

Time and Energy Efficient Matrix Factorization using FPGAs

Seonil Choi and Viktor K. Prasanna

Electrical Engineering-Systems
University of Southern California, Los Angeles, USA
{seonilch, prasanna}@usc.edu
<http://ceng.usc.edu/~prasanna>

Abstract. In this paper, new algorithms and architectures for matrix factorization are presented. Two fully-parallel and block-based designs for LU decomposition on configurable devices are proposed. A linear array architecture is employed to minimize the usage of long interconnects, leading to lower energy dissipation. The designs are made scalable by using a fixed I/O bandwidth independent of the problem size. High level models for energy profiling are built and the energy performance of many possible designs is predicted. Through the analysis of design tradeoffs, the block size that minimizes the total energy dissipation is identified. A set of candidate designs was implemented on the Xilinx Virtex-II to verify the estimates. Also, the performance of our designs is compared with that of state-of-the-art DSP based designs and with the performance of designs obtained using a state-of-the-art commercial compilation tool such as Celoxica DK1. Our designs on the FPGAs are significantly more time and energy efficient in both cases.

1 Introduction

FPGAs have become an attractive option for implementing digital signal processing applications because of their high processing power and customizability [7]. The inclusion of new features in the FPGA fabric, such as a large number of embedded multipliers, further enhance their suitability. Recent FPGAs such as Xilinx Virtex-II(pro) [15] and Altera Stratix [1] offer hundreds of multipliers and large memory on a single chip. FPGAs can now be considered for implementing massively parallel and computationally demanding applications [12]. Also, with the proliferation of portable and mobile devices [2], it has become increasingly important that systems are not only fast, but also energy efficient. Even though state-of-the-art configurable devices offer very few features for power control, we show how to effectively use them to improve energy performance.

In this paper, we consider one of the important signal processing kernels: matrix factorization. For example, matrix factorization is a fundamental kernel in adaptive beamforming [9]. Approaches to future wireless communications such as software defined radio (SDR), require the mapping of such signal processing kernels onto reconfigurable hardware like FPGAs [14]. Moreover, the implementations have to be time and energy efficient. First, we develop a linear array architecture based design for matrix factorization. Then we investigate and apply algorithmic techniques that use a block based approach to obtain time and energy efficient designs in FPGAs. Performance estimation (based on the time and energy performance models) is used for rapid design space exploration. Since block matrix factorization can be realized using various

block sizes, we identify an optimal block size that minimizes the total energy dissipation based on the estimation. Candidate designs are implemented. To the best of our knowledge, there are no FPGA based designs for LU decomposition. Hence for the sake of comparison, we implement the matrix factorization on FPGAs using the state-of-the-art compilation tool, Celoxica DK1, with Handel-C [4] and also implement it in software on TI DSP devices. The performance of our designs is compared with that of the DK1 based design and the TI DSP benchmarks.

The remainder of this paper is organized as follows. In Section 2, we present our algorithms and architectures for matrix factorization. In Section 3, time and energy performance is estimated for the proposed algorithms and architectures. Section 4 presents the implementation details and the performance of these synthesized designs. Also, a comparison with Handel-C based and TI DSP-based implementations is made. Finally, Section 5 summarizes our work and discusses possible areas for future work.

2 Time and Energy Efficient Designs for Matrix Factorization

Several methods are known for factoring a given matrix [5]. In this paper, we choose to implement LU decomposition on FPGAs. Essentially, LU decomposition factors a $b \times b$ matrix into a $b \times b$ lower triangular matrix L (the diagonal entries are all 1) and a $b \times b$ upper triangular matrix U .

We propose two designs using two theorems. In Theorem 1, a new algorithm and architecture for LU decomposition is developed for a linear array of processing elements (PEs). Each PE performs computations on the input or intermediate matrix and the results are fed to the neighboring PE. Data dependencies between input and intermediate matrices are solved by efficient and regular scheduling. Each PE uses only two input ports: one for feeding input or intermediate matrices and the other for outputting the decomposed matrix. With this fixed I/O bandwidth regardless of problem size, we achieve an optimal latency of $b^2 + b - 1$ with leading coefficient of 1. The best latency of previously proposed designs [3] is $2b(b + 1)$. In Theorem 2, a new parallel design on FPGAs for block LU decomposition is proposed. The design partitions a large matrix into multiple smaller blocks. To perform a computation for the smaller blocks, the architecture/algorithm in Theorem 1 is re-used. By varying the block size, we achieve time and energy efficient designs.

