurrently. Once this process is completed, adding the two vectors together is done, creating pipeline, shown in Figure 12. A speedup should be achieved since both multiplies can be done simultaneously.

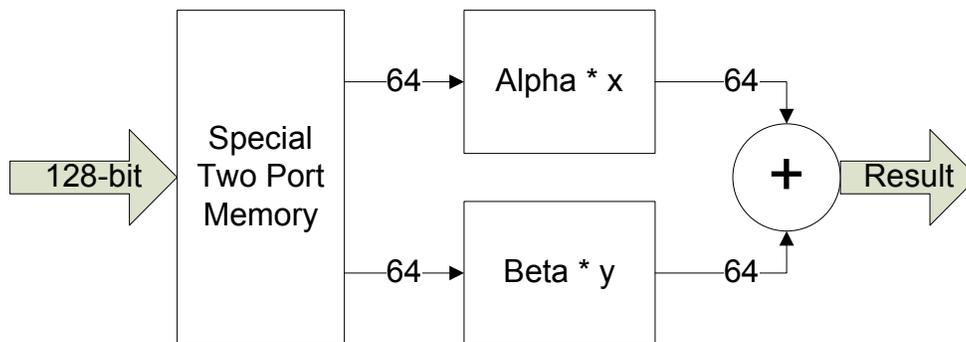


Figure 12 - Hardware Implementation of WAXPBY

Figure 13 shows timing to complete a single weighted vector add operation. From the start signal, StartWva, to the alpha and beta multiplies completing, AlphaDone and BetaDone, takes 5 clock cycles. Performing the addition of those results takes an additional 4 clock cycles, shown as WvaStrobe, making a weighted vector addition on two double precision numbers take 9 clock cycles. The additional transitions on the figure below are addition waxpby operations occurring in the pipeline.

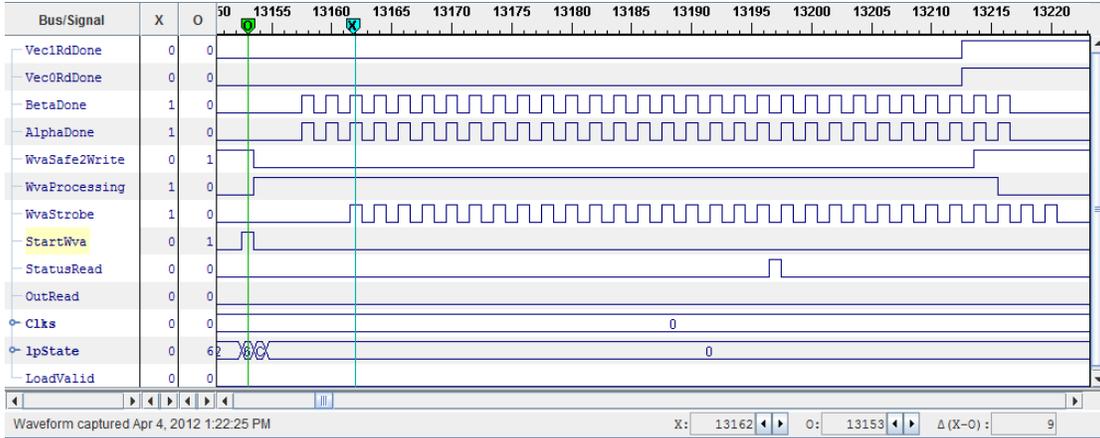


Figure 13 - ChipScope Timing of WAXPBY

SPARSEMV

The sparse matrix-vector multiply is calculated during the $\mathbf{A} \times \mathbf{p}$ operation which is a cross product, but with a sparse matrix instead of a dense one. A cross product is defined by

$$\mathbf{x} \times \mathbf{y} = x_1y_1 + \dots + x_Ny_N$$

Equation 9 - Cross Product Equation

where x_1 and y_1 represent the row and column vectors of the matrix, respectively.

Since one of the operands will always be a vector, optimizations can be made in hardware. Sending data to the accelerator can now be simply sending each row of the sparse matrix while the vector \mathbf{p} is only sent once. This is also the same process in the other type of dot product, $\mathbf{p} \cdot \mathbf{Ap}$, since \mathbf{Ap} has already been calculated at this stage in the program. The only difference between these two operations is that for the sparse matrix vector multiply a sparse row vector is sent instead of a row vector; however, to the accelerator the same operations occurs.

