Efficient Floating-point based Block LU Decomposition on FPGAs*

Vikash Daga[2], Gokul Govindu[1†], Viktor Prasanna[1], Sridhar Gangadharpalli[2] and V. Sridhar [2]

[1] Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089 USA
{govindu, prasanna}@usc.edu

[2] Applied Research Group
Satyam Computer Services Ltd.
Bangalore 560025, India
{vikash_daga, sridhar_gangadharpalli, sridhar}@satyam.com

Abstract

In this paper, we propose an architecture for floating-point based LU decomposition for large-sized matrices. Our proposed architecture is based on the well known concept of blocking and uses pipelined floating-point units to obtain high throughput. We first analyze the effects of block size and the deeply pipelined floating-point units on the performance of the architecture. We analyze and compare the performance of our double-precision based design with that of a GPP based design. Initial results show that an improvement of upto 23x in the total computation time can be achieved. We then, analyze the impact of algorithm level design (by varying block size) on the system-wide energy dissipation and resource-usage of our designs.

Categories: 1. Theory, Mapping and Parallelization and 4. Applications

1. Introduction

The increasing densities and capabilities of FPGAs now permit efficient floating-point arithmetic without being prohibitive in terms of either area or throughput. Recent FPGAs [8] [1] are now an attractive choice for implementing compute-intensive hardware accelerators. These FPGAs have embedded ASIC adders/multipliers (with fast carry chains, etc), efficient MUXes and a large number of flipflops, which can be used as components for building floating-point units. Alongwith these hardware resources, critical floating-point components like fixed-point adders/multipliers, shifters, priority encoders, etc are available as IP or can be easily synthesized. Hence the focus is moving away from building the floating-point units, to using them effectively in large architectures. Also, recent FPGAs feature large RAM blocks which can be effectively used as large and fast on-chip memory.

Applications like scientific computing and high-performance embedded computing demand high numerical stability and large dynamic range and thus require floating-point arithmetic. Applications like scientific computing demand high-performance and double-precision arithmetic. Constraints to such applications are large problem sizes, available resources and memory bandwidth. Hence, the architectures must be scalable (on either single or multi-chip FPGA resources), use high-throughput (using deep pipelining) floating-point units and fixed I/O (using linear array architectures). On the other hand, applications like high-performance embedded computing demand energy-efficiency and mostly single-precision arithmetic. FPGAs are not neccesarily low power but since architectures on FPGAs exploit parallelism the total energy used for a given problem can be reduced by analyzing energy-efficiency at the algorithm or architectural level and by obtaining an energy-efficient "scheme". By "scheme", we mean that the energy consumed in a solving a complete problem of given size is reduced, by trading-off latency with power.

LU decomposition is an important linear algebra and signal processing kernel. LU decomposition can be implemented on FPGAs to accelerate algorithms in scientific computing etc where latency performance of the hardware accelerator is important. Approaches to future wireless communications such as Software Defined Radio, require implementations of energy-efficient signal processing kernels (adaptive beamforming, etc) on reconfigurable hardware like FPGAs [7].

In this paper we propose and analyze a floating-point based architecture for LU decomposition which utilizes blocking for large-sized matrices. We use pipelined floating-point units. We first focus on performance

*This work is supported by the National Science Foundation under award No. CCR-0311823 and in part by an equipment grant from HP.

†Govindu's work is supported by Satyam Computer Services.

(throughput/latency and resources) of architectures using double-precision floating-point arithmetic (intended for use in scientific computing). We then focus energy-efficiency alongwith the performance of single-precision architectures (intended for use in high-performance embedded computing). To achieve the above goals rapidly without having to actually implement all the points in the design space, we employ domain-specific modeling proposed in [2]. We estimate the performance of our designs and also perform high level design tradeoffs by capturing algorithm and architectural details of the target FPGA device.

The rest of the paper is organized as follows. In Section 2, we discuss the basic and blocked versions of the LU algorithm and our previous work in developing fixed and floating-point architectures for LU decomposition. In Section 3, we present our architecture. In Section 4, we present the implementation details of the floating-point based block LU decompostion architecture. And finally, in Section 5, we discuss the scope for future work and conclude the paper.