2.1 LU Decomposition

Let A be a $b \times b$ matrix. $a_{x,y}$ denotes an element of matrix A , where x is the row index and y is the column index. Similarly, $l_{x,y}$ ($u_{x,y}$) denotes an element of matrix L (U). We assume that matrix A is a non-singular matrix and, further, we do not consider pivoting. The sequential algorithm in [8] consists of three main steps:

Step 1: The column vector $a_{x,1}$ where $2 \leq x \leq b$ is multiplied by the reciprocal of $a_{1,1}$. The resulting column vector is denoted $l_{x,1}$.

Step 2: $l_{x,1}$ is multiplied by the row vector $a_{1,y}$ ($= u_{1,y}$) where $2 \leq y \leq b$. The product $l_{x,1} \times u_{1,y}$ is computed and subtracted from the submatrix $a_{x,y}$ where $2 \leq x, y \leq b$.

Step 3: Step 1 and 2 are recursively applied to the new submatrix formed in Step 2.

An *iteration* denotes an execution of Step 1 and 2. During the k -th iteration, the column vector $l_{x,k}$ and the row vector $u_{k,y}$ where $k + 1 \leq x, y \leq b$ are generated.

The product $l_{x,k} \times u_{k,y}$ is subtracted from the submatrix $a_{x,y}$ where $k \leq x, y \leq b$ obtained during the $(k - 1)$ -th iteration.

The time complexity of the sequential algorithm is $\Theta(b^3)$. We propose an architecture and algorithm on a linear array shown in Figure 1 using b PEs. The number

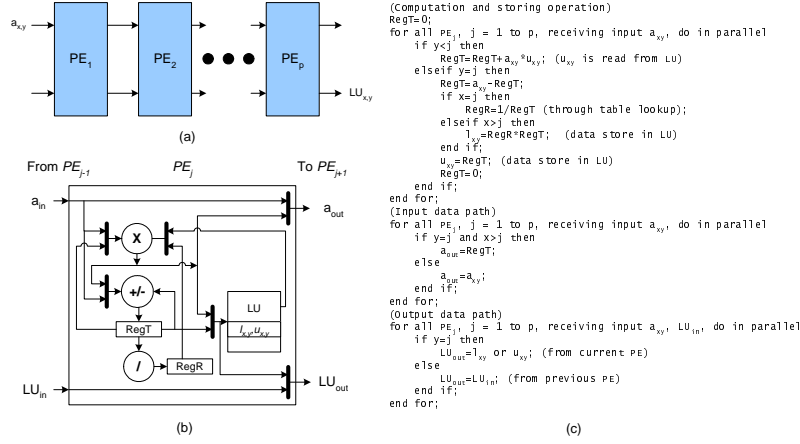


Fig. 1. (a) Overall architecture, (b) architecture of PE, and (c) algorithm for LU decomposition

of PEs p is the same as the problem size b . Essentially, PE_j performs computations for the j -th column of matrices L and U . Each PE consists of an adder/subtractor, a multiplier, a division lookup table, and a storage LU (p entries per PE). The storage LU of PE_j is used to store the j -th column of matrices L and U . Each PE has two input ports (\mathbf{a}_{in} , LU_{in}) and two output ports (\mathbf{a}_{out} , LU_{out}). \mathbf{a}_{in} and \mathbf{a}_{out} are used to feed in and out $a_{x,y}$ or $l_{x,y}$. LU_{in} and LU_{out} are used to output resulting matrices L and U to PE_b in a pipelined manner. Figure 1 (c) shows our algorithm by describing the operations in each PE during each cycle.