Figure 14 shows timing for completing a matrix vector dot product. The matrix is 32 x 32 doubles and the vector is also 32 doubles. Initially, the first row of the matrix is loaded into the dot product pipeline, followed by the vector. Now that the vector is in the accelerator, there is no

need to retransmit it because it is used for each dot product. The remaining rows can be sent sequentially. This optimization relieves stress from the FCB bus and should create a speedup.

The overall time to compute is 102.35 microseconds.

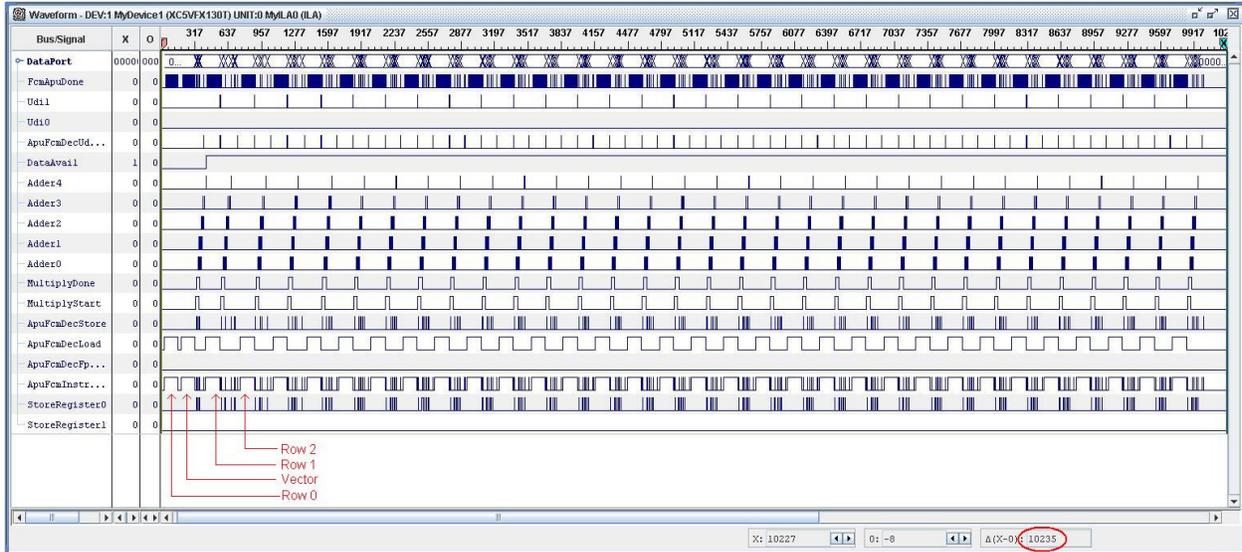


Figure 14 - ChipScope Timings for Matrix Vector Dot Products

Chapter 5 - Results

Using HPCCG as the benchmark, results were obtained by varying the input arguments. HPCCG accepts three numbers - nx, ny, and nz - which then represent the sparse matrix dimension size. The first approach of tests were done by creating a three dimensional structure, where each dimension was the same size, N x N x N. However, this approach was later changed to a single dimension, N x 1 x 1, which allows for finer granularity of results. N started at 1000 and incremented 100 each iteration; the maximum was 20000. This was chosen as the maximum to make run time at manageable levels.

Setup

HPCCG was implemented using two different cases: software implementation, and a custom hardware accelerator. Both implementations used the PowerPC running at 200 MHz with the FCB running at 100MHz. The software implementation used the PowerPC embedded on the FPGA along with a double precision FPU. This setup was chosen for software because in a typical system, the processor will offload some floating point operations to a FPU. The FPU is also located on the FCB, which is also the location of the hardware accelerator. This should make the memory bottleneck equivalent in both cases.