2 Background

2.1 The Basic LU Architecture

LU Decomposition is essentially factoring a square matrix into an upper triangular matrix and a lower triangular matrix. $A(a_{x,y})$, a $n \times n$ matrix, is decomposed into Lower triangular matrix L ($l_{x,y}$) and Upper triangular matrix U ($u_{x,y}$) both of size $n \times n$, where x is the row index and y is the column index. In this paper, we assume that matrix A is a non-singular matrix and, further, we do not consider pivoting. Details about the sequential algorithm can be found in [4].

In [5] we presented a minimal-latency and resource efficient architecture for LU decomposition of small-sized matrices. This architecture has been slightly modified to obtain the architecture used for block LU, shown in Figure 1 (b). The architecture is based on a circular linear array which has n PEs where n is the problem size. The input and output ports to the architecture are connected to the first PE, PE_1. PE_1 is different from the other PEs in that it has a divider and doesn't have the multiplier/subtractor as in the other PE_2 to PE_n which are identical. The elements of the input matrix are fed in column-major order. The output matrix is the combined L and U matrices. Each $PE_{2 \leq j \leq n}$, shown in Figure 1 (a), has two input and two output ports. The last PE requires only a single output port and this output port is connected back to the second input port of PE_1. This essentially facilitates scheduling for the division required for the L matrix, to happen in PE_1. Thus the data is input into PE_1 and passes through PE_2 to PE_n and comes back to PE_1 and is fed out as output. After division in PE_1, the elements of the L matrix are both fed out as output and fed into PE_2 through PE_n for further iterations. We use pipelined floating-point units (FPUs) in order to achieve high throughput and in [5] we proposed the approach of stacked matrices to resolve the data dependencies incurred due to the large latencies of the deeply pipelined FPUs. The stacked matrices approach is nothing but computing the LU of a series of matrices. That is, if $A1$, $A2$, ... As are s matrices, data is input as ($a^1{}_{1,1}$,$a^2{}_{1,1}$,...,$a^s{}_{1,1}$,$a^1{}_{2,1}$,...,$a^{s-1}{}_{n,n}$,$a^s{}_{n,n}$). The number of the stacked matrices, s should be more than the combined latency of the floating-point compute elements in the PEs, in order to resolve the data dependencies.

3 Architecture for Floating-point based Block LU Decomposition

Since the resources available on a single FPGA chip limit the number of PEs, for large-sized matrices, we propose a block LU decomposition architecture. This architecture (shown in Figure 1 (c)) is based on the block LU decomposition architecture proposed by Choi et. al. [3] and the LU architecture presented in [5]. LU Decomposition of a large matrix A of size $n \times n$ can be achieved by performing decompositions of submatrices of size $b \times b$ along with other operations needed to update the entries. Matrix A can be represented as four matrices A_{11}, A_{12}, A_{21} and A_{22}. A_{11} is a $b \times b$ matrix, A_{12} is an $b \times (n-b)$ matrix, A_{21} is $(n-b) \times b$ matrix and A_{22} is $(n-b) \times (n-b)$ matrix. This matrix has to be decomposed into two $n \times n$ matrix L and U, such that

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L'_{11} & 0 \\ L'_{21} & L'_{22} \end{pmatrix} \begin{pmatrix} U'_{11} & U'_{12} \\ 0 & U'_{22} \end{pmatrix}.$$

The steps of the algorithm are as follows:

Step 1: Perform a sequence of Gaussian eliminations on the $n \times b$ matrix formed by A_{11} and A_{21} in order to calculate the entries of L'_{11}, L'_{21}, and U'_{11}.

Step 2: Calculate U'_{12} as the product of $(L'_{11})^{-1}$ and A_{12}.

Step 3: Evaluate $A'_{22} \leftarrow A_{22} - L'_{21}U'_{12}$.

Step 4: Apply Step 1 to 3 recursively to matrix A'_{22}. During the k-th iteration, the resulting submatrices $L^{(k)}_{11}$, $U^{(k)}_{11}$, $L^{(k)}_{21}$, $U^{(k)}_{12}$ and $A^{(k)}_{22}$ are obtained.

