A POWER EFFICIENT LINEAR EQUATION SOLVER ON A MULTI-FPGA ACCELERATOR

Arvind Sudarsanam*, Thomas Hauser+, Aravind Dasu*, Seth Young*

* Department of Electrical and Computer Engineering
+ Department of Mechanical and Aerospace Engineering
Utah State University
Old Main Hill, UMC-4120
Logan, UT-84322, USA
arvind.sudarsanam@aggiemail.usu.edu; thomas.hauser@usu.edu;
dasu@engineering.usu.edu; sey@cc.usu.edu

Index Terms
FPGA, Linear Algebra, Right hand side solver

Abstract

This paper presents an approach to explore a commercial multi-FPGA system as high performance accelerator and the problem of solving a LU decomposed linear system of equations using forward and back substitution is addressed. Block-based Right-Hand-Side solver algorithm is described and a novel data flow and memory architectures that can support arbitrary data types, block sizes, and matrix sizes is proposed. These architectures have been implemented on a multi-FPGA system. Capabilities of the accelerator system are pushed to its limits by implementing the problem for double precision complex floating-point data. Detailed timing data is presented and augmented with data from a performance model proposed in this paper. Performance of the accelerator system is evaluated against that of a state of the art low power Beowulf cluster node running an optimized LAPACK implementation. Both systems are compared using the power efficiency (Performance/Watt) metric. FPGA system is about eleven times more power efficient then the compute node of a cluster.
I. INTRODUCTION

In recent years Field Programmable Gate Array (FPGA) based high performance computing systems [1][2] have gained attention due to their unique ability to support customized circuits for accelerating compute intensive applications. Research groups worldwide have explored the use of single and multi-FPGA systems for a variety of scientific computing problems such as linear algebra [3-6] and molecular dynamics [3][4]. While using single FPGA systems is well understood, at least by the digital circuit design community, multi-FPGA systems are still being explored. The complexity of using such systems is very high and well beyond the reach of non-circuit designers. Particularly their performance with respect to floating-point intensive applications and their power efficiency (performance/watt) advantages are not well understood or explored. In this paper, we present an investigation into the domain of solving a linear system using forward and back substitution [5]-[6] that is floating-point compute intensive, on a commercial multi-FPGA system: The Starbridge Hypercomputer (HC-36). The standard algorithm to solve a linear system of equations is based on the lower and upper triangular matrices that are obtained as a result of a LU factorization of the coefficient matrix. To solve the system for each right hand side (RHS) vector, a forward and back substitution is performed. We present our Viva implementation of the block-based forward and back substitution algorithms on a high-performance reconfigurable computer (HPRC) composed of a conventional host computer and multiple FPGAs. We discuss how multiple memory hierarchies are used to obtain maximum efficiency of the implementation and compare the performance of the HPRC system to the performance of a LAPACK [7] implementation on one node of a low power commodity cluster supercomputer [8]. The HPRC system used for this study is the Starbridge Hypercomputer HC-36 but the algorithm and implementation are portable to many similar systems. The paper is organized as follows: Section II discusses the prior work done towards developing matrix-based implementations for FPGA systems. Section III will describe the hardware platform used. Section IV provides an overview of the significant features of the Viva development tool. Section V will follow up with a description of the algorithm used to solve the factorized linear system. Section VI will discuss the hardware design approach to map the algorithm onto the hardware platform. Section VII describes the performance results obtained and compare these results to a quad core microprocessor and Section VIII concludes the paper.
II. LITERATURE REVIEW