The hardware acceleration implementation also has a PowerPC but does not have an FPU. Instead, the custom accelerator built for HPCCG is connected via the APU port. Removing the FPU should lower the power needed in the system, assuming that the custom accelerator is more power efficient than the FPU. However, without the FPU certain operations, such as square root, on floating point numbers will take longer to compute on the PowerPC. The CG method only contains one square root and one divide operation per iteration. However, these operations will not be timed; only power measurements will be taken during these operations. If large variations exist between software and hardware, this may be the culprit.

Performance Metric

Creating a performance metric requires an understanding of the problem under test. For this case, the important results are power in watts and FLOPS. The number of floating point operations can be calculated by looking at the algorithm being implemented. They can be broken into three main categories: dot product, waxpby, and sparsemv. Each category is dependent on number of rows in the sparse matrix. The nonzero elements are determined by the number of

rows. The rows are simply the input arguments of HPCCG; $N \times 1 \times 1$ for our case. FLOP characteristics are located in Table 6.

Category	Floating Point Operations
Dot Product	(2 * Iterations) * 2 * rows
WAXPBY	(3 *Iterations) * 2 * rows
SparseMV	Iterations * 2 * nonzero elements

Table 6 - Floating Operations

After obtaining these values for each test completed, compiling the power data recorded was the next step. Following the steps stated earlier, in System Monitor, yields the instantaneous power in watts the FPGA is using. Average power recorded during each test was used instead because FLOPS are constant over each test. The metric chosen was FLOPS/Watt, which is equivalent to FLOP/Joule since a watt is a joule per second.

Along with power measurements, measurements of time per iteration were also chosen. Time per iteration was chosen as another metric because showing computation time will provide the full representation in performance. This also will show where most of the work is being done and show if any speedups were achieved. Plots of each category -- dot product, waxpby, and sparsemv -- were plotted along with total time of the CG solve.

Initial Results

The first comparison to be made is between hardware and software implementations, Figure 15 and Figure 16. In terms of timings, the dot product in hardware is faster than software. However, both waxpby and sparsemv hardware implementations are slower than software. Figure 15 also shows that most of the time spent in HPCCG is in the sparsemv operation. The anomalies in the hardware implementations for dot product and waxpby -- jumps in time/iteration -- are still unknown. Investigation in several areas was done to find possible

answers. Work was done checking memory alignment of the data to ensure it was on the correct boundaries, 128-bit. This had no affect on the anomalies. Adjusting the FCB clock frequency was done to see if there might be some contention with the bus. However, this only affected overall times and not the anomalies. A comparison between the CG solve converging and the timing anomalies was also done; but no correlations were found between them. Despite all the investigations, no solutions were found.