The block LU decomposition architecture utilizes a floating-point matrix multiplication architecture [9] [6] and a single PE for performing matrix subtraction. Thus, two sets of PEs are used: one set of b PEs for $b \times b$ LU decomposition and another set of b PEs for $b \times b$ matrix multiplication and a single PE for matrix subtraction. Each set of PEs is connected to a memory bank where the initial input

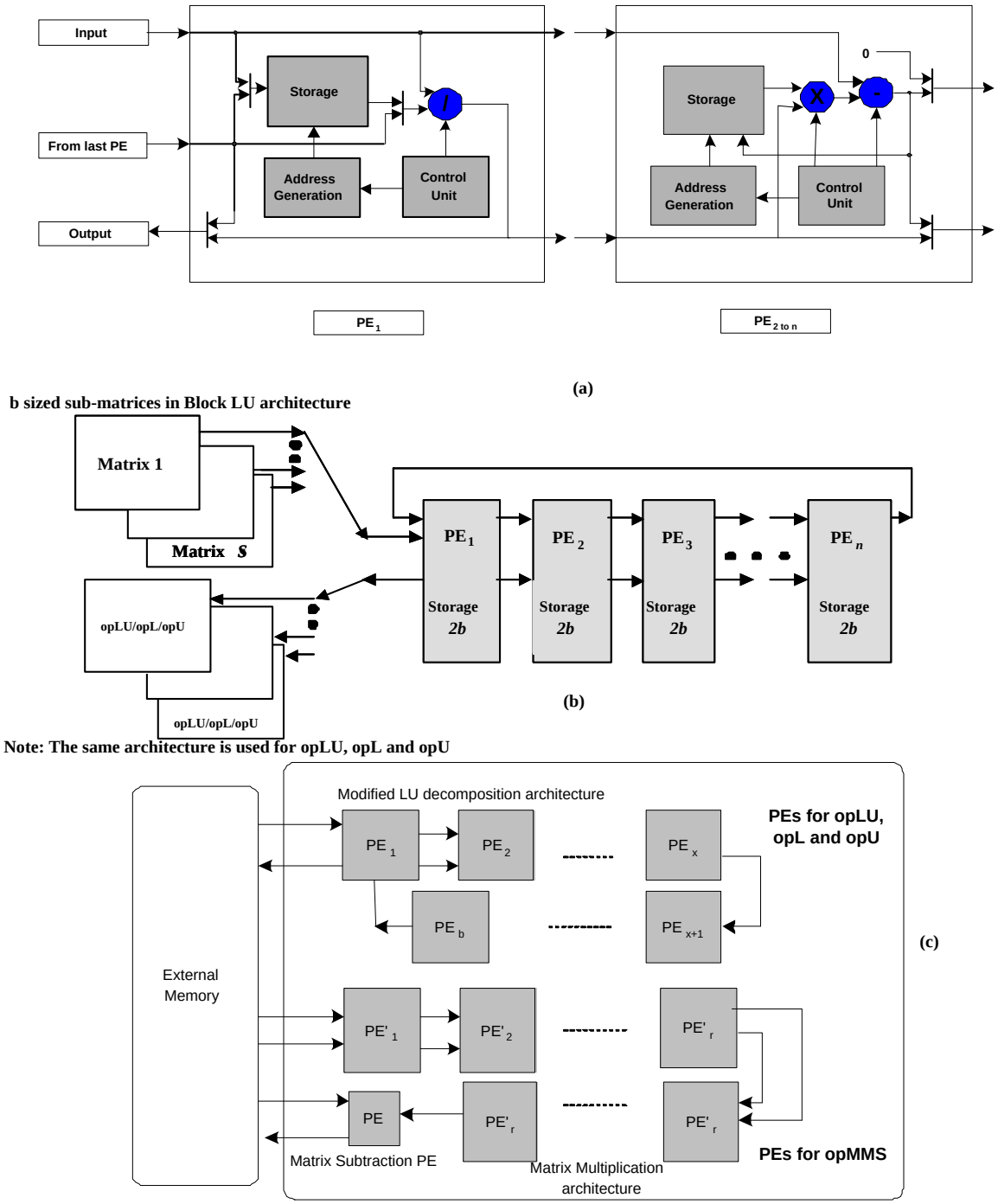

Figure 1. LU architecture:(a) Processing Element architecture: PE_1 has a divider only and rest of the PEs are identical with a multiplier and adder each, (b) Floating-point LU decomposition architecture, (c) The overall architecture. Note: opLU, opL and opU will be introduced later.

