FPGA based High Performance Computing

February 19, 2008

Dave Bennett¹, Eric Dellinger¹, Jeff Mason¹, Prasanna Sundararajan²
Andrew Putnam³, Olaf Storaasli⁴

¹Xilinx Research Labs, 3100 Logic Drive, Longmont, Colorado 80503
²Xilinx Research Labs, 2100 Logic Drive, San Jose, California 95124
³University of Washington, Box 352350, Seattle, WA 98195-2350
⁴Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge TN 37831

ABSTRACT

Current high performance computing (HPC) applications are found in many consumer, industrial and research fields. From web searches to auto crash simulations to weather predictions, these applications require large amounts of power by the compute farms or supercomputers required to run them. The demand for more and faster computation continues to increase along with a sharper increase in the cost of the power required to operate and cool these installations. The ability of standard processor based systems to address these needs has declined in both speed of computation and in power consumption over the past few years. HPC is in need of new and novel solutions to these problems. This paper presents a new method of computation based upon programmable logic as represented by Field Programmable Gate Arrays (FPGAs) that addresses these needs in a manner requiring only minimal changes to the current software design environment.

1. INTRODUCTION

Over the past few years, it has become apparent that standard Von Neumann based processor architectures are running out of steam [1]. No longer do users see the expected Moore’s law increase of density and clock speed that the industry has come to expect implying faster or cheaper execution. The increase in density is still occurring but it is no longer translating into a proportional increase in performance. Since 2002, the annual performance increase is less than 20%. No longer can one expect the processor of next year to perform 2 times faster than the one of today. This is due to a number of reasons and is the subject of much research in the industry [2].

Programmable logic, as represented by Xilinx FPGAs has been looked upon as a possible solution for this problem. Unfortunately, a few major obstacles have prevented programmable logic from realizing its potential in the HPC marketplace. These limitations can be broken down into three broad categories:

1. Physical characteristics and resources: not enough logic or routing
2. Difficulty of use: programming model is difficult or impossible
3. Conceptual acceptance: does not run same code as other platforms

With the use of modern FPGAs, compute intensive system boards/platforms and the introduction of the toolset described below, each of these limitations is addressed such that FPGA high performance computing (FHPC) is directly accessible to the standard HPC programmer.
Over the past 5 years, the amount of logic and routing available within a single FPGA has increased dramatically. The Xilinx Virtex family of FPGAs has been available for over 6 years now. The largest device in the original Virtex family of devices (XC6V1000 [3]) released in 1999 had about 1200 4-input lookup tables (LUTs), 1200 Flip Flops (FFs) and 16 Kbytes of on chip RAM (BRAM) running at 200 MHz. The largest device in the current Virtex5 family of devices (XC5VLX330 [4]) released in 2007 has 125,000 6 input LUTs, 125,000 FFs and 32,000 Kbytes of BRAM running at 550MHz. This is about an order of magnitude increase in device density and 250% faster execution speed in 7 years. Many operations that were not feasible in 1999 due to device size are now possible in this new generation of Virtex devices. This change, along with the creation of platforms containing multiple FPGAs addresses the first obstacle noted above.

The CHiMPS (Compiling High-level languages Into Massively Pipelined Systems) toolset has been developed to address the second and third obstacles: the programming model and conceptual acceptance. The main goal of this toolset is to create a programming model whose central tenant is ease of use for software (not hardware) developers and programmers. CHiMPS will allow software engineers to harness the power of FHPC without requiring knowledge of the underlying details of the hardware. This toolset is not intended to replace those applications or designers that understand these details and are proficient in exploiting them to attain speedups far beyond those that can realized with CHiMPS, but rather it is intended to open up the world of FHPC to developers to whom this world is currently a dark, unknown and unexploited mystery.