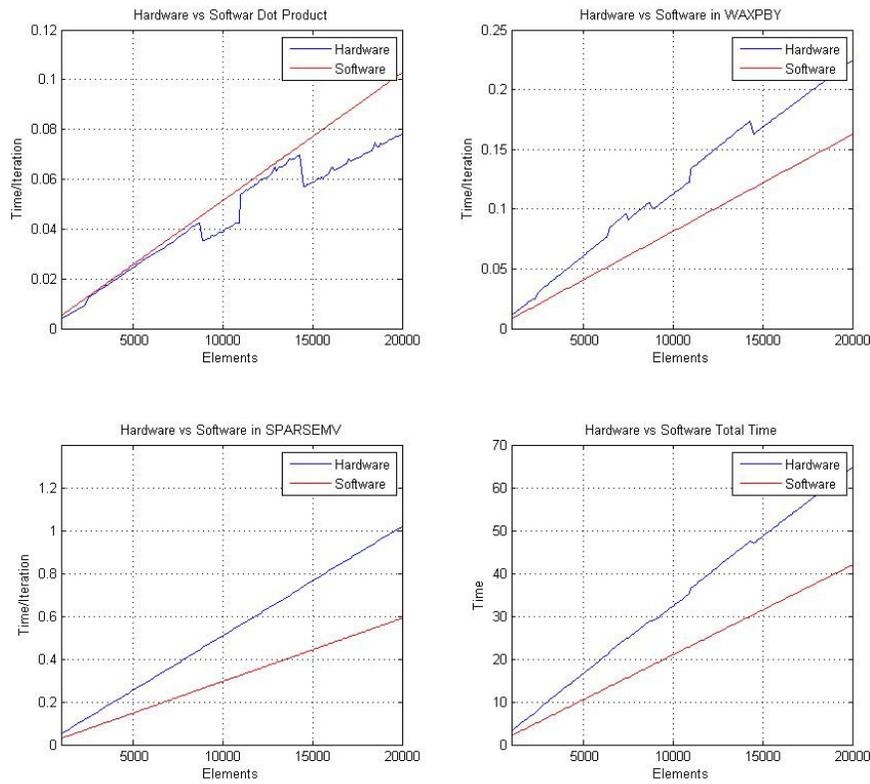


Figure 15 - Hardware vs. Software Time/iteration

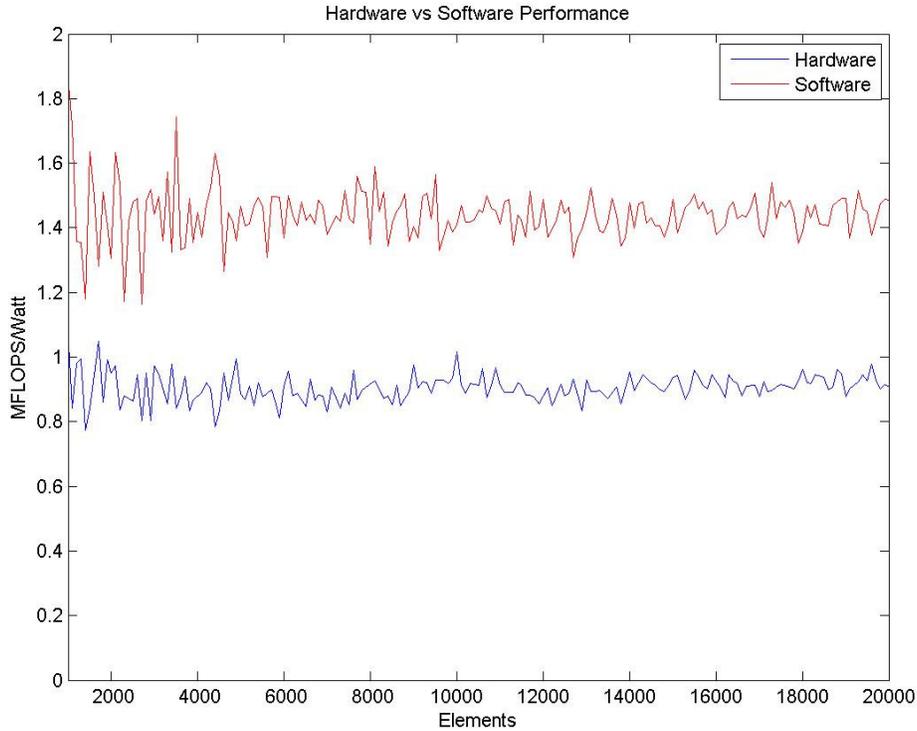


Figure 16 - Hardware vs. Software MFLOPS/Watt

Figure 16 shows that software can outperform the hardware acceleration in both power and performance. This is because software can finish the problem quicker than hardware and create power savings. Although these results differ with other hardware accelerators findings, [13], some acceleration was accomplished in the dot product. The sparsemv operation is the most intensive and if speedup can be found here then perhaps the hardware accelerator will outperform software.

Tuning

Most benchmarks have several options that can be adjusted to provide optimal performance. The same can be done with this accelerator. There are many options that can always be tweaked, however, two major but simple changes that can be done are adjusting the amount of accumulators in the dot product and altering the value of HardwareStep.

Accumulator Stages

Adjusting the stages of accumulators changes the amount of hardware needed to create the accelerator. To find the correct value of accumulators, the FLOPS/Watt must be at the optimal value. Several values were chosen for the number of accumulators: 5, 7, and 9. Accumulator stages were only changed; HardwareStep was constant during these tests. Additional stages of 3 and 11 were tried. With 3 stages the CG solve method took much longer to finish even the smallest element size, so it was thrown out. The case of eleven stages required too much hardware to implement and could not be built.