and intermediate block matrices are stored. Intermediate elements during computation are stored on-chip. The operations performed are identified in [3] as *opLU*, *opL*, *opU* and *opMMS*. *opLU* is performed to obtain L_{11}^k and U_{11}^k using the architecture in Figure 1 (a). *opL* from Step 1 is performed to obtain L_{21}^k . *opU* from Step 2 is performed to obtain U_{12}^k. Both *opL* and *opU* use the same architecture used for *opLU*. However, the effective latency for these operations increases to $n^2 + n$ unlike n^2 for $opLU$, due to schedule conflicts. *opMMS* is performed to obtain A_{22}^k which is essentially matrix multiplication and matrix subtraction. $\frac{n}{b}$ number of iterations are required to decompose a matrix A by the block decomposition method. Since our basic LU architecture also has the same latency as the basic LU architecture in [3], the latency for the block LU method will also be the same. It was proved in [3] that block LU decomposition of a $n \times n$ fixed-point matrix can be performed in $b^2 + 2b^2 + b^2\frac{1}{6}(\frac{n}{b})(\frac{n}{b} - 1)(\frac{2n}{b} - 1) + b^2 + b$ cycles, using b PEs for $b \times b$ LU decomposition, where b is block size.

We can use the stacked matrix approach for floating-point block LU decomposition, where the stacked matrices can be the $b \times b$ sized submatrices of the input matrix. Note that the stacked matrices approach is required only for the LU architecture. The matrix multiplication and subtraction do not require stacking [9]. As mentioned earlier, the number of the stacked matrices, s should be more than the combined latency of the floating-point compute elements in the PEs, in order to resolve the data dependencies. In other cases when we cannot stack the matrices either due to data dependencies or non-availability of more submatrices, we simply use stacked zero matrices. Zero matrix stacking due to data dependencies will occur for each of the n *opLU* operations with $s - 1$ zero matrices. As the iterations progress, there will be lesser number of block matrices to stack. Thus, from iteration to iteration, we will have one more zero matrix. However, the usage of zero matrices, just to satisfy dependencies is a necessary waste of energy and has to be minimized. Hence, the pipelining depth of the floating-point units assumes importance because of the zero matrix stacking required to satisfy the stack limit s. Operations on these zero matrices essentially consume wasteful energy.

The control is done using an FSM and the inputs to the FSM are the x and y index values of element $a_{x,y}$. These values can be simply generated using counters either inside the PEs or can be passed on through the PEs along with the data elements. The storage (using BRAMs) in each PE is $2 \times b$ and an address generator outputs the address to the BRAM based on the values of x and y.

In each iteration after the *opLU* is computed, *opL* and *opU* operations can be pipelined since they do not have data dependencies. During an iteration k, the number of matrices available for *opL* and *opU* operations will be $2(\frac{n}{b} - k)$. These can be pipelined without zero matrix stacking as long as $2(\frac{n}{b} - k) \geq s$. Similarly the number of matrices available for *opMMS* in iteration k is $(\frac{n}{b} - k)^2$. In Figure 2, the schedule is shown for different number of blocks.