Theorem 1. *LU decomposition without pivoting of a non-singular $b \times b$ matrix can be performed in $b^2 + b - 1$ cycles using the architecture and the algorithm in Figure 1 using b PEs.*

Proof. The elements $a_{x,y}$ of matrix A are fed in row major order ($a_{1,1}, a_{1,2}, a_{1,3}, \dots, a_{1,b}, a_{2,1}, \dots, a_{b,b}$) to \mathbf{a}_{in} of PE_1 . All data are fed from left to right. $a_{x,y}$ arrives at PE_j at cycle $b(x-1) + y + j - 1$ where $1 \leq j \leq b$. Seven operations are performed based on indices x, y and index j of PE_j . The indices x and y can be realized using counters in each PE. They also can be fed to PE_1 and propagated in a pipelined manner.

Op 1) Data propagation: $a_{x,y}$ is passed from PE_{j-1} to PE_{j+1} via PE_j except when $y = j$ and $x > j$. If $y = j$ and $x > j$, $l_{x,y}$ is generated at PE_j and is passed to PE_{j+1} via port \mathbf{a}_{out} . $l_{x,y}$ is also stored in the LU of PE_j .

Op 2) Multiplication/Accumulation: If $y < j$, a multiplication and an accumulation are performed in PE_j . $a_{x,y}$ ($= l_{x,y}$ generated at PE_{j-1}) is fed via port \mathbf{a}_{in} . During the k -th iteration, PE_j computes the product of the column vector $l_{x,k}$ and the j -th entry from $u_{k,y}$, where $l_{x,k}$ is a column vector generated in PE_k and $u_{k,y}$ is a row vector generated from PE_{k+1} to PE_b during the k -th iteration ($k+1 \leq x, y \leq b$). $u_{k,y}$ are stored in the LUs of PE_{k+1} to PE_b . An accumulation, $a_{x,y}^{(k)} = l_{x,k} \times u_{k,y} + a_{x,y}^{(k-1)}$, is performed after the multiplication during the same clock cycle. $a_{x,y}^{(k)}$ denotes the intermediate element of submatrix generated during the k -th iteration. $a_{x,y}^{(k)}$ is used either for another accumulation or for normalization (*Op 6*) and is stored in RegT . RegT is a temporary storage to hold $a_{x,y}^{(k)}$ during the accumulation. Note that accumulation and subtraction share one adder/subtractor since they do not occur simultaneously.

Op 3) Subtraction: If $y = j$, a subtraction is performed after all accumulations are complete. This ensures that $a_{x,y}^{(k)}$ is subtracted from the submatrix $a_{x,y}$ where $k \leq x, y \leq b$ during k -th iteration. For example, $u_{3,3}$ is computed as $\{- (l_{3,1}u_{1,3} + l_{3,2}u_{2,3}) + a_{3,3}\}$.

In Step 2 of the sequential algorithm, the subtraction is performed after multiplication and the result is stored for the next subtraction. These operations are done repeatedly. For example, $u_{3,3}$ is computed as $\{(a_{3,3} - l_{3,1}u_{1,3}) - l_{3,2}u_{2,3}\}$.

Op 4) Storing: If $y = j$, $l_{x,y}$ or $u_{x,y}$ is generated in PE_j . If $x \leq j$, $u_{x,y}$ is stored in LU. If $x > j$, $l_{x,y}$ is stored in LU after normalization (*Op 6*). This operation ensures that the j -th column of the decomposed matrices L and U is stored in PE_j .

Op 5) Reciprocal: Division is required since the normalization is performed by $u_{k,k}$ for the column vector $a_{x,k}^{(k-1)} = (a_{k+1,k}, \dots, a_{b,k})$ during the k -th iteration ($1 \leq k \leq b$). $u_{k,k}$ is stored in **RegT** after subtraction (*Op 3*) and the reciprocal value of $u_{k,k}$ is stored in **RegR**. The reciprocal operation occurs if $x = y = j$.

Op 6) Normalization: After the subtraction (*Op 3*), the value is stored in **RegT**. If $y = j$ and $x > j$, the values in **RegT** and **RegR** are multiplied. This operation generates the column vector $l_{x,k}$ where $k+1 \leq x \leq b$ in PE_k during the k -th iteration.