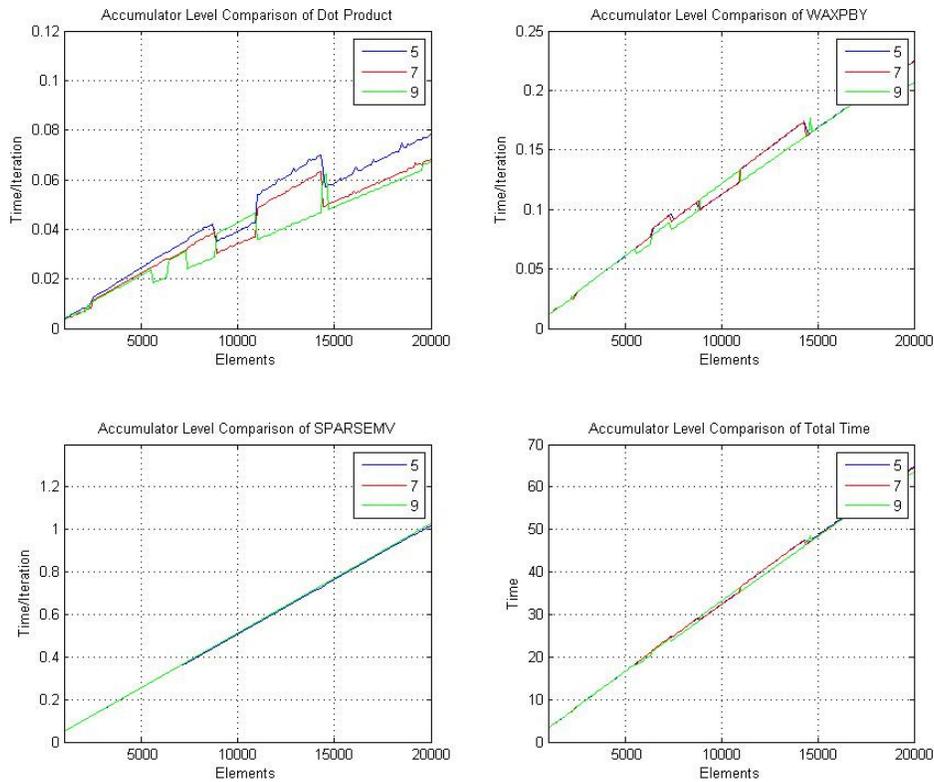


Figure 17 - Accumulator Stages Time/Iteration

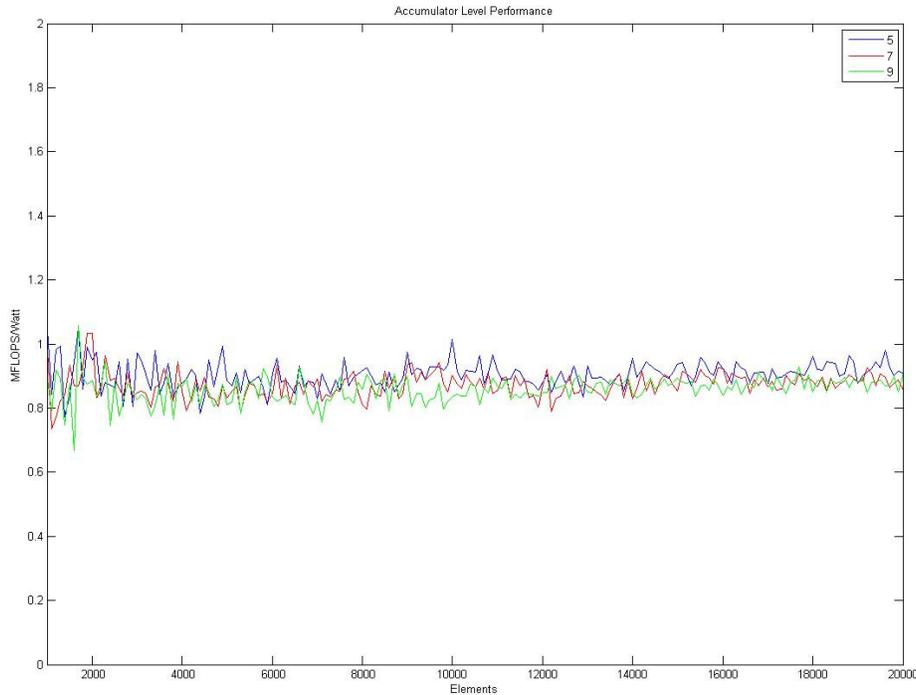


Figure 18 - Accumulator Stages MFLOPS/Watt

Figure 17 shows the change in time/iteration when changing accumulator stages from 5, 7, and 9. The only change that occurs between stage sizes is in the dot product, which is what is being tested. The difference between 9 stages and 7 stages is very small, around 3%, when anomalies aren't present. However, gain from 5 to 9 stages reaches around 13%, which shows that while adding stages to the dot product pipeline will increase performance, but there is a cap on performance, as shown above.

The total performance per watt achieved by adding more hardware is shown in Figure 18. The highest performance achieved was with only 5 stages. The reason for this is that the increase in power by adding more accumulators outweighs the small performance gain in the dot product operation. As a result, the ideal level is 5 stages.

HardwareStep Size

The other variable that was changed was HardwareStep. Changing this value is done in software. Several values were also chosen for this, 30, 60, and 120. The number of accumulator stages was unchanged during this experiment.