Theorem 1. *Block LU Decomposition of $n \times n$ floating point matrix can be performed in $2sb^2 + b^2\lceil\frac{\frac{1}{6}(\frac{n}{b})(\frac{n}{b}-1)(\frac{2n}{b}-1)}{s}\rceil s + sb^2 + \frac{n}{b}bs - s$ cycles with the architecture in Figure 1 using b PEs for $b \times b$ LU decomposition, where b is block size and s is the stack size.*

Proof: At each k^{th} iteration, one *opLU*, $(n/b - k)$ *opL*, $(n/b - k)$ *opU* and $O(n/b - k)^2$ *opMMS* operations are performed. Since two sets of PEs are used, *opMMS* can be performed in parallel with *opL* and *opU* if data dependencies are satisfied. After the first computation of *opLU*, *opL* and *opU*, the remaining operations are overlapped with *opMMS* except for the last cycle where only *opLU* will be performed. The floating-point block LU decomposition architecture essentially uses the $b \times b$ submatrices themselves as stacked matrices. However as the iterations progress, zero matrix stacking is required in order to satisfy the stack size limit. Due to the stacked matrices approach the effective latency of *opLU* becomes sb^2 and that of *opL* and *opU* becomes $sb^2 + sb$. With the stacked matrices approach, effective latency of *opMMS* becomes $b^2\lceil\frac{\frac{1}{6}(\frac{n}{b})(\frac{n}{b}-1)(\frac{2n}{b}-1)}{s}\rceil s$. An additional $sb - s$ clock cycles are required for the pipeline to fill. Hence, the effective latency for block LU decomposition is $2sb^2 + b^2\lceil\frac{\frac{1}{6}(\frac{n}{b})(\frac{n}{b}-1)(\frac{2n}{b}-1)}{s}\rceil s + sb^2 + \frac{n}{b}bs - s$.

4 Implementation Details and Analysis

In this section, we present and analyze the implementation details of the floating-point block LU decomposition architecture. To estimate the performance of our designs and also perform high level design tradeoffs, we have employed domain-specific modeling proposed in [2]. Domain-specific modeling is a hybrid (top-down plus bottom-up) approach to performance modeling that allows the designer to rapidly evaluate candidate algorithms and architectures in order to determine the design that best meets criteria such as energy, latency, and area. In the top-down portion of the hybrid approach, the designer's knowledge of the architecture and the algorithm is incorporated, by deriving the performance models to estimate energy, area, and latency. The bottom-up portion is the actual area, latency and power estimation of the individual components obtained from the tools and from low level simulations. In our implementations we observed that the error between the results from actual implementation and the results from the domain-specific modeling approach was usually less than 10% [2]. The components for the block LU architecture are the floating-point units, storage elements like registers,

