First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010), Atlanta, Georgia, December 5, 2010. Co-located with MICRO-43.

Programming the Convey HC-1 with ROCCC 2.0*

J. Villarreal, A. Park, R. Atadero
Jacquard Computing Inc.
(Jason,Adrian,Roby)
@jacquardcomputing.com

W. Najjar
UC Riverside
najjar@cs.ucr.edu

G. Edwards
Convey Computers
gedwards@conveycomputer.com

ABSTRACT

Utilizing FPGAs as hardware accelerators has been hampered by both the availability of affordable platforms and the lack of programming tools that bridge the gap between high-level procedural programming languages and the spatial computing paradigm that is implied on an FPGA. This paper reports on the experiences of programming the Convey Computers HC-1 system, a high-performance hybrid-core system consisting of eight 64-bit Intel Xeon processors coupled with four Xilinx Virtex 5 LX 330 FPGAs, using the ROCCC 2.0 toolset, an open source C to VHDL compilation framework specifically designed for the generation of FPGA-based code accelerators, which address both of these issues. The porting of the ROCCC 2.0 toolset was tested on Dynamic Time Warping, a data mining application, and the Viola-Jones face detection algorithm. We discuss the characteristics of these applications and the process of accelerating these applications through ROCCC by writing C that was compiled with ROCCC and mapped onto the HC-1.

1. INTRODUCTION

Multiple studies have repeatedly demonstrated the potential of FPGAs as high-performance computing platforms capable of achieving speedups measured in orders of magnitudes. However, their wider use has been impeded, to a large degree, by two challenges: (1) the availability of affordable high-performance computing platforms using FPGAs as accelerators, (2) programming tools that can bridge the gap between high-level procedural programming languages and the spatial computing paradigm that is implied on an FPGA device.

This paper reports on the experience of programming the Convey Computers HC-1 [2] systems using the ROCCC 2.0 (Riverside Optimizing Compiler for Configurable Computing) toolset [6]. The HC-1 is a high-performance computer systems consisting of an eight-core Intel Xeon with an FPGA coprocessor consisting of four Xilinx Virtex 5 LX 330 FPGAs. The cores and the FPGAs share a memory in a cache coherent mode. ROCCC is a C to VHDL compilation framework specifically designed for the generation of FPGA-based code accelerators.

The porting of the ROCCC toolset on the HC-1 was tested using Dynamic Time Warping (DTW) [7], a data mining application[1], and the Viola-Jones [10] object detection algorithm, a computer vision application.

This paper is organized as follows: Section 2 describes the HC-1 system. Section 3 describes the ROCCC 2.0 compilation framework as well as the interfacing between the ROCCC generated code (RGC) and the HC-1 system. Section 4 describes the implementation of the two applications, the challenges faced and the results obtained.

2. THE CONVEY HC-1 SYSTEM

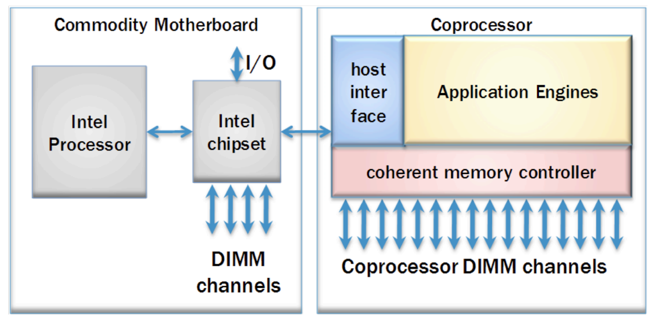

Figure 1: The Convey HC-1 System Architecture

2.1 System Architecture

The Convey HC-1 is a hybrid-core computer system that uses a commodity two-socket motherboard to combine a reconfigurable, FPGA-based coprocessor with an industry standard Intel 64 processor. The system architecture is shown in Figure 1. The coprocessor supports multiple instruction sets (referred to as "personalities"), which are optimized for different workloads and dynamically reloaded when an application is run. Each personality includes a base set of instructions that are common to all personalities, as well as extended instructions that are designed for a particular workload.