Op 7) Output: This operation sends out the results $l_{x,y}$ and $u_{x,y}$ in LU in a pipelined manner. If $y = j$, $l_{x,y}$ or $u_{x,y}$ is sent to port LU_{out} . Otherwise, $l_{x,y}$ or $u_{x,y}$ from PE_{j-1} is passed to PE_{j+1} via port LU_{out} .

To satisfy the data dependency of $l_{x,k}$ being generated during the k -th iteration and used during the $(k+1)$ -th iteration and to obtain the minimum latency, two conditions have to be satisfied. Note that the column vector $l_{x,k}$ ($k+1 \leq x \leq b$) is produced in PE_k during the k -th iteration. The first condition is that $l_{x,k}$ has to propagate from PE_k to PE_{k+1} after $l_{x,k-1}$ (generated during the $(k-1)$ -th iteration) propagates to PE_{k+1} and before $l_{x,k+1}$ is generated in PE_{k+1} during the $(k+1)$ -th iteration. Let $T_k(l_{x,j})$ be the sum of the time when $l_{x,j}$ is generated in PE_j and the propagation time when it reaches PE_k , which is $T_k(l_{x,j}) = b(x-1) + 2(j-1) + 1 + k - j$. Then, $T_{k+1}(l_{x,k-1}) = b(x-1) + 2k - 1$, $T_{k+1}(l_{x,k}) = b(x-1) + 2k$, and $T_{k+1}(l_{x,k+1}) = b(x-1) + 2k + 1$. Since $T_{k+1}(l_{x,k-1}) < T_{k+1}(l_{x,k}) < T_{k+1}(l_{x,k+1}) \rightarrow -1 < 0 < 1$, the condition is satisfied for all x where $k+1 \leq x \leq b$. To define the second condition, let $lu_{x,y}$ be $l_{x,y}$ if $x > j$, or $u_{x,y}$ if $x \leq j$ in PE_j . Note that $lu_{x,j}$ is computed in PE_j every b cycles. To output the resulting matrices L and U without delay, the second condition that $lu_{x,j}$ arrives at PE_k via port LU_{in} and LU_{out} before PE_k produces any $lu_{x,k}$ is required to satisfy. We assume $j < k$. Then, $T_k(lu_{x,j}) = b(x-1) + j + k - 1$ and $T_k(lu_{x,k}) = b(x-1) + 2k - 1$. Since $T_k(lu_{x,j}) < T_k(lu_{x,k}) \rightarrow j < k$, the second condition is satisfied for all x where $1 \leq x \leq b$. Total latency is calculated as the time taken for the last result $lu_{b,b}$ to be available as output: $T_b(lu_{b,b}) = b(b-1) + 2(b-1) + 1 = b^2 + b - 1$. \square

Since our design is a pipelined architecture, the first b cycles of the computations on the next matrix can be overlapped with the last b cycles of the computations on the current matrix. For a stream of matrices, one matrix can be decomposed every b^2 cycles. Thus the *effective latency* becomes b^2 , which is the time taken to obtain the first output data to the last output data during the current computation.

2.2 Block LU Decomposition

For large matrices, block LU decomposition can be performed. The sequential algorithm is given in [5]. An $n \times n$ matrix A is partitioned into four matrices: A_{11} , A_{12} , A_{21} , and A_{22} . A_{11} is a $b \times b$ matrix, A_{12} is a $b \times (n-b)$ matrix, A_{21} is an $(n-b) \times b$ matrix, and A_{22} is an $(n-b) \times (n-b)$ matrix. The algorithm is to decompose A into two $n \times n$ matrices, 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 $L_{11}^{(k)}$, $U_{11}^{(k)}$, $L_{21}^{(k)}$, $U_{12}^{(k)}$, and $A_{22}^{(k)}$ are obtained. An *iteration* denotes an execution of Step 1 to 3.