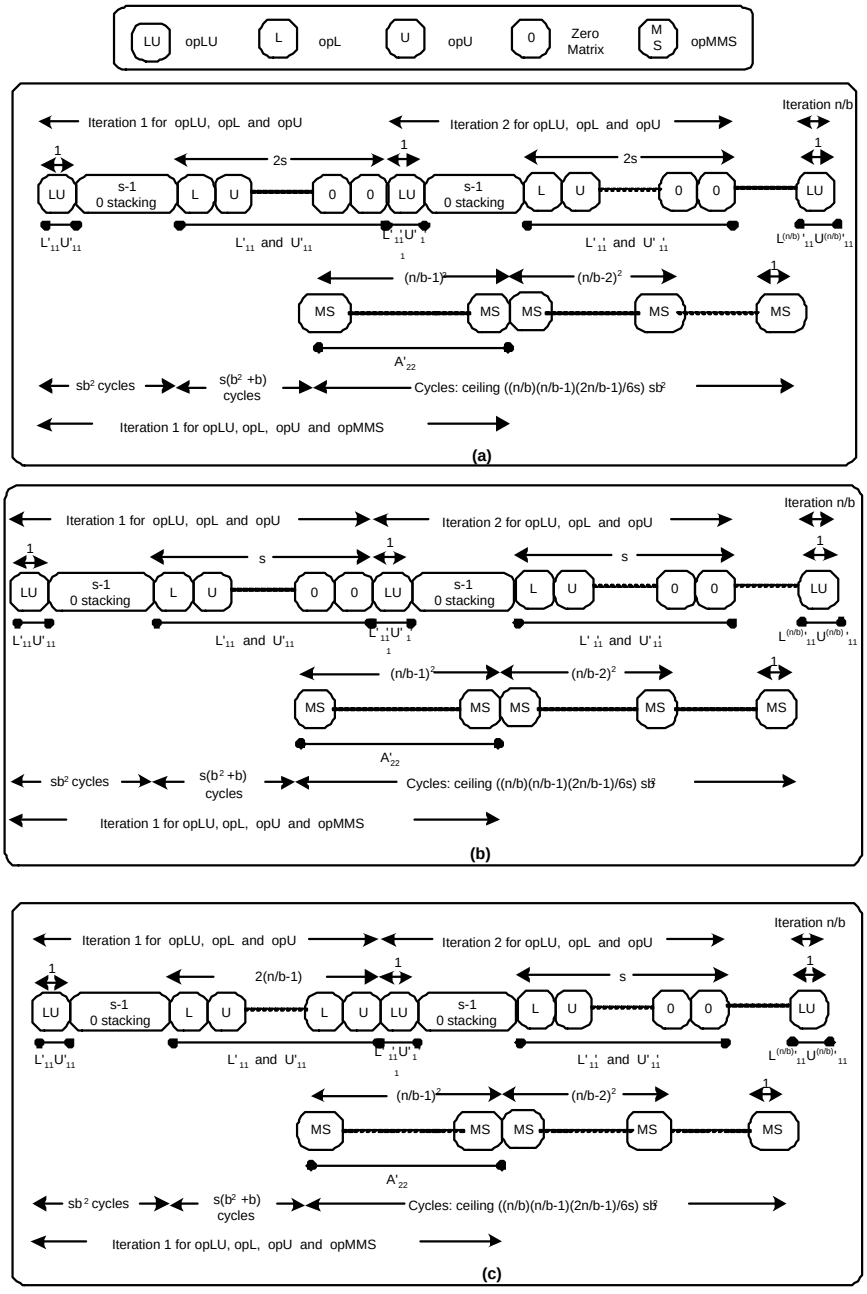

Figure 2. Scheduling for Block LU for (a)$s > (\frac{n}{b} - 1) > \frac{s}{2}$, (b)$(\frac{n}{b} - 1) < \frac{s}{2}$ and (c)$(\frac{n}{b} - 1) = \frac{s}{2}$

Table 1. Floating-point Units

	Single Precision									Double Precision		
	Subtractor			Multiplier			Reciprocator			Subtractor	Multiplier	Divider
	Min	Med	Max	Min	Med	Max	Min	Med	Max			
No. of Pipeline stages	6	12	16	4	7	10	4	4	4	19	12	32
Area (slices)	402	425	520	130	201	229	64	64	64	933	910	4041
Area(FFs/LUTs)	462/539	493/532	516/530	133/140	278/141	344/159	109/37	109/37	109/37	1148/1032	1546/948	4617/ 5,024
Clock Rate (MHz)	100	150	220	100	150	220	100	150	220	200	200	100
Power (mw)	235.4	303.6	445.5	133.7	156.4	231.4	94.3	118.6	183	-	-	-

slice and block-based memory and other elements like multiplexers, counters, etc. For low level simulation results, the component designs were coded in VHDL, implemented using Xilinx ISE 5.2i on the Xilinx Virtex-IIPro XC2VP125 device. The areas were obtained from the PAR report and power was obtained using ModelSim 5.7 and XPower.

Table 1 shows the resources, latencies and power values for the floating-point units. For single precison, we used floating-point units of various depths and for division we used a approximate reciprocator (using a BRAM based lookup table) and a multiplier. Using a look up table (we used one with an input of 2^{13} bits and an output of 2^{24} bits) based floating-point reciprocator unit does involve an error in precision but not in the dynamic range.