A number of other commercial companies have attempted to address this market by providing a simplified programming model for FHPC [5,6]. The CHiMPS system differs from these endeavors in its strict adherence to the goal of providing an ease-of-use model familiar to software programmers for FHPC. These other toolsets provide access to FHPC to programmers but at the expense of a programming model different from the one familiar to software programmers. These tools provide extended access into FPGA logic and resources but require the programmer understand the details of the FPGA hardware, which is outside the skill set for most HPC engineers. Concepts such as clock domains, explicit memory management (copy from point A to point B before access); parallel execution (run these three statements in parallel and then the run the next one) or foreign programming language constructs (forall) are not software programmer idioms. HPC software developers want to solve problems in their own domain of expertise (e.g. weather prediction, molecular dynamics, financial modeling) using familiar programming tools, models, and languages and are generally not interested in learning the details of a new hardware system. The CHiMPS toolset attempts to address this by providing a programming model and language that is completely familiar to the software programmer that folds into current computational models so that programming for FHPC is as straightforward as programming for current HPC. More importantly, CHiMPS allows software engineers to develop applications that will run on either conventional processors or FPGAs with only recompilation so they only need code their algorithms once.

2. CHiMPS

The CHiMPS toolset is an instruction level pipelined data-flow execution model of a standard sequential program. CHiMPS does not attempt to implicitly extract parallelism from a given piece of code, but rather relies upon instruction pipelining to attain its speedups. All data is passed through FIFOs to/from specialized instruction execution blocks to maintain local execution argument order. Since reconfigurable logic allows for the simultaneous execution of many instruction operations as long as input data is available, data pipelining can be used to speed up overall execution. In a sequential processor a loop of N iterations that takes C cycles per iteration will complete in N*C cycles. In a pipelined system this same piece of code will take C + N cycles to complete since once the pipeline is filled, an answer can be computed every cycle. As N grows greater than C, the gains to pipelining become obvious. For these reasons, pipelining of instruction execution can be used to dramatically increase overall system performance.
Since the original source code is from a sequential program, explicit synchronization controls are automatically inserted to control instruction execution order when needed (e.g. to preserve proper load-store ordering). All other instructions are executed as soon as their data is available. In this manner, a programmer thinking and writing in a sequential language will be able to access the benefits of FHPC without the need to recode their algorithms.

Finally, generalized hierarchical memory structures of local caches are created using local FPGA L1 memory (BRAM) along with external shared memory to increase the memory throughput for a given algorithm. Oftentimes HPC applications are limited only by their ability to get to/from memory. Multiple local caches provide the ability for simultaneous memory accesses to cached data that is not possible in standard processor based architectures. Allowing the tools and not the programmer to be responsible for this memory movement will again be of great appeal to the users of CHiMPS.

CHiMPS is not a set of standard libraries of FPGA HPC based routines such as BLAS [7] or FFTs [8]. While this would greatly help initial adoption, it would quickly fall short. The reason is that the majority of existing HPC use is not simply stringing prewritten routines together but rather in adding their own customization or ‘secret sauce’. From financial markets to weather simulation to molecular discovery, few organizations would find libraries of pre-canned functions sufficient for more than a few months. CHiMPS has been designed to allow the HPC programmer working solely in their High Level Language (HLL) of choice to compile and execute programs for FHPC. Since most applications that require HPC are large and the resources on an FPGA based system (even multiple FPGAs) are finite, code is split to be run within a mixed processor and FPGA fabric based system. Subroutines, the current level of execution granularity can be easily moved between the two execution environments with directives and does not require other HLL modifications. What few modifications are required at the application source code level are similar to those currently required to execute the code on a new type of sequential von Neumann type processor. These modifications are certainly less extensive than those that are required to take advantage of multiple threads or exploit more than cursory performance gains available in multi-core processors.