The coprocessor has a high bandwidth memory subsystem that is incorporated into the Intel coherent global memory space. Coprocessor instructions can therefore be thought of as extensions to the Intel instruction set—an executable can contain both Intel and coprocessor instructions, and those instructions exist in the same virtual and physical address space.

2.2 Coprocessor

The coprocessor has three major sets of components, referred to as the Application Engine Hub (AEH), the Memory Controllers (MCs), and the Application Engines (AEs).

The AEH is the central hub for the coprocessor. It implements the interface to the host processor and to the Intel I/O chipset, fetches and decodes instructions, executes scalar instructions and passes extended instructions to the AEs. The 8 Memory Controllers support a total of 16 DDR2 memory channels, providing an aggregate of over 80GB/sec of bandwidth to ECC protected memory. The MCs translate virtual to physical addresses on behalf of the AEs, and include snoop filters to minimize snoop traffic to the host processor. The Memory Controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs, which are optimized for transfers of 8-byte bursts and

* In this work W. Najjar is supported in part by NSF Awards CCF0905509 and CCF0811416.

[1] The compilation and FPGA performance evaluation of the DTW algorithm has been previously reported in [7].

provide near peak bandwidth for non-sequential 8-byte accesses. The coprocessor therefore provides a much higher peak bandwidth, and often can deliver a much higher percentage of that bandwidth, than what is available to commodity processors.

The infrastructure provided by the AEH and MCs is common to all personalities, ensuring that access to coprocessor memory, memory protection, and communication with the host processor are always available.

The Application Engines (AEs) implement the extended instructions that deliver performance for a personality. The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands, and to the memory controllers via a network of point-to-point links that provide very high sustained bandwidth. Each AE instruction is passed to all four AEs. How they process the instructions depends on the personality. For instance, a personality that implements a vector model might implement multiple arithmetic pipelines in each AE, and divide the elements of a vector across all the pipelines to be processed in parallel.

2.3 Personalities

A personality defines the set of instructions supported by the coprocessor, as well as the behavior of those instructions. A system may have multiple personalities installed and can switch between them dynamically to execute different types of code, but only one personality is active on a coprocessor at any one time. Each installed personality includes the loadable bit files that implement a coprocessor instruction set, a list of the instructions supported by that personality, and an ID used by the application to load the correct image at runtime.

A scalar instruction set is available to all personalities. These instructions include scalar arithmetic, conditionals, branches, and other operations required to implement loops and manage the operation of the coprocessor. In addition to the scalar instructions, each personality includes extended instructions that may be unique to that personality. Extended instructions are designed for particular workloads, and may include only the operations that represent the largest portion of the execution time for an application. For instance, a personality designed for seismic processing may implement 32-bit complex vector arithmetic instructions.

Coprocessor instructions use virtual addresses compatible with the Intel® 64 specification, and coherently share memory with the host processor. The host processor and I/O system can access coprocessor memory and the coprocessor can access host memory. The virtual memory implementation provides protection for process address spaces as in a conventional system. Figure 2 shows the architecture of a coprocessor personality.

The common elements to all personalities--Instruction Set Architecture, instruction dispatch interface and virtual memory--ensure that compilers and other tools can be leveraged across multiple personalities, while still allowing customization for different workloads.