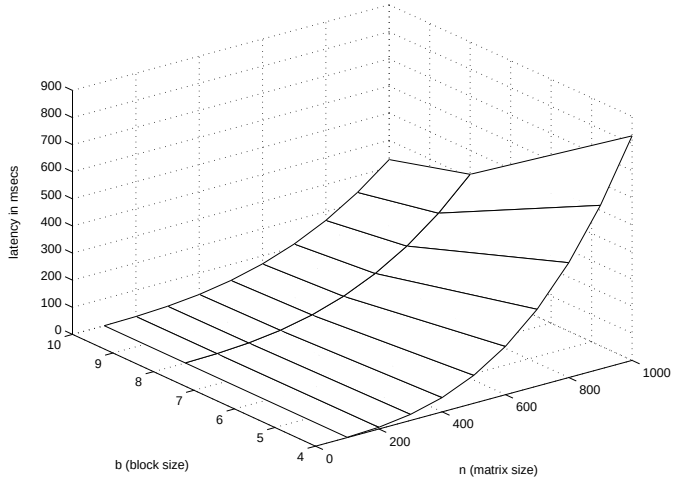

Figure 3. Latencies for LU decomposition of different problem sizes and different block sizes

We first compare the performance of double precision designs. Figure 3 (a) shows the comparison of the latencies of double precision LU decomposition of matrices of sizes ranging from 100 to 1000. The maximum block size is limited to 10 because only 10 double-precision PEs of LU and matrix multiplication each can fit into the largest Xilinx FPGA, which has 55,000 slices. Note that, when we use deeply pipelined floating-point units, we pay a penalty in terms of the zero matrix stacking, in order to attain high throughput rates. For the sake of comparison, we consider the performance of a GPP (Intel Pentium M with SSE2, 1400MHz, 1MB L2 and 512MB) as the baseline (not necessarily high-performance), in Table 2. We see that the our architecture can achieve substantial improvement.

Table 2. Latency comparison between our architecture and a GPP based design

	Our architecture, 100Mhz	Pentium M (SSE2), 1.4GHz	
Size($n \times n$)	Latency(msec)	Latency(msec)	Improvement (x times)
100x100	0.46	9.11	19.8
300x300	8.76	134.2	15.4
500x500	40.5	661	16.3
800x800	167.6	2984.5	17.8
1000x1000	328.4	7871.5	23

We now analyze the energy performance of single precision block LU decomposition and the effects of the depth of pipelining of the floating-point units. Figures 4 and 5 show the energy, area, latency values for a small problem size of 48 and a large problem size of 1000 respectively for various block sizes and FPU sets. Figures 4 (a) and 5 (a) show the energy for various block sizes for problem size 48 and 1000 respectively. Figures 4 (b) and 5 (b) show the resources. Figures 4 (c) and 5 (c) show the latency for various block sizes. Figure 4 (d) and 5 (d) show the energy distribution for $s = 10$ and $s = 19$ respectively. We can infer from the figures that for a small problem size, a smaller s and b reduce the overall energy. This is because smaller s and b reduce wasteful energy dissipation due to the zero matrix stacking effect. For large problem sizes, the medium s and b ($s = 19, b = 20$) reduce the overall energy. This essentially a tradeoff between area (increases with increase in s, b) and latency (decreases with increase in s, b). Moreover, midsize s and b reduce the zero matrix stacking. This analysis can be easily extended to multi-chip designs also.

5 Concluding Remarks

In this paper we have proposed an architecture for block LU decomposition on FPGAs. Preliminary results show a substantial improvement in the total computation time. Future work will involve the implementation of the architecture coupled with a host computer. Also, the PowerPC processor on the Virtex II Pro platform FPGA will be used to perform some of the address generation and dataflow control.