The CHiMPS system currently consists of an HLL compiler (initially C language support followed by FORTRAN), an intermediate CHiMPS translation language (CTL), an integrated simulator, a hardware generator and the existing suite of FPGA implementation tools. Partitioning of code between processor and FPGA fabric is currently done with pragmas (or #defines) in the code such that the original source code can continue to be compiled and executed on processor-only based systems. The FPGA fabric targeted code is passed through the CHiMPS compiler to generate the CTL output. The processor-targeted code is compiled with any existing compiler for the system processor with subroutine stubs inserted to correctly execute the FPGA fabric subroutines. These stubs, as generated by the CHiMPS system are also responsible for moving memory to/from the FPGA(s) on which a given subroutine is executed. Currently the types of memory that are shared between processor and FPGA targeted subroutines need to be explicitly annotated (with a pragma/define) but future versions will be able to figure this out without programmer intervention. The CTL of CHiMPS instructions can be interactively simulated in conjunction with the processor-targeted functions to give the programmer a view of the execution speedup, memory usage, resources required and accuracy of results. All of the familiar concepts of pointer based memory access, double precision floating point, full looping support and structure referencing are provided.

Specific optimizations or tailoring of memory access patterns via local cache layout can be made to improve locality of reference and overall memory access throughput. Similarly sharing of resources on the FPGA fabric can be made to decrease the amount of required resources. These are both performance optimizations that are not required for initial execution of the code within the FHPC environment. Other performance optimizations similar to those used on processor-based systems (such as loop unrolling, common sub-expression elimination, loop
reordering etc.) can be made if desired but are not required for execution. Future versions of the compiler should be able to figure many of these optimizations.

Once the programmer is satisfied with the current partitioned code the hardware generator is used to generate VHDL code for the FPGA fabric targeted functions. The VHDL code is then synthesized, placed and routed with the standard FPGA implementation tools to create FPGA configuration bitstreams. Access to these bitstreams and their instantiated code is then tied together with the processor-based software to create the full running application. A later phase of this project is intended to bypass the generation of VHDL and directly generate the required bitstreams. This type of operation is possible due to the known structure of the input circuit as described in the intermediate CTL format. This later phase will address the current problem with extremely long implementation times due to vendor compilation to bitstreams.

3. HARDWARE PLATFORMS

The advent of larger FPGA chips does not automatically open up the world of FHPC. A need still exists for the systems or platforms to tie such chips together such that a complete program can be executed. A couple such systems are discussed below.

The Berkeley Emulation Engine 2 (BEE2) board [9] is an FPGA development platform consisting of five Xilinx Virtex 2VP70 FPGAs. The relevant device parameters for BEE2 are listed in Table 1. The platform was initially designed for radio astronomy, but has since been adapted for a wide variety of applications such as processor architectural simulations (RAMP [10]). The five FPGAs are divided into one ‘control’ and four ‘worker BEE’ FPGAs. Each FPGA directly connects to a local bank of up to 4 Gigabytes of DDR2 memory. Each FPGA on the BEE2 board has four 10Gbit XAUI [11] connections available to connect to additional BEE2 boards. This allows systems to scale easily by attaching additional BEE2 boards. There is no embedded processor so FPGA fabric must be used when code is required to run on the processor. This can either be the embedded hard PPC [12] on the V2P70 chips or an instantiated MicroBlaze [13] processor.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>V2P70  (~75,000 LUTs, 755 Kbytes BRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Four 1 Gbyte DDR2 DIMMs per FPGA</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Four 10 Gbit XAUI connectors per FPGA</td>
</tr>
<tr>
<td></td>
<td>100BaseT Ethernet (Control FPGA only)</td>
</tr>
<tr>
<td></td>
<td>RS232 (Control FPGA only)</td>
</tr>
</tbody>
</table>

Table 1: BEE2 Device Parameters

The Accelerated Computing Platform (ACP) [14] is an FPGA acceleration platform that is being jointly developed by Xilinx and Intel. The relevant device parameters for ACP are listed in Table 2. It consists of a ‘system board’ that can be inserted into a Xeon slot in a server platform that communications over the Front Side Bus (FSB) as a peer processor. The current working system consists of a single XC5VLX330 running at 550 MHz. Future versions of the platform will allow for additional FPGAs, local memory and faster communication speeds over the FSB bus. This system differs from the BEE2 in its use of shared processor memory (albeit at a much slower speed) which makes it more ideal for a co-processor model of computation (where the FPGA is used only for small kernel execution similar to FPU chip).