Hardware-based matrix operator implementations have been studied by many researchers. Ahmed-El Amawy [9] proposes a systolic array architecture consisting of \((2N^2 - N)\) processing elements which computes the inverse in \(O(N)\) time, where \(N\) is the order of the matrix. However, there are no results to show that the large increase in area (for large values of \(N\)) is compensated by the benefits obtained in speed by this implementation. Lau et al. [10] attempt to find the inverse of sparse, symmetric and positive definite matrices using designs based on Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) architectures. This method is limited to a very specific sub-set of matrices and not applicable for a generic matrix and hence has limited practical utility. Edman and Owall [11] also targeted only triangular matrices. Zhuo and Prasanna [12] propose a methodology to implement various linear algebra algorithms on a particular FPGA device. This paper also proposes a performance prediction model that incorporates (a) implementation parameters - number of data paths, memory required etc. (b) resources available on the target FPGA and (c) algorithm parameters - block size, matrix size, etc and uses this model to predict the set of parameters that results in the best performance. However, this paper has limited discussion on multi-FPGA based implementation and does not discuss the logic required for off-chip to on-chip data transfer. Choi and Prasanna [13] implement LU decomposition on Xilinx Virtex II FPGAs (XC2V1500), using a systolic array architecture consisting of 8/16 processing units. This work is extended to inversion and supports 16-bit fixed-point operations. Hauser, Dasu et al. [14] presented an implementation of the LU decomposition algorithm for double precision complex numbers on the same star topology based multi-FPGA platform as used in this paper. The out of core implementation moves data through multiple levels of a hierarchical memory system (hard disk, DDR SDRAMs and FPGA block RAMS) using completely pipelined data paths in all steps of the algorithm. Detailed performance numbers for all phases of the algorithm are presented and compared to a highly optimized implementation for a low power microprocessor based system. Additionally, they compare the power efficiency of the FPGA and the microprocessor system. Vikash and Prasanna [15] propose a single and double precision floating point LU decomposition implementation based on a systolic array architecture described in [13]. The systolic array architecture is a highly parallel realization and requires only a limited communication bandwidth. However, every element in the systolic array needs to have local memory and a control unit in addition to a computation unit, which adds significant overhead. Wang and Ziavras [16] propose a novel algorithm to compute the LU decomposition for sparse matrices. This algorithm partitions the matrix
into smaller parts and computes the LU decomposition for each of them. The algorithm to combine the results makes use of the fact that most of the sub-blocks of the matrix would be zero blocks. However, this method cannot be extended to find LU decomposition for dense matrices. Research efforts towards parallel implementations of LU decomposition largely deal with sparse linear systems. In some cases, these implementations make use of a software package called SuperLU_DIST, which may be run on parallel distributed memory platforms [17],[18]. Other work using similar software package routines are found in [19]. A common platform that has been used for sparse matrix systems involving LU factorizations is the hypercube [20],[21]. Other implementations involving parallel LU linear system factorization and solutions may be found in [22],[23],[24],[25]. As the number of logic elements available on FPGAs increase, FPGA based platforms are becoming more popular for use with linear algebra operations [26],[27],[28]. FPGA platforms offer either a distributed memory system or a shared memory system with large amounts of design flexibility. One such design, presented in [28], utilizes FPGA based architecture with the goal of minimizing power requirements. Any application implemented on an FPGA that uses external memory must provide some means of controlling the memory structure to store/access memory in an efficient manner. A common application that requires control of external memory is image processing. One group from Braunschweig, Germany has designed an SDRAM controller for a high-end image processor. This controller provides fixed address pattern access for stream applications and random address pattern access for events like a cache miss [29]. Another image processing application being worked on by a group from Tsinghua University in Beijing utilizes a memory controller specifically designed to reduce the latency associated with random access of off chip memory [30]. A design made to handle multiple streams of data was made by a group from the University of Southern California and the Information Sciences Institute. In this design, each port in the data path as a FIFO queue attached to it. These data paths are also bound to an address generation unit used to generate a stream of consecutive addresses for the data stream [31]. The design presented in this paper is similar to the above-mentioned work in the fact that it must both fetch and write data to an external memory device. However, in terms of complexity, the design in this paper is much simpler in that it provides specific streams of data at specific times for the LU processing engine. In such light, it is not very flexible. However, simplicity has worked to the advantage that the design is easily replicated across multiple processing nodes. Another advantage that comes with simplicity is the low resource count the memory controllers take roughly 13% of the available FPGA slices. This leaves much more room for the LU processing engine than a more complex design would.
III. THE STARBRIDGE HYPERCOMPUTER HC-36

![Diagram of the Starbridge Hypercomputer HC-36]

The target hardware system that this design is implemented on is a distributed memory FPGA architecture called a hypercomputer (HC-36) built by Starbridge Systems Inc [32]. The architecture consists of a main controller (an Intel Xeon based PC), which is connected to several independent FPGA processing nodes (called processing elements, or PEs) through a common PCI-X bus. Such a network topology is termed "star topology" and is illustrated in Figure 1. The FPGAs are Xilinx Virtex II 6000 devices, each of which has an independent off-chip DRAM (dynamic random access memory) of 2GB. Each FPGA consists of 33,792 slices, 144 Block RAMs (324 Kb in total), and 144 18x18 hardware multipliers. The HC-36 is connected to a host processor (Intel Xeon with a clock speed 3.8GHz) via a PCI-X bus consisting of 64 bi-directional wires capable of operating at a maximum speed of 66 MHz. All of the FPGA hardware design was done using a structural HDL (hardware description language) called Viva provided by Starbridge Systems Inc [33]. FPGA mapping, Place-and-Route, and bitstream generation are performed using Xilinx 10.1 tools [34].