2.4 Personality Development Kit

The Personality Development Kit is a set of tools and infrastructure that enables a user to develop a custom personality for the HC-1 system. A set of generic instructions and defined machine state in the Convey ISA allows the user to define the behavior of the personality. Logic libraries included in the PDK provide the interfaces to the scalar processor, memory controllers and to the management processor for debug. The user develops custom logic that connects to these interfaces. Functional

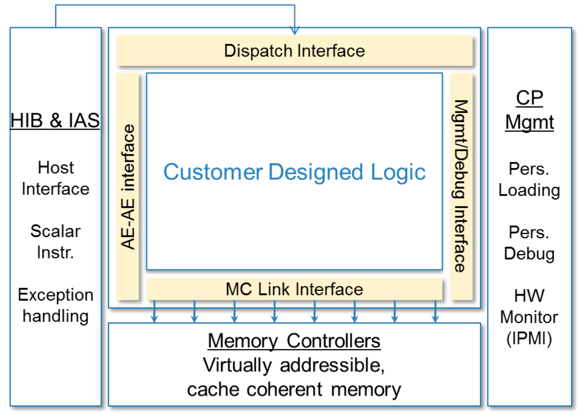

Figure 2 : FPGA Personality Architecture

verification of the personality is done using the Convey architectural simulator, which allows users to verify the design using the application to drive the simulation. The PDK also includes a tool-flow to ease the process of generating the FPGA image to run on the system.

3.THE ROCCC 2.0 TOOLSET

3.1 ROCCC 2.0 Program Structure

ROCCC 2.0 supports a bottom-up modular approach to hardware accelerator design in which components are written in a strict subset of C and used to build larger components and systems. The ROCCC 2.0 compiler generates VHDL for two types of C code, which are referred to as *modules* and *systems*.

Modules are concrete hardware blocks that perform a computation on a known number of inputs in a fixed delay and are fully pipelined, generating a fixed number of outputs every clock cycle. Once compiled through ROCCC, modules are available for integration in larger code through standard C function calls. ROCCC also supports the importing of module code from other sources, such as predefined VHDL or FPGA-specific netlists.

Figure 3 shows an example ROCCC module that takes an integer and returns that integer to the 10^{th} power. Modules require a definition of a *struct* that identifies all of the inputs and outputs to the hardware block. Variables of the *struct* with the suffix "*_in*" are identified as inputs and variables with the suffix "*_out*" are identified as outputs. The computation of a module must be described in a function that both takes and returns an instance of this *struct*. The function performs the computation and must read from all of the inputs and assign a value to all outputs. When compiled, modules are placed in a database (as an integral part of the ROCCC GUI) and accessible to other ROCCC code, with all properties such as latency reported to the user.

ROCCC systems are critical loops that perform extensive computations on large streams of data or windows in memory. Generic memory interfaces are created for each input and output *stream* detected in the system code and configured based upon their use. Data reuse between consecutive loop iterations is detected and used to minimize the number of memory fetches generated. Systems can utilize all available modules, which integrate directly into the pipelined structure created. The loops contained in system code are heavily optimized to extract parallelism and maximize throughput, and the hardware generated can further be guided by explicit user control of optimizations.

ROCCC supports both integers of any bit width and floating point values through the addition of intrinsic cores. The hardware

```
typedef struct
{
  int x_in ;
  int result_out ;
} pow_t ;

pow_t Pow(pow_t p)
{
  int i ;
  int total ;
  total = 1 ;
  for (i = 0 ; i < 10 ; ++i)
    total *= p.x_in ;
  p.result_out = total ;
  return p ;
}
```

Figure 3: ROCCC 2.0 Module Code

support for these operations can come from a variety of known library components, such as Xilinx CoreGen [11], as long as the compiler is made aware of their existence.

3.2 Compilation Flow

ROCCC code goes through several phases when being compiled into hardware. First, high level transformations are performed and information is collected. This is then passed in an intermediate format we refer to as Hi-CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics) that is then further optimized to extract parallelism and is transformed into a pure data-flow graph that represents the constructed hardware.