<table>
<thead>
<tr>
<th>FPGA</th>
<th>V5LX330 (125,000 LUTs, 860 Kbytes BRAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Shared with processor</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Front Side Bus (FSB)</td>
</tr>
</tbody>
</table>

Table 2: ACP Device Parameters
Neither of these platforms is completely acceptable as an eventual FHPC computing machine but rather are initial steps to providing such an FHPC compute environment. The BEE2 suffers from older generation FPGAs with minimal logic/routing resources whereas the latter does not provide enough local memory for a non-coprocessor compute environment. Both though provide enough capabilities for an evaluation of CHiMPS working on real hardware. Both are initial instantiations of their vision and newer versions are being actively developed.

4. SOFTWARE & HARDWARE – power consumption

The runtime environment for CHiMPS is implemented as a set of C library functions that provide primitives for communicating with the CHiMPS circuits and the processor. The library includes routines to write inputs, read outputs, write and read memory, start FPGA processing and to query the status of the CHiMPS circuit. These runtime routines are used to generate function-call stub routines that are inserted into the part of an application that is to be executed upon a standard processor. When an application executing on a processor makes a subroutine call to a function that is implemented in the FPGA fabric, a stub routine is instead executed that is responsible for communicating with the CHiMPS runtime executing on a specific control FPGA. This runtime is responsible for initializing the different FPGAs used for the computation, transferring arguments/memory (if needed) to appropriate FPGAs, collecting results upon completion and transferring arguments/memory back to the calling processor based application. The part of the application running on the processor (referred to as the ‘software portion’) generally has to do with initialization, disk file access, farming out of data to different execution nodes and collection/correlation/output of results. Since these operations are performed a minimal number of times (usually once) and are often slow due to off-system communication needs (read/write files), they are not considered as candidates for implementation within the FPGA circuitry. The part of the application running on the FPGAs (referred to as the ‘hardware portion’) generally consists of multiply nested loops performing some set of computation intensive arithmetic operations. Support of a given platform is not built into the CHiMPS tools but rather is indicated as a configuration file provided during the compilation.

A standard FPGA will consume anywhere from 15-25 watts during execution whereas modern processors consume anywhere from 125-300 watts. The power savings realized from FHPC has to do with replacing N > 1 processors with one processor and some number of FPGAs to obtain greater performance. Some applications will need only minimal processor support so they could easily be implemented within a BEE2 type platform with a slower though power minimized FPGA processor. Other applications will require that some portion of their computation still occur on a standard processor due to the sheer amount of code. These will be more amenable to a platform like the ACP that provides a fast processor. Oftentimes the time critical code that is targeted to the FPGA fabric can be instantiated multiple times to further increase performance and further decrease the need for additional processors. Other platforms that combine the advantages of power savings and faster memory access are being examined in detail as well. Future versions of the system will work to remove the requirement of a standard processor to execute this code due to its inherent large power need.