IV. OVERVIEW OF VIVA

Viva is the graphics-based hardware design tool used to develop the FPGA-based designs for the RHS solver. It consists of a graphics-based development environment, a C++ based design analysis, and a place and route tool at the front-end that uses the Xilinx tools to synthesize the design and then place and route the design for the target FPGA platform. Viva is supported by a large library of computation, memory, control, and I/O objects. Most of these objects are polymorphic and completely pipelined to operate with a throughput of one data unit per time unit. Viva also provides a methodology to design new polymorphic objects using a recursive technique. Required computation is broken down recursively.
into multiple copies of a basic implementation, known as the leaf implementation. The depth of recursion will depend on the input data. Many of the computation units used in matrix-based operations are already available in Viva with data type polymorphism incorporated. Viva provides support for a new data type called 'List' which is a collection of data points and all the computation and memory units are designed to operate on any given List size and data type. This feature facilitates the task of designing polymorphic hardware. Additional circuitry is required to support order tensor polymorphism.

Polymorphism is a trait of any hardware/software design that enables the user to generate the best-suited working model for multiple input types and design specifications. For instance, in the proposed block-based RHS solver, the algorithm will vary based on the size of the matrix and the block size. Polymorphic hardware circuits will be able to morph themselves based on the input specifications, and thus reduce overall design time. This section explores the concept of how the polymorphic design approaches in Viva are applied to our RHS solver. The three different types of polymorphism used in our design are listed below.

- Data type polymorphism
- Information rate polymorphism
- Order tensor polymorphism

Following sub-sections discuss these three types in detail.

A. Data type polymorphism

\[
\begin{align*}
\text{Sum}[\text{Low}] &= A[\text{Low}] + B[\text{Low}] \\
\text{Sum}[\text{High}] &= A[\text{High}] + B[\text{High}] + \text{Cout}[\text{Low}] \\
\text{Cout} &= \text{Cout}[\text{High}]
\end{align*}
\]

Fig. 2. Data type polymorphism illustrated using an 8-bit adder

Data type polymorphism enables inputs and outputs for a polymorphic functional unit to be decided during compile time (not during design time). This reduces the design time by reusing the same circuit
for any data type. A recursive technique is provided by Viva to design data type polymorphic objects. To explain this concept, implementation of a simple 8-bit ripple carry adder is illustrated in Figure 2. As shown in the figure, an N-bit ripple carry adder can be broken down into two N/2 bit ripple carry adders, with the carry-out of the lower adder acting as the carry-in of the higher adder. The N/2 adder can be further broken down into two N/4 adders and this break-down process will lead to single bit adders. Since Viva can synthesize objects based on the input types, this recursive implementation is possible. A major challenge for a hardware designer is that he will need to develop a recursive algorithm that will perform the required computation.

In polymorphic implementations, there is always an overhead due to the logic supporting the polymorphic implementation. To analyze this overhead, the polymorphic circuit was compared with a fixed 8-bit adder circuit (which is also implemented as a ripple carry circuit) that was developed using gates. It was observed that there was an overhead of 14 slices due to the use of polymorphic adder. An example of a polymorphic multiplier in Viva is shown in Figure 3. This figure demonstrates the technique used to select the data types of the ports A and B of the multiplier during compile-time using the user interface (shown on the right).

B. Information rate polymorphism

Information rate polymorphism is defined as the amount of data that is processed by the computation unit in every clock cycle. This data is limited by two factors: (i) The amount of I/O bandwidth associated with
the computation unit, and (ii) the amount of hardware real estate available to implement the computation unit. Information rate polymorphism is denoted by the parameter 'K' (Information rate factor), that specifies the number of data points consumed by the computation unit in one clock cycle.

An example designed in Viva is shown in Figure 4. In this figure, two implementations are shown in Figures 4(a) and 4(b) respectively. As seen in the figure, Viva provides a specific data type called 'List'. This object is a compile time entity, and does not incur any run-time overhead. A 'List' is an abstract representation of wires grouped together. The polymorphic circuit design will need to have additional information to break down this 'List' into individual wires before feeding them into the various parallel units. The two multiplier units used in the implementations shown in Figure 4 are equivalent objects in the library of arithmetic objects. Based on the number of input values in the 'List', the required number of parallel multiplier objects is generated by the compiler. In Figure 4(a), four parallel multipliers are generated and in Figure 4(b), 8 parallel multipliers are generated. This illustrates information rate polymorphism.

C. Order tensor polymorphism

Order tensor polymorphism enables the hardware designer to specify the order of the matrix to be processed during run-time. The minimum order supported is the value 'K' (Information rate factor) and the architecture will be reused to support any order greater than that. The control unit takes care of scheduling this repetitive execution.

An example is illustrated in Figure 5. In this figure, the 'Counter' module is a counter that generates values from 1 to N, where N is an input into the 'Counter' module. The values generated are used as
addresses into the RAM modules. Data read from the RAM modules are used as inputs to the adder unit. This circuit adds a list of $N$ numbers present in the top memory module with the corresponding set of $N$ values in the bottom RAM module and stores the $N$ results in the right-most RAM module. Value of $N$ can be set during run-time or compile-time. This example is a simple illustration, as the addition of two sets of $N$ numbers can be easily broken down into individual addition operations. Complexity in implementing order tensor polymorphism arises when it is not easy to break down $N$ computations into individual computations.