At each stage of compilation the user has control over both the choice and scope of optimizations to perform. At the high level, a large number of standard software optimizations are performed such as constant propagation and scalar replacement. Also, users have the choice of choosing additional optimizations such as loop unrolling. Unlike software compilers, ROCCC offers greater control over loop unrolling and allows users to select individual loops to be unrolled in a user-defined order along with the amount they should be unrolled. This allows for users to maximize the bandwidth of the platform they are targeting.

In addition to standard compiler optimizations, there are high level optimizations that are specifically targeted to generating hardware. ROCCC can generate a one dimensional hardware systolic array from C code that is written as a wavefront algorithm on a two dimensional array if the user enables systolic array generation. Similarly, temporal common subexpression elimination will find common expressions across loop iterations and replace them with feedback variables, decreasing the amount of area used by the generated circuit.

At the high level, streams and memories are identified by analyzing array accesses inside of loops. Information regarding the window size and reuse patterns of each stream is collected and passed to the low level hardware generation. The user has the option of tuning each stream as appropriate for the targeted platform.

At the low level, users may select if generated integer arithmetic truncates bits at every step or increases precision until operations are performed. For example, if two eight bit integers are multiplied together and then added to a sixteen bit integer the user can chose if the multiplication is stored in a sixteen bit integer and then added with another sixteen bit integer to have a final result of seventeen bits or if the multiplication is truncated to eight bits before being added to the sixteen bit number. Additionally, users have the option of transforming long chains of arithmetic operations into a balanced tree structure or keeping the generated hardware as a chain. This optimization is left as a choice to the user as floating point operations may produce different, but still correct, results than software when the order of operations is different than software.

Fine grained control over pipelining is achieved by allowing the user to select weights for various operations as they correspond to the actual cost of these operations on the platform being targeted. ROCCC will traverse over the data flow graph and combine instructions into the same pipeline stage if the combined weight is less than the maximum weight per cycle as defined by the user, allowing the user to directly control the number of pipeline stages generated and trade-off latency and clock speed as appropriate.

3.3 Interfacing to Platforms

On a CPU the interface implicit in all ISAs is that of a large and uniformly accessible memory (not necessarily a flat address space). The communication to/from the outside world is done via memory mapped I/O. There is no equivalent standard model with FPGAs. Building on the experience acquired with the first version of ROCCC, it was clear that the code generated by the compiler should be made, as much as possible, platform agnostic to prevent a blow up in complexity of the compiler. The ROCCC 2.0 compiler supports two abstractions for interfacing to external devices: a randomly addressed memory and a stream.

A memory interface is generated for each stream that was detected during compilation. If the window of accesses for a stream is a noncontiguous memory block, for example a 3x3 *sliding window* over a picture, a memory interface and address generator is created to read consecutive windows. If the window of access for a stream is strictly contiguous we generate a stream interface that acts as a fifo and does not generate addresses to fetch.

ROCCC code treats memories as being word addressable, where the word size is determined by the size of the individual elements in the stream in the original C code. For example, if the original C code read from an array of 8-bit integers, the generated memory interface would create an address and issue a fetch for each 8-bit value that needed to be read, regardless of the underlying platform.

In order to interface the generated ROCCC memory interface to an actual platform, several parameters for tuning are available to the user. First, the size of the individual elements of the stream may be changed to match the native word length of the platform, although this inefficient in most cases.

More importantly, the amount of incoming and outgoing channels per stream may be adjusted. By default, each stream interface expects to read one value per clock cycle. On a stream-by-stream basis, the user can change the number of values read each clock cycle. For example, the user may change an 8-bit stream to read four values every clock cycle, enabling that stream to be connected to a platform specific 32-bit bus with little to no overhead glue logic.

Another parameter that can be adjusted for each stream is the number of outstanding memory requests. By default, ROCCC generated code creates interfaces that allow for one memory request per stream at a time. This enables fast access to data stored close to the generated circuit, such as data in BRAMs, but proves problematic when the memory fetches have a large latency. The number of outgoing memory requests can be changed to any number and the only requirement ROCCC expects is that the data is returned in the order it is requested.