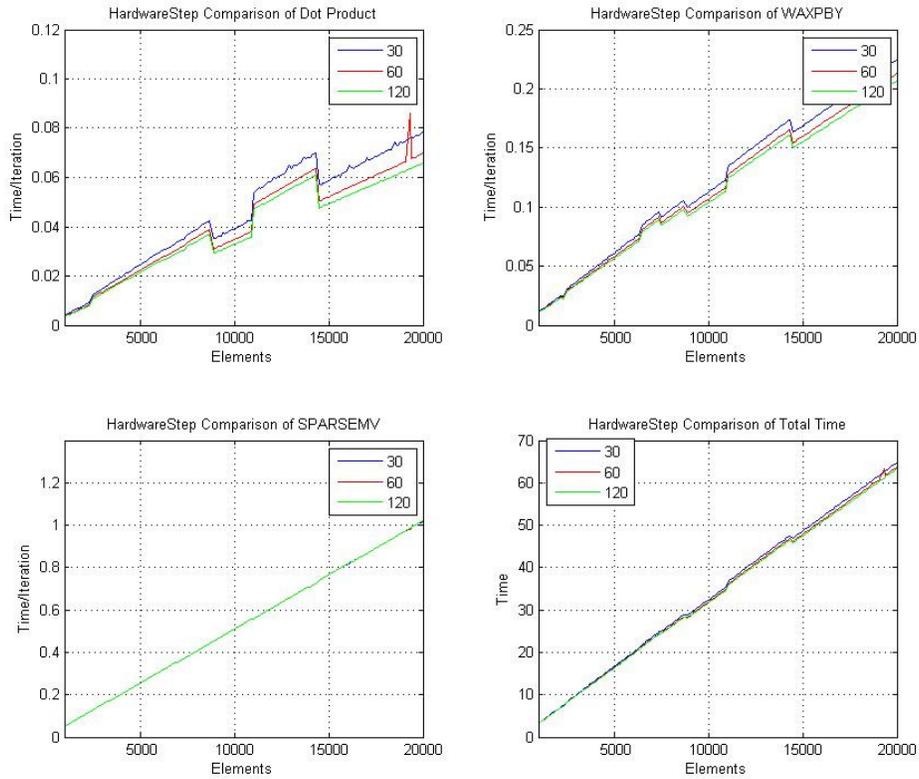


Figure 19 - HardwareStep Comparison Time/Iteration

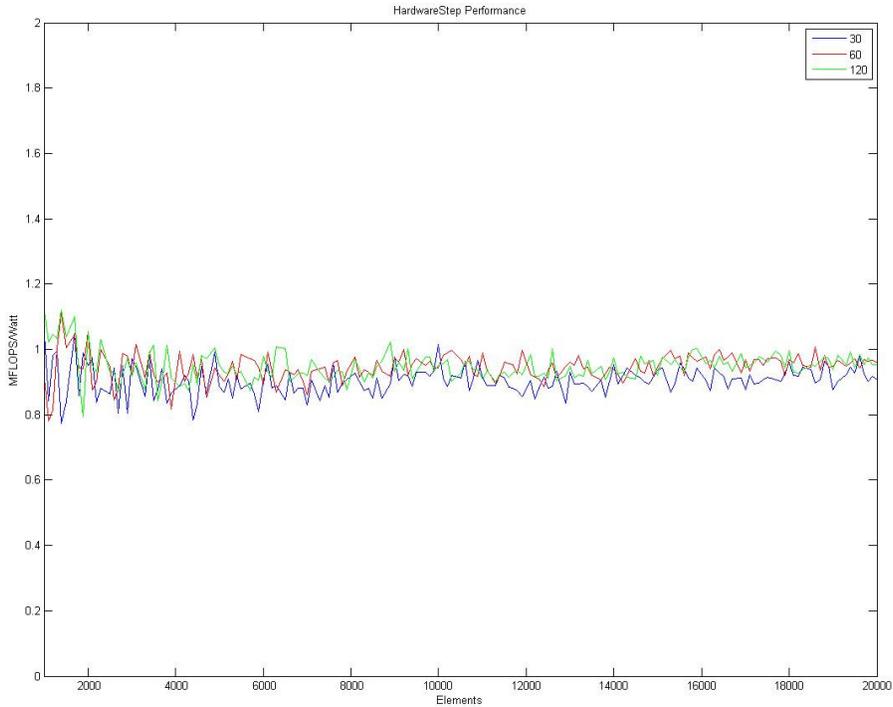


Figure 20 - HardwareStep Comparison MFLOPS/Watt

As shown in Figure 19, the largest gain achieved by increasing HardwareStep is in the dot product of 18%. Small increases in waxpby times are also gained by this and amounted to around 8%. The reason that the sparsemv operation does not improve from increases in HardwareStep is because only small chunks of data in one row are sent at a time. The sparse matrix is built so that the maximum number of non zero numbers is 27, which is why increasing HardwareStep has no effect. Figure 20 shows the same trend as the previous figure. Increasing HardwareStep has around a 5% of benefit. Sending larger streams of data to the accelerator is more efficient than smaller streams of data. However, the CG method rarely converged with increases to HardwareStep. This could be due to streaming too much data to the accelerator at a time and losing the information being sent.

Chapter 6 - Conclusion

Creating hardware accelerators for complex projects is performed frequently in many designs. GPUs, FPGAs, and other specialized hardware, such as a FPU, can make offloading time critical program segments possible and beneficial. These implementations are done in many consumer and industrial products. Along with the computation speedup achieved, power efficiency is also increased. Along with computing power, energy consumption has begun to be a critical specification in industry. In portable devices, such as cell phones and laptops, low power usage is essential. Large scale systems, such as supercomputers, also need to have lower power requirements for computation of problems to be possible.

An FPGA was chosen for this project to accelerate a common high performance computing algorithm called the conjugate gradient solve. Although the results recorded did not show increase in the total time, the dot product operation was faster in hardware than when implemented in software. Future work can also be done to create a more efficient method of accelerating the CG method. Regardless, the project showed that accelerating a specific process in hardware will create a speed-up.

Selecting a process to accelerate requires understanding common industry methods used, such as FIR filters, cryptography, or HPC applications. The CG method was chosen to accelerate because of its vast usage in complex applications. Rather than implementing a problem that requires the CG method, the HPCCG benchmark was chosen instead. This implements the characteristics of a real-world application without all the overhead needed with complex applications. The CG method consists of three operations: dot product, weighted vector addition, and sparse matrix vector multiply. Creating accelerators for these processes using an FPGA was

chosen to provide both timing acceleration of the process as well as comparisons in different operations.

Along with decreasing operation of time, accelerators can reduce the power required. Measuring this power was done using custom hardware on the FPGA. This allows the problem and power monitoring to occur independently of each other. Power usage was found by measuring the current used by the FPGA. Although, the monitor logic is located on the FPGA as well, the power required should be small.