5. KERNEL SIMULATION RESULTS

The CHiMPS system consists of an integrated simulator that can be used to verify execution correctness, resource requirements and timing. Insertion of pragmas or defines into the HLL allows for the tailoring of FPGA based code for either implementation (what goes onto the FPGA) or optimization. An example of an optimization pragma would be to explicitly state that two pointers are guaranteed to be non-overlapping (similar to the ‘restrict’ keyword in C99). Providing this information allows the CHiMPS compiler to implement these pointers such that the caches they reference are distinct internal memory caches thereby allowing for parallel and non-
conflicting operation. Reads from one pointer will not evict data needed by the other pointer. Since the use of this information will change the operation of the application, a method of verification is needed. To this end, the CHiMPS simulator is used to provide a cycle accurate simulation of the full application running between the processor and FPGA. A full application simulation environment behaves in a manner similar to that described above of inserting stub routines for the hardware portion of the application. The hardware portions are then run directly from the generated CTL information in the CHiMPS simulator providing timing details and resource estimation. Since the CHiMPS simulator is itself processor based it can be used to more quickly estimate the performance, resource usage and accuracy of a given application.

The CHiMPS simulator can also be used in a standalone mode to provide the execution times of a given 'kernel' that is to be placed into FPGA circuitry. Table 3 lists a number of kernels that have been run through the simulator and their speed-up over a standard processor.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>CPU speed up</th>
<th>Area consumed</th>
<th>GOPs/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith Waterman</td>
<td>7.2x</td>
<td>9%</td>
<td>28.8</td>
</tr>
<tr>
<td>Monte-Carlo (Black Scholes)</td>
<td>24x</td>
<td>8%</td>
<td>162</td>
</tr>
<tr>
<td>FFT</td>
<td>4.4x</td>
<td>67%</td>
<td>26.4</td>
</tr>
<tr>
<td>DGEEMM*</td>
<td>0.12x</td>
<td>7%</td>
<td>0.5</td>
</tr>
<tr>
<td>DGEEMM*</td>
<td>2.8x</td>
<td>8%</td>
<td>12.6</td>
</tr>
<tr>
<td>AES encryption</td>
<td>5x</td>
<td>40%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Kernel execution speedups

1. 2.6Ghz Xeon w/ 4MB cache versus 150MHz Virtex2p FPGA
2. Percentage of area consumed by single kernel – multiple kernel could be implemented
3. Giga Operations per Watt, assuming a power reduction of 0.17 for FPGA implementation
4. FFT used in SWM application below
5. Downloaded version with no memory optimization
6. Memory optimizations with pointer restrict

These results were generated solely with the use of the CHiMPS simulator whose execution is much faster than implementing the code directly within the hardware. A number of these kernels were then implemented on the BEE2 board and the performance measured. The CHiMPS simulator results were found to be extremely accurate even though they could not fully model the off-chip memory accesses. This correlation has led to refinements within the CHiMPS simulator so that it more closely mirrors execution on a given hardware platform. As more platforms become available with more amiable memory access systems the CHiMPS simulator accuracy will be increased, but its current behavior has been verified as correct.

6. APPLICATION RESULTS
6.1 SWSTM:
Simulation of kernels that can be wholly run within an FPGA is an indicator of full application performance but does not fully model the final system behavior and performance. Thus, a representative application was chosen to fully implement in hardware to more accurately model full system performance. While any specific application can be argued to be non-representative of other applications, it was important to model something that was in use in the world.

The Spectral Transform Shallow Water Model (STSWM [15]) software program provided by Oak Ridge National Laboratory was chosen for implementation on the BEE2 board to demonstrate the capabilities of CHiMPS. This model is used within current weather forecasting and weather
prediction systems. The STSWM code is quite general and serves as a kernel for both oceanic and atmospheric models. STSWM calculates the fluid behavior of a simplified atmosphere by solving the nonlinear shallow water equations on a rotating sphere using the spectral transform method. During each timestep, state variables (longitude-latitude-vertical grid tensor product) are transformed to spectral coefficients used to solve the differential equation. Key to this transformation is fast Fourier transforms (FFT) for each latitude integrated via Gaussian quadrature to obtain spectral coefficients. This is followed by an equivalent in reverse transformation via inverse real FFTs. Over 90% of STSWM, computations are concentrated in FFT computations that can be performed in parallel.