With all ROCCC generated memory interfaces, internal data reuse is handled through the creation of a smart buffer [3]. The smart buffer stores data that is used across loop iterations, allowing for the minimal amount of data to be read between activations of the data path.

3.4 The ROCCC GUI

The ROCCC GUI is built as an Eclipse plug-in. It provides the user with full control over the programming, compilation and interfacing of the ROCCC code. It provides an integrated way to manage the instantiation of modules and cores into C code, control both high level and low level (including pipelining) transformations, interface with generic structures such as BRAMs, and generate testbenches for verification.

4. APPLICATIONS

In this section we describe the ROCCC implementation of two applications on the Convey Computers HC-1: Dynamic Time Warping and Viola-Jones Face Detection.

4.1 Dynamic Time Warping

Subsequence similarity search, the task of finding a region of a much longer time series that matches a specified query time series within a given threshold, is a fundamental subroutine in many higher level data mining tasks such as motif discovery, anomaly detection, association discovery, and classification. Dynamic Time Warping (DTW) has been shown to be the best form of determining distance between two time series across a wide range of domains.

The DTW algorithm as described in [7] searches for a subsequence that is 128 characters long. The time series being compared to this sequence must be normalized, but the entire time series cannot be normalized once and processed. Instead, each substring of length 128 extracted from the time series must be normalized against itself before being compared with the candidate subsequence.

The software algorithm for computing DTW first creates a normalized subsequence and then performs a wavefront algorithm on a two dimensional array, creating a warping matrix that describes the optimal alignment of the candidate subsequence and the current position of the time series. This is then repeated for each position in the time series and is very computationally expensive.

Our hardware design is based off of the hardware as described in [7]. The design is split into two major components: an online normalizer as shown in Figure 4 and a systolic array that calculates the warping matrix. Both components were coded in C and generated with ROCCC. The normalizer was written as a system of modules that generated a normalized subsequence every clock cycle. The warping matrix was written as a wavefront algorithm on a two dimensional array that was transformed through ROCCC optimizations into a systolic array.

The warping matrix takes 128 cycles to completely process a substring and must then be reset before processing the next substring. Since the normalizer generates substrings each cycle, the complete circuit reaches maximum throughput with 128 instances of the warping matrix.

We explored several options for partitioning the hardware onto the FPGAs on the Convey HC-1. We first explored moving the normalizer to software and filling the hardware with systolic array warping matrices that read from memory. After synthesizing to the Virtex 5 LV 330 on the HC-1, we discovered that a single warping matrix on the FPGA ran at 95 MHz and took up 38% of the area, which means a single systolic array can process approximately 742,000 subsequences per second assuming maximum bandwidth. Running in software, the normalizer component was only able to generate 3137 subsequences per second, or 203x less than we can support. With this result, we decided to move the normalizer into the FPGA.

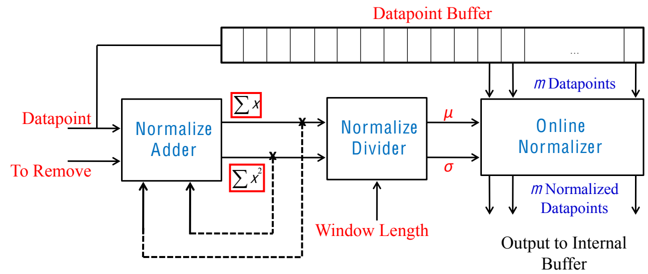

Figure 4: Block Diagram of the Hardware Normalizer

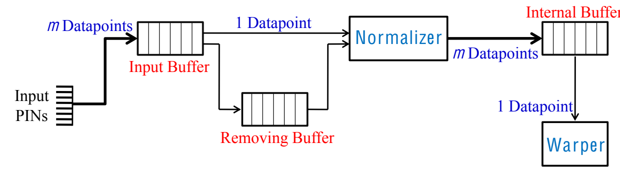

Figure 5: Block Diagram of the DTW Circuit

This resulted in three different options we explored. The first option was to place the normalizer and one systolic array on one FPGA. We were able to fit two systolic arrays on the FPGA. This lead to the third option, in which we put the normalizer on one FPGA, wrote the results out to memory, and then placed the systolic arrays on another FPGA , which read the normalized values from memory.