By utilizing the architecture and algorithm in Theorem 1 in combination with a matrix multiplication/subtraction architecture, we propose an architecture for block LU decomposition on FPGAs as shown in Figure 2. The block size b is later used as the parameter to realize time and energy efficient designs. There are two sets of PEs: one set performing a $b \times b$ LU decomposition and the other performing a $b \times b$ matrix multiplication/subtraction. Each set of PEs is linearly pipelined and both sets are connected to a memory bank. The input matrix is stored in the memory bank and fed to both sets of PEs. After computation, the results are stored back to the memory bank and used for next computation. Four different operations are identified: *opLU*, *opL*, *opU*, and *opMMS*. *opLU* is performed to obtain $L_{11}^{(k)}, U_{11}^{(k)}$. *opLU* from Step 1 is realized by using the algorithm and architecture proposed in Theorem 1. *opL* from Step 1 is performed to obtain $L_{21}^{(k)}$. The same architecture in Theorem 1 is used. However, the matrix $U_{11}^{(k)}$ and the reciprocal of its diagonal entries are required to perform *opL*. Since *opL* is performed after *opLU*, all PEs already hold the reciprocals in RegRs. $U_{11}^{(k)}$ is fed via port LU_{in}. We add one more data path that feeds the data from port LU_{in} to storage LU. *opU* from Step 2 is performed to obtain $U_{12}^{(k)}$. *opU* also uses the same architecture. It requires $L_{11}^{(k)}$ from *opLU*. $L_{11}^{(k)}$ are fed via port LU_{in} to storage LU. In Step 3, matrix multiplication/subtraction (*opMMS*) is performed. Once *opL* and *opU* are complete, $L_{21}^{(k)}$ and $U_{12}^{(k)}$ are available for *opMMS*. We have proposed an architecture for this operation [7]. Since there is matrix subtraction after matrix multiplication, additional subtraction logic is added. The matrix multiplication algorithm takes two $b \times b$ submatrices C from $L_{21}^{(k)}$ and D from $U_{12}^{(k)}$ and computes the product $E \leftarrow C \times D$. Another $b \times b$ submatrix F is taken from $A_{22}^{(k)}$. Then the final values are obtained by $E \leftarrow F - E$. If b is large, the $b \times b$ matrix multiplication can be decomposed into $(\frac{b}{r})^3 r \times r$ matrix multiplications, where r is the sub-block size.

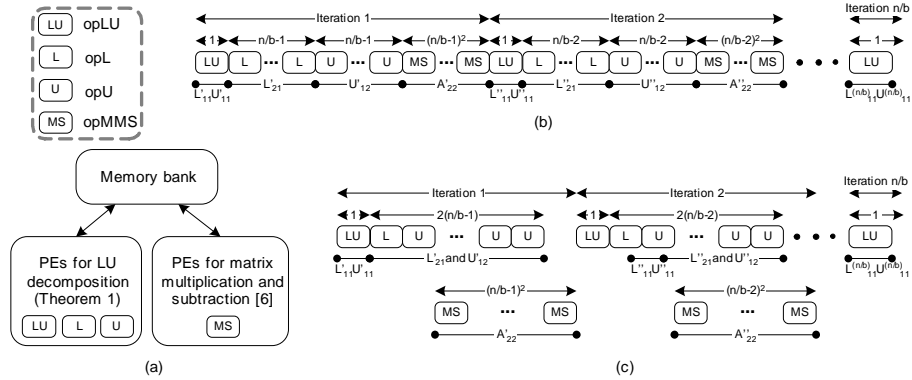


Fig. 2. (a) Overall architecture for block LU decomposition, (b) a schedule for Theorem 2 and (c) a schedule for Corollary 1

Theorem 2. *LU decomposition of $n \times n$ matrix can be performed in $n^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) \times (\frac{2n}{b} - 1) + b - 1$ cycles using the architecture in Figure 2 (a) using b PEs for $b \times b$ LU*

decomposition and r PEs for $r \times r$ matrix multiplication/subtraction, where b is block size and r is sub-block size.