Polymorphic design techniques explained in this section are used to realize the design for the proposed RHS solver for a single FPGA. Details are provided in Section VI.

V. FORWARD AND BACKWARD SUBSTITUTION FOR THE SOLUTION OF A LINEAR SYSTEM

A. Standard Algorithm

In this paper we consider the solution of a linear system $Ax = B$, where $A$ is a full $N \times N$ matrix. The standard way to solve this linear system involves first the LU decomposition, $[L U] = lu(A)$, followed by a forward, $Lz = B$, and backward, $Ux = z$, substitution. In our implementation, the matrix $A$ is assumed diagonally dominant, so no pivoting strategy is necessary and the condition number is of order one. The forward substitution for an element of the intermediate solution $z_j$ can be written as shown in equation 1.
\[ z_j = \left( B_j - \sum_{i=1}^{j-1} (L_{ji} \times z_i) \right) \]  

(1)

The final solution \( x_j \) can be computed as shown in equation 2. Equation 2 is similar to equation 1 except for division by the diagonal element of \( U \) \((U_{jj})\), which is not unity.

\[ x_j = \frac{1}{U_{jj}} \times \left( z_j - \sum_{k=j+1}^{N} (U_{jk} \times x_k) \right) \]  

(2)

In Equations 1 and 2, the main computation is a vector dot product, which has been implemented by Underwood and Hemmert [35]. However, sub-components of the solution have some data dependencies, which need to be considered when designing parallel micro and macro architectures to execute them on a multi-FPGA system. We have adopted a block-based method to expose as much parallelism as possible on the multi-FPGA system without overloading the communication link between the host PC and the multi-FPGA board. The block-based method was specifically chosen to mitigate, to some extent, certain limitations of the HC-36 architecture. Three of the salient features of block-based algorithm is listed below.

1) From Equations 1 and 2 it can be seen that there is no data reuse i.e. the data points in the input matrices and vectors are used only once and then discarded. Trying to directly implement the pseudo-code algorithms on processing nodes would be a problem if the entire data system was too large to fit in the FPGA node’s local memory. The block-based parallel algorithm for the forward and backward substitution overcomes this problem because each FPGA node is only required to hold a subset of the data system.

2) The block-based algorithm nullifies the data dependencies so there is no need for communication between FPGA nodes. This allows more FPGA nodes to be added to the design without needing to change the hardware configuration loaded onto those nodes, increasing the parallelism as the number of resources increase.

3) By splitting the matrix into smaller blocks or vectors, task level parallelism can be achieved by mapping the different blocks onto the multiple FPGA devices on the board.
Fig. 6. (a) Block partitioning of the equation (b) Data dependency for computing sub-vector $Z_3$

B. Block-based parallel algorithm

Figure 6(a) illustrates the proposed partitioning for the system described by the equation $Lz = B$. Here the sub-matrices of $L$ are of size $N \times N$, the sub-vectors of $z$ and $B$ have dimensions $N \times 1$. The blocks that include the diagonal elements of $L$ are diagonal sub-matrices themselves. One step of the computation will be performed on a single block of data from $L$ and the corresponding sub-vector from $B$ to provide a sub-vector of $z$. In addition to the required block and sub-vector for each computation, there is also intermediate data required. However, this additional data is in the form of a vector that will not exceed the size of the sub-vector from $B$.

The parallel algorithm is based on observations of the data needed to calculate each element in the vector $z$. On inspection, it can be seen the blocks of data needed to compute the sub-vector $Z_3$ are $L_3$, $L_6$, $L_8$. In addition, sub-vectors $Z_1$, $Z_2$, and $B_3$ are also required. This data dependency can be seen in Figure 6(b). On closer inspection it can be seen that the computations involving the blocks $L_3$ and $L_6$ along with their corresponding sub-vectors $Z_1$ and $Z_2$ are just a sum of products (matrix-vector multiplication). Based on this idea we can use independent processing nodes to provide the intermediate sum-of-products (SOPs) needed in the computation of sub-vector $Z_3$. The idea of intermediate SOPs can be further seen in Figure 7. Here, the intermediate SOP $Z_4(1)$ is computed as the matrix-vector multiplication of $L_4$ and $Z_1$ (shown in Figure 7(a)). In order to compute the next intermediate SOP the product of $L_7$ (Block) and $Z_2$ (Top) is added to the previous intermediate SOP $Z_4(1)$ (Left) to produce $Z_4(2)$ (shown in Figure 7(b)). Continuing in this manner intermediate SOP computations may be assigned to independent processing nodes which may do the processing in parallel.