When we synthesized a combined normalizer and systolic array, we found that the resulting circuit was able to run at 115MHz and required approximately 40% of the area, resulting in a total of approximately 900K subsequences per second processed at maximum bandwidth.

Our next option was to run the normalizer on one FPGA and run 2 systolic array warpers on a different FPGA, with the communication occurring through memory. The normalizer portion alone ran at 152 MHz, took 3% of the slices and was able to generate 1,187,500 subsequences per second at maximum bandwidth. The two systolic arrays ran at 60 MHz, took 75% of an FPGA, and could process approximately 937,500 subsequences per second at maximum bandwidth.

Further tuning is available in ROCCC by selecting the number of channels each stream can support. The data handled by the hardware DTW is 8-bit integers, while the Convey HC-1 has a native 64-bit data bus. ROCCC allows us to maintain maximum bandwidth for a given architecture by changing the number of incoming channels to match the available bandwidth.

The ROCCC generated code does not know the base address of the streams, but instead only generates offsets into each stream. The software running on the Convey machine is responsible for both managing the transfer of stream data to the hardware and the passing of the base address in memory of each stream in a register to the hardware. This is performed through several calls to the personality API.

The address space on the Convey machine is typically partitioned based upon three bits of the address to one of eight memories, which makes connecting the streams to a particular memory controller a tricky proposition when the addresses of the data to be passed to the hardware is unknown at compile time. In order to make the porting of ROCCC code easier, a cross-bar was created that enables any stream to be connected to any memory controller

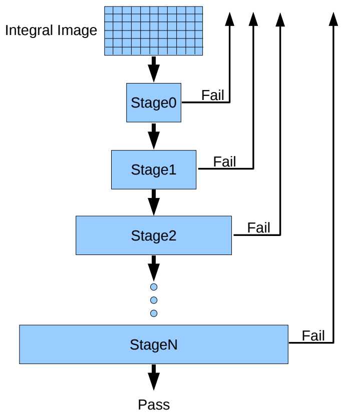

Figure 6: Software Structure of Viola-Jones Algorithm

and the addresses generated will be routed to the correct memory partition.

4.2 Viola-Jones Face Detection

Viola-Jones, as presented in [10], detects the existence of faces in an image by searching the subwindows of the image and running a multi-stage classifier cascade against that subwindow. Each stage of the classifier cascade can detect the presence of faces with a very low rate of false negatives, and a relatively low rate of false positives. By chaining multiple stages of classifiers into a cascade, faces correctly detected in one classifier stage will continue to pass further classifier stages, while non-faces falsely detected are eliminated by more rigorous classifier stages. Each classifier stage itself is composed of many features, which are very weak classifiers capable of around a 51% accuracy rate. Multiple weak classifiers can be grouped together to create a strong classifier, as shown in [6]. Particularly, the number of features in each stage determines both the accuracy and the computation time needed to evaluate that classifier stage, with more features giving a greater accuracy at the cost of computation time. One benefit of Viola-Jones over previous algorithms is the idea of using smaller classifier stages at the beginning of the classifier cascade; windows which are obviously not faces can be quickly eliminated, while windows requiring more computation time are passed along to classifier stages with more features, and thus more accuracy, for further processing. The structure of the software algorithm is shown in Figure 6.