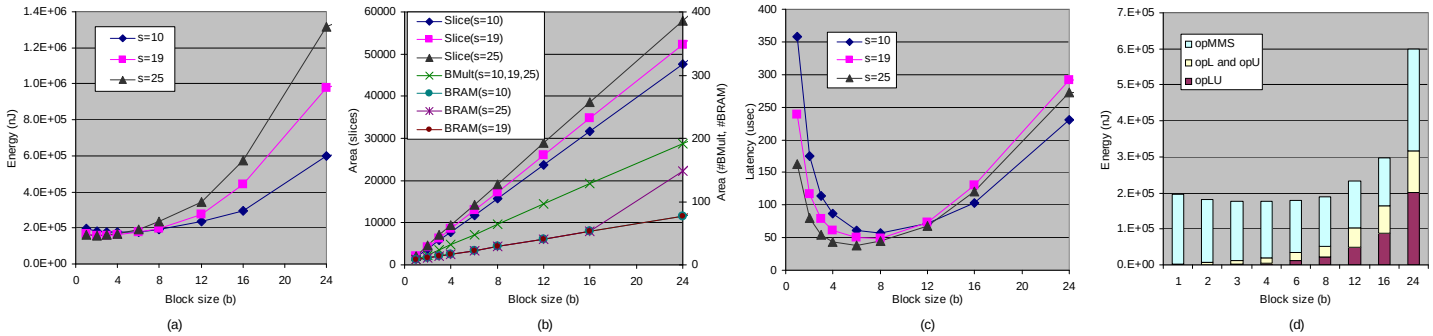

Figure 4. (a) Energy vs block sizes, (b) Resources vs block sizes, (c) Latency vs. block sizes and (d) Energy distribution with $s = 10$ for problem size $n = 48$.

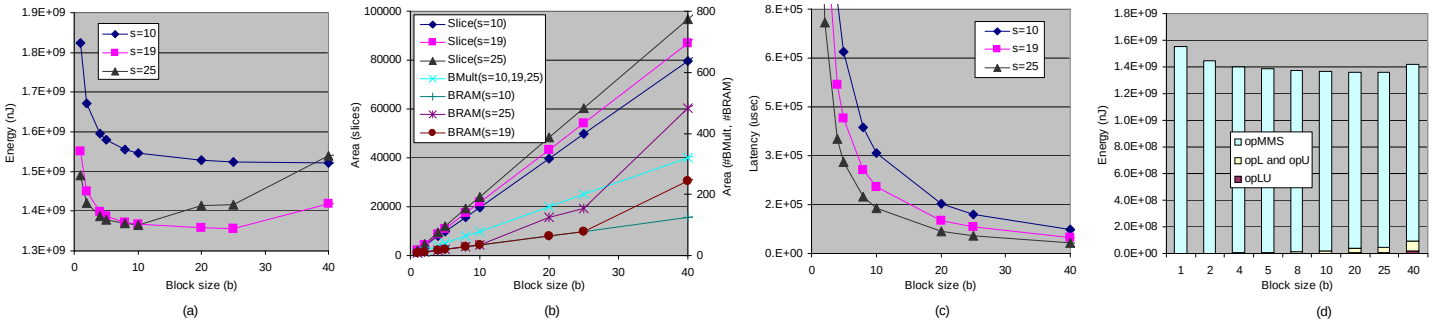

Figure 5. (a) Energy vs block sizes, (b) Resources vs block sizes, (c) Latency vs. block sizes and (d) Energy distribution with $s = 19$ for problem size $n = 1000$

6 Acknowledgement

The authors would like to thank Seonil Choi, who had initially developed the LU decomposition architecture, for his invaluable assistance.

References

[1] Altera. http://www.altera.com/.

[2] S. Choi, J. Jang, S. Mohanty, and V. K. Prasanna. Domain-Specifi c Modeling for Rapid System-Wide Energy Estimation of Reconfi gurable Architectures. *ERSA*, 2002.

[3] S. Choi and V. K. Prasanna. Time and Energy Effi cient Matrix Factorization using FPGAs. *13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, September 2003.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. 2001.

[5] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar. A High-performance and Energy Effi cient Architecture for Floating-point based LU Decomposition on FPGAs. In *Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.

[6] J. Jang, S. Choi, and V. Prasanna. Area and Time Effi cient Implementation of Matrix Multiplication on FPGAs. In *Proc. of The First IEEE International Conference on Field Programmable Technology*, California, USA, December 2002.

[7] W. Tuttlebee. *Software Defined Radio: Enabling Technologies*. 2002.

[8] Xilinx. http://www.xilinx.com.

[9] L. Zhuo and V. Prasanna. Scalable and Modular Algorithms for Floating-point based Matrix Multiplication on FPGAs. In *International Parallel and Distributed Processing Symposium*, New Mexico, USA, April 2004.