The salient points of this program that made it amenable for our initial implementation was its use of single precision floating point calculations (double precision is supported but consumes more resources), its obvious computational bottlenecks, its memory access patterns and it is readily available for non-controlled access as well as being in current usage. Several profiling tools were used at Oak Ridge National Laboratory and Xilinx to determine that the computational hot spot FFT calculations were ideal candidates to accelerate on FPGAs.

6.2 Porting and Partitioning:
The initial STSWM code, written in FORTRAN was tailored with compile-time parameters to run one of a number of related algorithms. For our demonstration purposes, we decided to implement only one of these algorithms. We choose FTRNEX, an explicit time stepping algorithm that consists of the following basic functions:

1) Initialization of local memory
2) Reading input data from input files
3) Simple processing of input data into local memory
4) Plotting/outputting of initial conditions
5) Loop through some number of ‘steps’ to process data
   a) Further distribution of input data according to current step
   b) Perform 3 modified FFT on the data for that step
   c) Post-process data for that step
6) Final processing of all data upon completion of all steps
7) Plotting/outputting of final conditions
8) Writing data to output file

Based on performance profiling data, the most time-consuming function, an instance of the modified FFT, was ported to a single worker BEE while the rest of the algorithm was run on the control processor. On BEE2, this processor is the embedded PowerPC running at 300 MHz and offers a significant slowdown over current conventional Opteron or Xeon processors. Since only a portion of the program was targeted to be executed in the FPGA fabric, and since the BEE2 does not support direct access of worker BEE DRAM by the control processor (or vice-versa), it was required to explicitly move data between the processor and the FPGA. These restrictions are not inherent to the CHiMPS system, or the STSWM code but rather are related to the execution platform (BEE2) available.

Since the initial STSWM code was written in FORTRAN and CHiMPS only currently supports a ‘C’ compiler, an initial port was required. A first attempt was made to use an automated tool (f2c [16]) for this conversion but the results generated from the executed program were incorrect and the output ‘C’ code was unintelligible. A manual port of this code was then made and the results were verified. The first thing noted was that generated results between the FORTRAN and C versions did not agree 100% due to the order of floating point operations. This disagreement was nothing new to those familiar with porting floating-point applications between processors but indicated one of the difficulties faced when trying to confirm correctness. The ported ‘C’ code was then annotated with pragmas/defines and compiled with CHiMPS for implementation within the BEE2 board. Table 4 lists the sizes of the code and compilation times:
<table>
<thead>
<tr>
<th>Total lines of C code</th>
<th>1481</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of C code in FFT for FPGA execution</td>
<td>390</td>
</tr>
<tr>
<td>Time to compile app for process/CTL simulation</td>
<td>2 seconds</td>
</tr>
<tr>
<td>Time to process testdata on processor</td>
<td>3 seconds</td>
</tr>
<tr>
<td>Time to process testdata with CHiMPS simulator</td>
<td>43 seconds</td>
</tr>
</tbody>
</table>

Table 4: STSWM compilation/execution statistics

6.3: More problems
Compilation of this code through the CHiMPS compiler and hardware generator created a 23,000-line VHDL file. The resource estimate generated by the CHiMPS simulator indicated that this circuit would consume about 35,000 LUTs. A compilation of this VHDL file through the Xilinx ISE 8.1i FPGA implementation tools verified that the CHiMPS simulator estimates were accurate. The target platform (BEE2) contained Xilinx 2vp70 FPGAs and each FPGA contained only 32,000 LUTs. The initial version was targeting the use of a single FPGA for hardware portion implementation so there was a new problem, use of more resources than supported on a single FPGA. At that time, there were two choices available:

1. Split the circuit across multi FPGAs
2. Try to ‘shrink’ the generated circuitry to fit into a single FPGA.

At that time, the BEE2 platform did not offer any standard interfaces for FPGA to FPGA communication and the FPGA specific DRAMs were only accessible to the control FPGA and not symmetrically between the worker FPGAs. Combining this with a lack of resources for development of the FPGA IP that was needed, a decision was made to begin to ‘shrink’ the code to fit into a single 2vp70 FPGA.