A classifier stage classifies a window by summing all of the feature's calculated value and comparing that sum against a stage threshold. Each feature in a classifier stage returns a weight, which are summed and compared to the stage threshold. Each feature in Viola-Jones is a Haar-like feature, consisting of either two, three, or four rectangular regions of the current subwindow of the image. Each rectangular region of the feature is evaluated by summing the pixel values within that region, then multiplying by a particular weight for that region. As the weights can be either positive or negative, a difference of summed areas can be evaluated. This difference of summed areas is compared to a threshold and one of two weights is returned based on this comparison. Viola-Jones works on an Integral Image, which enables a reduction in the number of calculations necessary in determining the area of region into 9 arithmetic expressions.

Viola-Jones requires that the subwindow currently being processed is variance normalized in order to minimize the effects of different lighting conditions, as the weights chosen reflect a

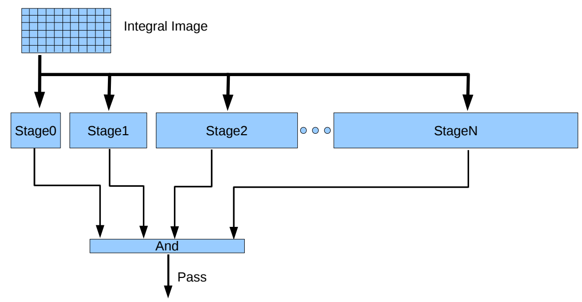

Figure 7: Hardware Structure of Viola-Jones Algorithm

similar variance normalization done at training time. Thus, for every subwindow, it is necessary to find both the standard deviation and the mean of the pixel values of the subwindow, and to then preprocess each pixel by subtracting off the mean and dividing by the standard_deviation. As the variance normalization is done at the subwindow level, with the mean and standard deviation being calculated off of the values of the pixels in the subwindow, the variance normalization cannot merely be done on the image as a whole. Particularly in hardware, this represents a very costly set of operations to perform.

When analyzing the algorithm, we determined that the cascading structure of the algorithm is a software convention to eliminate clock cycles. Thus, the cascade architecture is much less useful for hardware than for software. However, one major benefit of hardware over software is that all of the stages can be calculated in parallel, as there are no dependencies between them. Instead, the final decision of whether the subwindow contains a face or not depends on all of the stages passing. Our architecture takes advantage of this by parallelizing all stages to perform calculations simultaneously, as shown in Figure 7.

The original Viola-Jones algorithm includes three issues that make porting to hardware particularly difficult. These three issues were usage of floating point, use of division, and use of square root. All three issues involved operations that were extremely expensive in hardware.

In hardware, floating point operations are much more expensive than integer operations of the same type, in both area and frequency. We took advantage of the fact that the algorithm involves multiplying floating point constants against integers, and converted the floating point operations to fixed point operations. Each floating point constant was multiplied by 2^{12}, which gave consistent results with the other 32 bit integer calculations done by the feature calculation.

In order to find the mean and standard deviation of the subwindow currently being processed, integer division by constants was necessary. We replaced all integer divisions by constants with an approximation of the form $a/b \sim= (((2^k)+b-1)/b * a)>>k$. As k and b are both constants, $((2^k+b-1)/b$ can be calculated at compile time; k is chosen by profiling the code and finding a value between 1 and 32 that works well, which we found to be 16.

The calculation for standard deviation involves taking the square root of the variance; square root is generally available as an IP core. Through profiling, we were able to determine that the standard deviation was never more than 120, which meant that a general purpose 32-bit square root core was not necessary; rather, only a 16 bit square root core was necessary. We composed a module in ROCCC that performed the square root of 16 bit values by binary searching for the largest number whose square was smaller than the value in question; this generated a module, callable from ROCCC, that could perform square root in one clock cycle.

Porting to the Convey architecture was straightforward and involved mapping each stream to a memory controller and selecting a total of 256 outgoing memory requests.