Based on these properties of the algorithm and the features of the target hardware infrastructure, three important design decisions were made:

1) Intermediate SOP computations involving blocks $L_2$, $L_3$, and $L_4$ may be done in parallel. Therefore,
I. Source

on the Starbridge HC-36 system each of these intermediate SOP calculations were assigned to a separate PE.

2) Computations involving the diagonal blocks which produce the final results for the sub-vectors of \( z \) are considerably different from the computations for the intermediate SOPs. In addition to this, the diagonal block computations cannot be done in parallel with any of the other computations. Because of these reasons, the diagonal computations were done on the host PC within the governing C program.

3) Computations performed for the forward substitution are nearly the same as those required for the backward substitution, so the same intermediate SOP process was used in both substitutions.

VI. HARDWARE MACRO/MICRO-ARCHITECTURE DESIGN METHODOLOGY

![Diagram](image)

---

The RHS solver implementation on the HC-36 may be seen as a three-step process, as listed below.

1) Data is transferred from the host PC to a specific target PE.

2) PE processes the data to provide the intermediate SOP.

3) Processed data is sent back to the host PC.
Data transfers to and from the host PC and data processing are controlled locally on each PE by a four-state Finite State Machine (FSM) circuit. The FSM provides control signals to initiate and execute the three main steps of the solver implementation. Once the host PC has loaded the bitstream onto a PE, the FSM is initiated by a start code sent from the host PC, which also contains a constant, required for computation. After the start code has been sent, the PE design is ready to run over continuous iterations processing intermediate SOPs without reprogramming. The top level block diagram for the processing node hardware design can be seen in Figure 8. The design consists of the following circuits:

1) RHS Controller state machine - Sub-section A
2) Sequence detector.
3) Data to BRAM and Data from BRAM controllers.
4) BRAM module - Sub-section B
5) Compute engine process - Sub-section C

In this section, we will only discuss the details of circuits '1, 4 and 5’. The reader is referred to previous publications [36],[37] for details on circuits '2 and 3’. Nevertheless, for the sake of flow is discussion, the following paragraph provides a brief overview of the overall process.

Data from the host PC is made available to the PE through a communication circuit called Data from PCI. Likewise, data is sent back to the host PC through the Data to PCI circuit. The start code from the host PC is detected from the data stream coming from the Data from PCI by the Seq Det module. Upon detecting the start code, the Seq Det module provides an initiation signal to the state machine. Upon reception of the start code from the host PC, the data following the code is known to be valid and is written by the Data to BRAM module (enabled by the FSM) into the BRAM memory of the FPGA. Upon completion of the data read from the host PC, the Process module is initiated. This module implements the matrix-vector multiplication of the Block and Top data sets and adds the result to the Left vector. The result overwrites the section of BRAM where the original Left vector was stored. Once the Process module has finished with the data, the FSM enables the Data from BRAM module, which reads out the computed intermediate SOP value and sends it back to the host PC through the Data to PCI circuit.

A. RHS Controller

RHS controller consists of a FSM controlling the hardware with four states. The flow diagram for the state machine is shown in Figure 9. The FSM is implemented using two flip-flop registers and simple
logic gates to determine the state/next-state transitions. Upon loading the bitstream into an FPGA, the FSM defaults to state zero, or Reset, which is an idle state. The FSM waits here until it receives the initiate signal gleamed from the data coming from the PC by the Seq Det module.

The start code indicates that the data following the code will be valid and needs to be written into the BRAM. Upon receiving this start signal, the FSM transitions from the Reset state to state one, the Fill state. Upon entering the Fill state, the FSM provides an enable signal to the Data to BRAM module. Once all the data from the host PC has been written to the BRAM the Data to BRAM module sends a done signal back to the FSM, which allows the FSM to transition to state 2, the process state. Here FSM provides an enable signal to the Process module. When the Process module has completed its computations, it returns a done signal to the FSM causing it to transition to state 3, the Write Back state. On transition into this state, the FSM sends an enable signal to the Data from BRAM module, which reads the results from the BRAM and sends them back to the host PC through the Data to PCI circuit. When the write-back process has finished, the Data from BRAM module notifies the FSM with a done signal that it is done. The FSM then transitions back to state 1, the fill state, to wait for the next batch of data. In this manner, the FSM controls the PE as it processes data for all iterations in the RHS solver algorithm. Figure 10 shows the Viva implementation of this state machine. One-hot encoding technique is used to realize this state machine. Each state is represented using two bits ($D1$ and $D0$). Given the current state, the circuitry for computation of next state and some output control signals (Write Done, Read Done, Process Done) is shown in Figure 10. It is observed that Viva requires the designer to develop the logic required for state transitions. A feature to automatically derive state machine logic from an abstract state transition diagram can be a useful addition to a future release of Viva.
B. Memory Module

The BRAM module shown in the top level block diagram (Figure 8) consists of a bank of three separate dual port BRAM modules and a grouping of multiplexers to handle the input signals from the other modules of the design. Three separate dual port BRAMs are used to hold the three separate data structures so they may all be accessed simultaneously. A simplified block diagram of the BRAM module is shown in Figure 11.