6.4: Shrinkage of generated circuitry
The first attempt to shrink the generated circuitry was to modify the ‘C’ source code to remove any previously unnoted common subexpressions. It is common, and not very expensive in ‘C’ code to perform the same operation a number of times in different places. An example of this would be to access a variable in a given array and index at two separate locations such as

A = x[i];
.....
B = x[i];

Common subexpression elimination could be performed to compute the pointer x[i] once and use that in both locations. The current CHiMPS compiler does virtually no optimizations of this nature so manual modifications of the source code were required. A number of other expression computations were also located and eliminated through modifications to the source ‘C’ code. Though these modifications resulted in some savings in the resources needed, they were extremely time consuming to perform/verify and resulted in code much different from the original. The original tenant of unmodified source code was getting further away.

A second attempt to shrink the generated circuitry consisted of rewriting the instruction block models to consume fewer resources. The endeavor was not considered worthwhile due to the older generation of FPGAs used and a lack of human resources available at that time.

The final set of changes purposed to reduce the footprint required was to ‘share’ expensive resources. The default operation of CHiMPS is to ‘inline’ or make an explicit copy of every instruction to gain full pipelining. A sequence of 10 floating point multiplies would generate 10 copies of the floating point multiply instruction connected up to the appropriate inputs/outputs. Since a floating point multiply consumes many resources, it would more advantageous from a resource usage point of view to ‘share’ these by generating fewer of them that could be mux’d
together. The downside of the sharing is that it will break the pipelining of certain operations to slow the final circuit and, if not properly protected by the tools, can lead to erroneous results. CHiMPS uses the insertion of pragmas or defines to indicate the sharing of certain instructions. It was quickly realized that the knowledge required to correctly share resources of this type was extensive and not something that an HPC programmer would want to perform. Again, the tenet of unmodified source code was moving further away due to platform limitations.

6.5: Defeat
As these shrinkage modifications (common sub expression elimination, block optimizations and sharing) were progressing it began to become obvious that the original vision of making this easy for the programmer was slipping away. Further distressing was the fact that these modifications were not being made because of limitations in the CHiMPS toolset but rather in the choice of available FHPC platform for execution. Though these changes could have performed to get the application to execute, the reasons for doing so were less than compelling. Combined with the expect poor performance due to slower processor execution and memory transfer times, it was decided to stop work on the application porting until a more suitable FHPC platform could be found. Until such a platform could be found to be available, work was begun on a more fully optimizing compiler, new feedback paths to automate the sharing of resources and automatic multi-FPGA partitioning. This work is ongoing at this time.

7. COMPILATION
The current CHiMPS compiler is less than optimal and performs few optimizations that are available in current modern compilers. This was initially required because many of the optimizations of current compilers are targeted to processor-based systems and often the information needed by CHiMPS compilation is thrown away as not relevant. The initial CHiMPS compiler was created to allow this information to exist throughout the process but this oftentimes resulted in the loss of features and optimizations. CHiMPS compiler modifications are being worked on to address these issues at this time.

The current CHiMPS compiler required hand modification of the input source code for execution within the CHiMPS system. Since the driving vision of the CHiMPS system is ease of use, the set of modifications are of interest to the reader. The following set of changes was made to the source code and, other than the first one listed are mostly viewed as things that any optimizing compiler could easily support.