Proof. At a given time, only one operation is performed and the schedule is shown in Figure 2 (b). As all matrices are fed as streaming input, the computation on the current matrix and the next matrix can be overlapped. Therefore, each of $opLU$, opL , and opU has an effective latency of b^2 . $opMMS$ has an effective latency of $\frac{b^3}{r}$ [7]. There are $\frac{n}{b}$ iterations to complete the block LU decomposition. During each iteration, only one $b \times b$ $opLU$ is performed. The effective latency of all $opLU$ is $(\frac{n}{b})b^2$. During the k -th iteration, $(\frac{n}{b} - k)$ opL and opU for $b \times b$ block size are performed. The effective latency of all opL and opU is $b^2 \sum_{k=1}^{n/b} (\frac{n}{b} - k)$. During the k -th iteration, $(\frac{n}{b} - k)^2$ $opMMS$ for $b \times b$ block size are performed. Since $b \times b$ matrix multiplication can be decomposed to $(\frac{b}{r})^3$ $r \times r$ matrix multiplications, the effective latency of all $opMMS$ is $\frac{b^3}{r} \sum_{k=1}^{n/b} (\frac{n}{b} - k)^2$. The total latency is $(\frac{n}{b})b^2 + 2b^2 \sum_{k=1}^{n/b} (\frac{n}{b} - k) + \frac{b^3}{r} \sum_{k=1}^{n/b} (\frac{n}{b} - k)^2 + b - 1$, which includes the time to fill the pipeline stages. \square

Theorem 2 uses a straightforward schedule since only one set of PEs performs computations at a given time. We can utilize the two sets of PEs in parallel to reduce the total latency.

Corollary 1. *LU decomposition of $n \times n$ matrix can be performed in $3bn - 2b^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) (\frac{2n}{b} - 1) + b - 1$ cycles using the schedule in Figure 2 (c).*

Proof. After opL and opU for the first blocks are performed, the input matrices for $opMMS$ are ready. Thus, opL and opU can be performed in parallel with $opMMS$. The effective latency to complete the k -th iteration is $3b^2 + (\frac{n}{b} - k)^2 \cdot \frac{b^3}{r}$. During the $\frac{n}{b}$ -th iteration, only one $opLU$ is performed. Thus, the total latency is $\sum_{k=1}^{n/b-1} \{3b^2 + (\frac{n}{b} - k)^2 \cdot \frac{b^3}{r}\} + b^2 + b - 1 = 3bn - 2b^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) (\frac{2n}{b} - 1) + b - 1$, which includes the time to fill the pipeline stages. \square

While Corollary 1 reduces the total latency compared with Theorem 2, it does not reduce the amount of computation. Total energy is the sum of the energy used for computation and quiescent energy (the energy for configuration memory, static energy, etc.) used by the device even when the logic is idle. The quiescent energy depends only on the total latency. Since Corollary 1 reduces the latency, the quiescent energy and hence the total energy are reduced. Thus we use the architecture and algorithm in Corollary 1 to obtain both time and energy efficient designs.

3 Performance Estimation and Design Trade-offs

For a given problem size n , varying the parameters such as block size b and sub-block size r creates a large design space. Before implementing the designs and performing low level simulation, we estimate the performance of possible designs, prune the design space, and finally identify "good" candidate designs for time and energy efficiency. The candidate designs were implemented using VHDL (See Section 4).

3.1 High Level Performance Model

To estimate the performance of our designs, we have employed domain-specific modeling proposed in [6]. 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. An architecture is divided into *RModules* and *Interconnects*. RModules are hardware elements that are assumed to dissipate the same amount of power no matter where they are instantiated on the device and Interconnects are the wires connecting the RModules. From the algorithm, we know when and for how long each RModule is active. With this knowledge, we can calculate the latency

of the design. Additionally, with estimates for the power dissipated by each RModule and the Interconnect, we can estimate the energy dissipated by the design. 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 power estimation of RModules and Interconnects from low level simulations. In our designs, the RModules are multipliers, adders, multiplexers, RAM, reciprocal lookup tables, and registers. The power values of each RModule are as follows. P_{Mult} ($= 11.25$ mW) is the power dissipation for a 16×16 multiplier, P_{Add} ($= 1.34$ mW) for a 16-bit adder/subtractor, P_{Div} ($= 7.31$ mW) for a division lookup table (1024×16 bit), P_{BSRAM} ($= 7.31 \lceil x/1024 \rceil$ mW) for an on-chip memory where x is the number of entries, P_R ($= 1.17$ mW) for a 16-bit register, and P_{Store} ($= 0.126 \lceil x/16 \rceil + 2.18$ mW) for a storage LU where x is the number of entries. Table 1 lists the performance models of our designs. The latencies are converted to seconds by dividing them by the clock frequency.