Creating a system that can perform complex computation as well as monitor the power required is needed more in industry. This also allows a way to measure performance versus power consumption. This metric can then provide analysis of a system where both performance and power consumption are important parameters. Many supercomputers will give both power usage and performance, but not against each other.

Future Work

Additional work can be done in several areas to look at improving performance. The sparsemv operation should be further investigated because it is the most time consuming operation. Possible projects to implement:

- Alter accelerator architecture to accept more instructions and keep data on the accelerator. Doing this will decrease the amount of data needed on the FCB bus, which should improve performance.
- Use a higher clock frequency for the accelerator. This should allow operations to compute faster. However work needs to be done in the FCB wrapper module to implement two independent clocks.

- Make the sparse matrix vector multiply faster. This can be tried in several ways, such as sending multiple rows during each call or creating a burst transfer, which queues up instructions and sends when a buffer is full.
- Investigate the source of the anomalies found in the hardware accelerator results.
- Customize the waxpby operation to send alpha and beta and then the corresponding vectors sequentially, reducing the memory bottleneck.

Chapter 7 - Bibliography

- [1] B Dally, "Power, Programmability, and Granularity: The Challenges of ExaScale Computing," *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, p. 878, May 2011.
- [2] Peter Kogge, "The Tops in Flops," *IEEE Spectrum*, pp. 48-54, Feb 2011.
- [3] Top500. (2011, Nov) Top500. [Online]. <http://www.top500.org/lists/2011/11>
- [4] NVIDIA. (2011) NVIDIA. [Online]. <http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf>
- [5] Intel. Intel Core i7-3900 Desktop Processor Extreme Edition Series. [Online]. http://download.intel.com/support/processors/corei7ee/sb/core_i7-3900_d_x.pdf
- [6] Intel. 2nd Generation Intel Core i7 Extreme Processor. [Online]. <http://ark.intel.com/products/family/59135>
- [7] Microsemi. (2011, June) Dynamic Power Reduction in Flash FPGAs. PDF. [Online]. http://www.actel.com/documents/Dynamic_Power_Reduction_AN.pdf
- [8] Xilinx Inc. (2011, June) ML510 Embedded Development Platform User Guide. PDF. [Online]. http://www.xilinx.com/support/documentation/boards_and_kits/ug356.pdf
- [9] Zehan Cui, Yan Zhu, Yungang Bao, and Mingyu Chen, "A fine-grained component-level power measurement method," *Green Computing Conference and Workshops (IGCC)*, pp. 25-28, July 2011.
- [10] John D. McCalpin. STREAM. [Online]. <http://www.cs.virginia.edu/stream/>
- [11] Mersenne Research, Inc. Great Internet Mersenne Prime Search. [Online]. <http://www.mersenne.org/various/works.php>
- [12] Julian Seward. (2011, August) SPEC CPU2006 Benchmark. [Online]. <http://www.spec.org/auto/cpu2006/Docs/401.bzip2.html>
- [13] P. Possa, D. Schaille, and C Valderrama, "FPGA-based hardware acceleration: A CPU/accelerator interface exploration," *Electronics, Circuits and Systems (ICECS)*, pp. 347-377, Dec 2011.
- [14] Microsemi. Microsemi SoC Products Group. [Online]. <http://www.actel.com/products/igloonano/default.aspx>

- [15] Xilinx Inc. (2011, February) Virtex-5 FPGA System Monitor User Guide. Datasheet. [Online]. http://www.xilinx.com/support/documentation/user_guides/ug192.pdf
- [16] Sandia National Laboratories. Mantevo. [Online]. <https://software.sandia.gov/mantevo/>
- [17] Netlib. (1996, September) The Linpack Benchmark. [Online]. <http://www.netlib.org/benchmark/top500/lists/linpack.html>
- [18] Sandia National Laboratories. The Trilinos Project. [Online]. <http://trilinos.sandia.gov/>
- [19] Jonathan Richard Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," Carnegie Mellon University, Pittsburgh, 1994.
- [20] Nathan Bell and Michael Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA, 2008.
- [21] Xilinx Inc. (2010, February) Embedded Processor Block in Virtex-5 FPGA. PDF. [Online]. http://www.xilinx.com/support/documentation/user_guides/ug200.pdf