Three groups of multiplexers are required to route the correct command signals to the three BRAM modules. Data from BRAM module requires access to write commands of each of the three BRAMs. The Process module needs read access to all three BRAMs and write access to the BRAM module holding the Left data set. The Data to BRAM module needs read access to all three BRAM modules. For the cases
where more than one module needs the same kind of access to a BRAM block, the command signals are passed through multiplexers. The multiplexers’ outputs are selected by the FSM according to what state the design is currently running in. The amount of onboard BRAM memory determines the amount of data a single PE can process in a single iteration. Specifically, the amount of BRAM determines the block-size $b$ for the Block, Left, and Top data sets. The Virtex II 6000 FPGAs used as PEs have 144 18Kbit memory blocks available for a total of 324Kb of on-chip memory available. A block size of $b=128$ is the maximum block size chosen for this implementation because it is the largest value of $b$ (even power of 2) that will fit on the available BRAM.
C. Forward/Backward Substitution Compute Engine

Process module is based on a double loop counter, which is used to index the data values stored in the Block, Top, and Left BRAM modules. As described earlier in this paper, the operations to be done in the Process module are a matrix-vector multiplication followed by a vector subtraction. This has been integrated into a multiply-subtraction process. However, if the multiply and subtract operators have a substantial pipeline delay, the operation is slowed considerably. This is because the inputs to the subtract circuit must pass completely through the pipeline and produce a result before it can be subtracted from the next available product of the multiplier. Such an implementation with a pipeline length of $k$ would require $k$ clock cycles for each subtract operation to be computed (the pipeline cannot be filled to provide a higher throughput). This problem was overcome by changing the order in which the multiply and subtract operations are done in the matrix-vector multiplication operation. Top-level block diagram for the Process module shown in Figure 12 consists of the following modules: (I) Double loop counter (ii) Multiply-subtract unit and (iii) Address calculator. Double loop counter is used to provide the index values that are, in turn, used to calculate the addresses to the Block, Top, and Left memory modules. Address calculator takes the two index values from the double loop counter and generates a separate address for each of the three memory blocks holding the data. Data read from the memory module is fed into the Multiply-subtract unit. Through this circuit, the multiplier and subtractor pipelines are kept continuously full and the processing operations can be completed without excess delay.
Figure 13 shows the Viva implementation of the *Process* module that was illustrated in Figure 12. In this implementation, the variable block size \( (b) \) controls the total number of iterations, and can be modified during compile-time and/or run-time. Computational structure of the forward and back substitution algorithm is found to be loop-intensive and thus enables the realization of this order tensor polymorphic design. Figure 14 shows the Viva implementation of the Multiply-subtract unit. InputA, InputB, and InputC are the three different inputs from the Top, Left, and Block data elements respectively.
TABLE I
RESOURCE UTILIZATION FOR DIFFERENT BLOCK SIZES

<table>
<thead>
<tr>
<th>Block size (b)</th>
<th>Slices</th>
<th>BRAM</th>
<th>MULT18x18</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17049</td>
<td>4</td>
<td>36</td>
</tr>
<tr>
<td>32</td>
<td>17049</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>64</td>
<td>17049</td>
<td>34</td>
<td>36</td>
</tr>
<tr>
<td>128</td>
<td>17049</td>
<td>130</td>
<td>36</td>
</tr>
</tbody>
</table>

These inputs can be fixed-point, floating-point, floating-point complex etc (data type polymorphism). Also, the inputs can be a single data element, or a list of data elements (information rate polymorphism). Number of multipliers and adders instantiated during compile-time is equal to the number of data elements specified by the designer. In the proposed implementation, data-type is set to be a double precision floating-point complex number. Given the resource constraints of the target FPGA, number of data elements is limited to one, as only a single double precision floating-point complex multiplier and adder could be fit into this FPGA. Results are presented in the next section for this configuration of the architecture.