While placing all of the approximately 3000 features necessary for the software Viola-Jones on a single FPGA is infeasible, splitting the design onto multiple FPGAs is possible due to our completely parallel architecture. We synthesized the first six stages of the face detection algorithm (189 features) and took up approximately 37% of the FPGA running at 22MHz. Since all features are the same computation with different constants, this scales to approximately 510 features per FPGA, or 2040 total features on the Convey HC-1, giving us approximately 343 600x600 frames per second at full potential.

Since this falls short of the number of features in software, we looked into how to reduce this number. We found that the initial stages were more amenable to software speedup than computational accuracy, so we could use only the last 8 stages with results that were very close to the original algorithm, which contained 1467 total features. As future work we plan to change the algorithm to better reflect the parallel hardware architecture. By retraining the classifiers to take advantage of a parallel architecture, it should be possible to maximize the detection rate for the given number of features or decrease the total number of necessary features.

To verify correctness of our face detection modifications, we ran our modified algorithm on several images collected from the MIT + CMU data test set [1], the Open CV distribution [5], and various pictures of faces taken from Google Image Search and compared them to the results from the original software implementation. The results are summarized in Table 1. The percentage difference refers strictly to the number of faces detected with both algorithms. False positives represent values reported as a face in the modified algorithm that are not identified in the original algorithm while false negatives are values that are detected in the original algorithm and not identified in the modified algorithm.

In a few instances, we had differences of 25% or higher when comparing the number of faces in the original algorithm to our modified version. Some of these high percentage differences, such as the instance with the Mona Lisa image, are a little misleading. Since the original version collected six face regions on the Mona Lisa and our modified collected three more, the percentage was large because the original number was small to begin with. Also, most of these false positives and negatives are purely an instance of the modified algorithm collecting the face a few pixels shifted away compared to the original algorithm.

5. CONCLUSION

We have reported on our experiences programming the Convey Computers HC-1 system using the ROCCC 2.0 toolset. By creating implementations of both Dynamic Time Warping and Viola-Jones appropriate for the HC-1 through writing C code and tuning with optimizations, we have shown a potential of 900 thousand subsequences per second processed with DTW and 343 frames per second on face detection.

Figure 8: Difference in results from Original Viola-Jones (left) and Modified Algorithm (right) on ManySizes

Image Name	Faces on Original	Faces on Modified	% difference	False Positives	False Negatives
Friends	6	6	0	0	0
Addams	26	26	0	0	0
NewsRadio	20	27	25.925	7	6
ManySizes	39	38	2.613	5	6
MonaLisa	9	6	50	1	4
ManyFaces	151	154	1.948	16	13
Wright	2	2	0	0	0

Table 1: Original Viola-Jones Algorithm versus Modified Algorithm.

REFERENCES

[1] CMU Image Database. http://vasc.ri.cmu.edu/idb/html/face/frontal_images/

[2] Convey Computers. http://www.conveycomputer.com/

[3] Z. Guo, A. Buyukkurt, and W. Najjar. Input Data Reuse in Compiling Window Operations Onto Reconfigurable Hardware. In ACM Symposium on Languages, Compilers, and Tools for Embedded Systems, 2004.

[4] E. Keogh. Exact Indexing of Dynamic Time Warping. VLDB 2002: 406-417.

[5] Open Source Computer Vision. http://opencv.willowgarage.com/wiki/

[6] ROCCC 2.0. http://roccc.cs.ucr.edu

[7] D. Sart, A. Mueen, W. Najjar, V. Niennattrakul, and E. Keogh. Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs, in Int. Conf. on Data Mining, Sydney, Australia, Dec. 2010.

[8] R. Schapire. The Boosting Approach to Machine Learning: An Overview. In the MSRI Workshop on Nonlinear Estimation and Classification, 2002.

[9] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C With ROCCC 2.0. The 18th International IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2010.

[10] P. Viola and M. Jones: Robust Real-time Object Detection, IJCV 2001.

[11] Xilinx Core Generator. http://www.xilinx.com/ise/products/coregen_overview.pdf