Table 1. Time and energy performance models

Design	Metric	Performance model
Theorem 1	Latency	$L_{Thm1} = b^2$
	Power	$P_{Thm1} = 6bP_R + bP_{Add} + bP_{Mult} + bP_{Store} + bP_{Div} + 2P_{BSRAM}$
	Energy	$E_{Thm1} = L_{Thm1} \cdot P_{Thm1}$
Theorem 2	Latency	$L_{opLU} = (\frac{n}{b})b^2, L_{opL} = L_{opU} = \frac{1}{2}(\frac{n}{b})(\frac{n}{b}-1)b^2$
		$L_{opMMS} = \frac{1}{6}(\frac{n}{b})(\frac{n}{b}-1)(2\frac{n}{b}-1) \cdot \frac{b^3}{r}$
		$L_{Thm2} = L_{opLU} + L_{opL} + L_{opU} + L_{opMMS}$
	Power	$P_{opLU} = P_{opL} = P_{Thm1}$
		$P_{opU} = 6bP_R + bP_{Add} + bP_{Mult} + bP_{Store} + 2P_{BSRAM}$
$P_{opMMS} = 8rP_R + rP_{Add} + rP_{Mult} + 2rP_{Store} + 3P_{BSRAM}$		
Energy	$E_{Thm2} = L_{opLU} \cdot P_{opLU} + L_{opL} \cdot P_{opL} + L_{opU} \cdot P_{opU} + L_{opMMS} \cdot P_{opMMS}$	
Corollary 1	Latency	$L_{Cor1} = L_{opLU} + 2(\frac{n}{b}-1)b^2 + L_{opMMS}$
	Power	the same as the ones in Theorem 2
	Energy	$E_{Cor1} = E_{Thm2}$

3.2 Design Trade-offs for Time and Energy Efficiency

To achieve time and energy efficient designs, we explore the various design parameters such as frequency, block size, precision, and number of PEs. All parameters contribute to energy dissipation, latency, and area of a design. For example, the latency and energy of Corollary 1 are a function of the block size b and the sub-block size r . By choosing $b = r = n$, the minimum latency of n^2 ($546.1 \mu\text{sec}$ at 120 MHz for $n = 256$) can be achieved. However, this design does not necessarily have minimum energy dissipation. We explore the parameters, b and r , and determine their values that minimize the energy dissipation. The estimates are based on 120 MHz designs by considering the operating frequency that can be achieved after implementation (See Section 4). Figure 3 (a) shows the energy dissipation as a function of b and r for $n = 256$. The minimum energy is obtained around $r = 16$ and $b = 16$ while the latency is $2743.5 \mu\text{sec}$. Note that the energy optimal design runs 5 times longer than the latency optimal design. Figure 3 (b) shows the energy distribution over four operations when $n = 256$ and b varies. When $b = 16$, we achieve the minimum energy design and $opMMS$ is the dominant source of energy dissipation. Through design space exploration, we found that the energy efficient designs are obtained when $b = r = 16$ for $n \geq 32$. Another interesting results are the energy distribution on the core operation, MAC (multiply-and-accumulate) and the rest of operations. In Figure 3 (c), approximately 50% of the total energy is used by MAC, which is relatively high compared with the a general purpose processor or DSP processor.

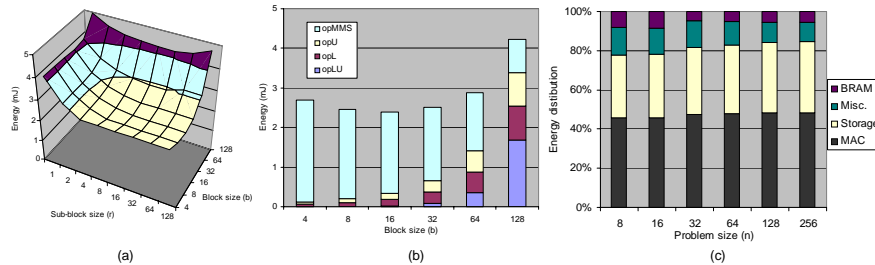


Fig. 3. (a) Energy dissipation as function of b and r for $n = 256$, (b) energy distribution as function of b for $n = 256$, and (c) energy distribution as function of n

4 Design Synthesis and Simulation Results

To obtain time and energy efficient designs, we briefly discuss the optimization techniques used in our designs. Then the synthesized designs for various problem sizes and the results from low level simulations are presented.