VII. RESULTS AND ANALYSIS

We implemented the RHS solver architecture that was discussed in Section VI using Viva 3.0. Resource utilization for different block sizes is shown in Table I. Our implementation was benchmarked on a Starbridge Hypercomputer board HC-36 (see Section III). Although the IP cores for individual arithmetic units (obtained for the Viva coregen library) can be clocked at 100MHz or higher, the vendor caps the board to run only at 66MHz. Our proposed implementation is designed to operate at this board frequency, but can be enhanced further using the digital clock manager (DCM) resident on each FPGA to realize multiple operating frequencies. For our benchmarking, matrix sizes are varied between 1000x1000, 50000x50000, and 100000x100000. Each element in all of the matrices is a 64-bit double precision (52- mantissa; 11-exponent; 1-sign) complex number. Implementations for two different block sizes (32 and 64) are evaluated. Number of FPGAs (nFPGA) is varied between 1, 2, and 4. Timing results are obtained using the clock function provided in a C header function that has a sampling rate of 1 KHz. Wall clock timing results are obtained by measuring the time across the execution of the outermost loop for both forward and backward substitution and include FPGA configuration time. Figure 15 shows the timing results for block size of 32 and 64. An increase in block size leads to a reduction in overall execution time. Larger block size results in a lower number of blocks, and the total overhead that is associated with setting up the processing of each block is reduced. Overall timing results do not scale with the increase in number.
of FPGAs because of the architectural limitations of the star topology of the HC-36. This architecture limits the data transfer between PC and FPGA to be performed over one single PCI-X bus during the distribution of the blocks.

To further explain the above results, a performance prediction model is developed and different factors contributing towards the overall execution time are analyzed. In the discussion of the performance model based results, the following three phases in the execution are differentiated.
• PC to BRAM time: This describes the time taken to transfer data from the host PC to various BRAM resources on the FPGA board
• FPGA processing time: This includes all computational processing on the FPGA
• BRAM to PC time: This is the time taken to transfer the data back to the PC

Performance prediction model proposed in this paper is an analytical model and formulas are derived based on the system parameters of FPGA and board-level details, along with the algorithm based parameters like size of the matrix size, block size etc. Impact of various parameters on the execution time of the three phases will be discussed in the sub-sections below. Following sub-sections discuss the trend in execution time based on variations in matrix size, block size, and number of FPGAs respectively.

A. Performance model based results for variations in matrix size

![Fig. 16. Performance based on matrix size variations](image)

In Figure 16, run-times for several matrix sizes are compared for each of the different phases of the algorithm. Block size \( (b) \) is fixed at 64 and two FPGAs \((nFPGA)\) are used in this analysis. Matrix sizes \((N)\) are varied from 50000x50000 to 200000x200000. In these results, it is observed that the PC to BRAM data transfer time is nearly 4 times the FPGA processing time and the BRAM to PC data transfer time is negligible compared to the execution time of remaining two phases. RHS solver algorithm discussed in this algorithm is a block-based algorithm. For processing a single block, \( b^2 \) 128-bit data elements need to be fetched from PC into the BRAM. PCI-X bus on the HC-36 system used to facilitate this transfer
is a 64-bit bus, and $2b^2$ clock cycles are required to transfer a single block. In comparison, the compute engine architecture is a fully pipelined architecture (but no parallelism, due to lack of resources), and thus requires $b^2$ clock cycles to process a single block. Also, the PCI-X bus is shared between the two FPGAs, whereas the computing resources of the two FPGAs are available for parallel processing. These limitations in the PC to FPGA connectivity result in a considerable time spent towards PC to BRAM data transfer. For a single block of size $b^2$, only $b$ elements are transferred back to the PC, and BRAM to PC data transfer time is found to be negligible. Overall execution time is found to be directly proportional to $N^2$, for a given value of $N$. Total number of elements to be transferred (and processed) is proportional to $N^2$.

B. Performance model based results for variations in block size

In Figure 17, run-times for several block sizes are compared for each of the different phases of the algorithm. Matrix size ($N$) is fixed at 100000 and two FPGAs ($nFPGA$) are used in this analysis. Block size ($b$) is varied between 16, 32, and 64. The trend in comparison between execution times for different phases is similar to the one found in Figure 16. It is observed that variations in block sizes do not impact the overall execution time by more than 10%. FPGA processing time is found to be the same for variations in $b$. Small reductions in the execution time are due to reduction in total number of blocks. Data transfer from PC to FPGA is performed in terms of blocks and each transfer is associated with a set-up time. Overall set-up time decreases with increase in $b$. 

![Figure 17. Performance based on block size variations](image-url)
C. Performance model based results for variations in number of FPGAs

In Figure 18, run-times for several number of FPGAs are compared for each of the different phases of the algorithm. Matrix size \((N)\) is fixed at 100000 and block size \((b)\) is fixed at 64. Number of FPGAs \((n_{FPGA})\) is varied between 1, 2, and 4. From this figure, it is clear that the data transfer times do not vary. FPGA processing time is found to scale with number of FPGAs. But the overall execution time is dominated by PC to DRAM data transfer time and thus the speed-up obtained from increasing the number of FPGAs is amortized.