- Translation into to C. The initial compiler is for ‘C’ code only though it is envisioned that a number of the initially targeted applications will be written in FORTRAN. To this end, the CHiMPS compiler is written to handle multiple front ends for compilation from different input source languages.
- Common subexpression elimination. Since some portion of the code was to be directly implemented in the FPGA fabric, oftentimes-dramatic minimization of the generated circuitry could be realized by removal of common subexpressions. All modern compilers perform this optimization while our initial C compiler had not implemented this phase of optimization. This type of optimization is expected to be performed by the compiler by all modern HPC programmers since it can dramatically increase performance (E.G. only do 1 floating point multiply before a loop and save the result rather that recomputing the multiply every iteration of the loop).
- Removing previously implemented performance modifications. While this is not something that a compiler can be expected to perform, it undoes what had previously been done when porting the code from one processing environment to another. For example, the STSWM program had two versions of a specific subroutine available. One was called when one of the input arguments was the 1.0, the other called in all other cases. The first version of the subroutine did not perform an initial floating point multiply within an innerloop (since the argument was 1.0) whereas the second version did perform the operation. Within fully instruction level pipelined code generated by CHiMPS, the

Page 9 of 11
execution time difference of the two subroutines would be unnoticeable at the expense of extra-added circuitry. The modification here was to return the code to its initial version where the floating-point multiplication was always performed. This was effectively returning the code to form in which it was originally written before it was ‘optimized’ to perform in a processor based sequential execution model where there is a significant performance penalty for performing an extraneous floating point operations within a loop.

8. CONCLUSIONS
The chosen application (STSWM) has not yet been run in a full FPGA execution environment from which to examine performance speedups or power diminishment. In this sense, the data is not there for critical evaluation. On the other hand, the development of the CHiMPS toolset and the lessons learned have dramatically moved FHPC closer to average software developer. As new FHPC platforms become available, and as the CHiMPS toolset continues to mature a new alternative to high performance computing has begun to emerge.

The novelty of these results does not lie in the estimated overall execution speed (2-10X faster) or in the estimated diminishment of the power consumed (25-75% less). The novelty lies within the ease with which source HPC applications can be converted to run within an FHPC based system. Once FHPC is made available to the software engineer (and scientist, researcher, analyst ..) of today in a simplified manner, the door is opened for further optimizations and changes to more fully take advantage of non-processor based computing. These changes might consist of rewriting code into other ‘parallel’ based languages; using hand generated high performance IP for common operations or building new methods of distributing code across computation nodes of a different granularity and functionality than those of today. It will probably consist of things that are not even considered at this point of time of solely processor-based computation.

9. FUTURE RESEARCH
The BEE2 platform was chosen as the initial target simply because it was one of the few multi-FPGA platforms available. In the process of our implementation, we discovered that the memory access was severely hampered by non-direct FPGA to FPGA memory access. We were also limited to using the existing set of software tools for the control FPGA and worker FPGA access. Finally, the FPGAs on this board are of a very small size and two generations prior to those currently available. Other multi-FPGA platforms (such as the ACP) were not available at that time. Use of existing or soon to be release FHPC platforms continues to be an area of future research. Finally, the generation of a FHPC based platform that could more fully exploit the benefits of enhanced memory access and pipelining should be considered.

There are a number of possibilities for future research related to the CHiMPS toolset itself. The initial implementation attempted to use only a single small FPGA in the platform for application implementation. This proved to be detrimental to getting a real world application running in hardware. New tools for partitioning across multiple FPGAs need to be designed and developed. One of the initial obstacles that will need to be overcome concerns the issue of multiple replicated memories. Development of a tool that aids the programmer in identifying and manipulating memory access problems, code targeted for FPGA fabric implementation and extraneous data dependencies that can slow pipelined execution needs to be examined. All of these additions would enhance the programmer’s ability to take advantage of the multiple FPGAs within a system for faster processing.

Finally, the complete packaging of these tools for release to the general programmer is needed. Currently these tools are available only within Xilinx Research Labs and are not generally available. The CHiMPS compilation and implementation system will give the developers and
programmer the tools necessary to take advantage of FPGA acceleration offered by the FHPC platforms that are now starting to be available.

10. REFERENCES

[3] Virtex FPGA Data Sheet, ds003-1 (v2.5), Xilinx, April 2, 2001