D. Performance model based results for a Virtex-4 FPGA

To estimate impacts of changes of the FPGA hardware in the hypercomputer (using a newer FPGA with more resources), proposed performance model was modified. This section discusses in short some of the results obtained through the performance model, when using a more advanced FPGA hardware. Our performance model contains a set of system parameters which can be obtained for any FPGA from their respective data-sheets. Hence, our model can be extended to support any multi-FPGA hypercomputer platform. Here we compare the performance model results for a Xilinx Virtex-4 LX160 FPGA with the overall timing results from our benchmark platform using Xilinx Virtex-II 6000 FPGAs. For each target FPGA the number of data paths \((P)\) is set such that the resource utilization of the FPGA is maximum. In this implementation, the data path consists of double precision floating-point complex multiplier and
Fig. 19. Comparison of overall execution time between Virtex-II 6000 and Virtex-4 LX160 FPGA

adder. We found $P = 5$ for the Virtex-4 FPGA and $P = 1$ for the Virtex-II FPGA. In addition, we included the effect of switching from the floating-point objects provided by the Viva Corelib to the Xilinx IP Coregen library [34] in our performance model results. Figure 19 shows comparative results for the two target platforms for different values of the matrix size ($N$). Block size ($b$) is fixed at 32 and two FPGAs ($nFPGA$) are used in this analysis. Matrix sizes ($N$) are varied from 50000x50000 to 200000x200000. It is seen that the Virtex-4 implementation is faster than the Virtex-II implementation, but the speedup is not proportional to the number of data paths ($P$). Using a larger FPGA provides more resources for realizing a larger number of data paths. However, the overall execution time of the RHS solver algorithm is dominated by PC to DRAM data transfer time and thus the speed-up obtained from increasing the size of the FPGA is reduced.

E. Power measurement results and comparison with commodity microprocessor

Power efficiency has recently become an important metric for high performance computing systems. We follow the work of Kamil et. al [38] in defining the workload and power measurement. Generally, the power fluctuates during a benchmark run, but we present the average power usages for our benchmarks similarly to [38]. Power efficiency, defined as the ratio of floating point performance per watt drawn, is a popular metric used to compare systems. It is possible to use this metric for system comparison, as long as systems of similar size are compared [39],[40]. Our power measurements methodology followed one
of the approaches described in [38]. We measured the voltage drop across the hypercomputer board and the amount of current drawn by the board. For comparison purposes we present the average power drawn in Figure 20.

Figure 20 shows that an increase in the matrix size does not increase the power consumption, but an increase in the number of FPGAs increases the power consumption linearly. Best power efficiency for the RHS benchmarks is obtained for the following parameters: \( nFPGA = 1; \ b = 32; \ N = 50000; \) and
is equal to 69 MFlops/Watt. Increasing or decreasing the block size changes the number of BRAMs used in the design, but does not have an impact on power consumption. This can be explained by the observation that the number of data reads and writes of the RHS engine per clock cycle stays constant. For comparison, the power consumption and efficiency was measured on a low power commodity CPU cluster which was specifically designed to run on a single 20A 110V circuit [8]. A compute node of this system consists of one Quad-Core Intel Xeon X3210 processor, 8 Gigabytes of RAM and no disk drives. The processor Xeon X3210 by itself is not a low power processor, but during the design of the cluster we observed that memory can contribute significantly to the power consumption. Our node design provided for the most cores per a single 20A 100V circuit. It was not possible to measure the power consumption of one node independently. Therefore, we measured two identical nodes running the same benchmark, and the resulting power reading was divided by two. Power was measured using a power analyzer called Watts UP. Implementation of RHS solver on the cluster node uses the Intel MKL library (version 9.1) [41], specifically the LAPACK routine 'zgetrs’. This library is highly optimized for Intel processors and provides highly scalable implementations for multiple threads. Time taken for the execution of the RHS solver was 0.345s, 0.342s, 0.343s and 0.343s for one, two, three and four threads for a matrix size of 10000. Power consumption for idle, one, two, three and four threads was measured as 154, 227, 277, 316, and 353 Watts respectively. Resulting power efficiency comparison is shown in Figure 21. We compare one compute node of a cluster computer with the FPGA accelerator. Both systems, the cluster compute node and the FPGA board, need another system (master node or a host system) in order to function properly. However, both systems can be built identically, and hence power consumption of the master node and host system is left out of the comparison. Power consumption is found to be 1159 MFLOPs on the Intel processor compared to 64 MFLOPs for the FPGA-based system. However, the average power consumption is found to be 227 Watts, hence resulting in average performance per watt value of 5.10. Therefore, the FPGA-based system provides 11 times better MFLOPs/Watt performance compared to the state of the art commodity cluster computer node.

VIII. CONCLUSION

A multi-FPGA based accelerator for solving a system of linear equations was proposed and the design was implemented on the Starbridge Hypercomputer, HC-36. Block-based versions of the forward and backward substitution algorithms are proposed. Our FPGA design is scalable and can be ported to any
similar multi-FPGA system. Results obtained are compared with a state-of-the-art low power cluster and an improvement of 11 times in the power efficiency metric is obtained for the proposed design.

REFERENCES