4.1 Optimizations for Time and Energy Efficiency

In this section, we summarize the energy efficient design techniques [7] employed in our designs. First, we have chosen a linear array of PEs. In FPGAs, long wires dissipate a significant amount of power [10]. For energy efficient designs, it is beneficial to minimize the number of long wires using a linear array since each PE communicates only with its nearest neighbors. Additionally, the linear array architecture facilitates the use of *parallel processing* and *pipelining*. Both techniques decrease the effective latency of a design and can lead to lower energy dissipation. Another technique is *block disabling*. We design the algorithm such that it utilizes the clock gating technique [15] to disable modules that are not in use during the computation. In our designs, since *opMMS* takes longer time than other operations, a set of PEs for *opLU*, *opL*, and *opU* becomes idle and is disabled to save energy. Another technique is choosing the appropriate *bindings*. In the Xilinx Virtex-II, the storage LU can be implemented as registers, distributed RAM, or embedded Block RAM. When the number of entries > 64 , Block RAM is used since it is energy efficient for large memory; otherwise, distributed RAM is used. Similar decisions can be made such as choosing between (embedded) Block multiplier or configured multiplier. We choose Block multiplier since it is energy efficient when both inputs are not constant. To implement the division unit, a lookup table approach is used. This technique is faster and uses less energy compared with other division algorithms [11]. To calculate a/b , we first obtain $1/b$ via a lookup table and perform the multiplication $a \times (1/b)$. The approach is effective if the multiplication is fast. Using Block multipliers, fast multiplication (within one cycle) can be performed. The lookup table for reciprocal is generated as $Inv(b) = Round(2^m/b)$ where b is the value to be inverted, m is the number of bits used to represent the output, and $Round$ is the rounding function.

4.2 Simulation Results

Using the performance models defined in Section 3, we identified the energy and time efficient designs based on the parameters. By considering different criteria such as area, latency, and energy, we identified several designs. The minimal energy designs are chosen as candidate designs and are implemented in VHDL. The precision of all designs was 16 bits. These designs were synthesized using XST in Xilinx ISE 4.2i and the frequency achieved was 120 MHz. The place-and-route file as an .ncd file was obtained for the Virtex-II XC2V1500 bg575-6 device. The input test vectors for the simulation were randomly generated such that their average switching activity was

50%. Mentor Graphics ModelSim 5.6b was used to simulate the designs and generate the simulation results as a .vcd file. These .vcd and .ncd files were then used by the Xilinx XPower tool to evaluate the average power dissipation. Energy dissipation was obtained by multiplying the average power by latency. We also compared estimates from Section 3 against actual values based on implemented designs to test the accuracy of the performance estimation. We observed that the energy estimates (See Table 2) were within 10% of the simulation results. The the average power dissipation of the designs on Virtex-II included the quiescent power of 150 mW (from XPower).

Table 2. Performance of the designs based on Corollary 1 (from low-level simulation)

n	TI DSP (600MHz)			DK1 based design (50MHz)			Our design (120MHz)						
	b	L (usec)	E (uJ)	b	L (usec)	Slice/Mult	E_m (uJ)	b	L (usec)	Slice/Mult	E_{est} (uJ)	E_m (uJ)	Err in E_{est}
8	8	2.9	4.2	4	9.6	810/6	2.52	8	0.5	835/8	0.27	0.29	6.8%
16	16	7.2	10.6	4	35.0	810/6	10.27	8	2.1	1955/16	1.3	1.4	7.1%
32	16	31.5	46.6	4	218.2	810/6	42.86	16	10.7	3550/32	8.6	8.2	6.8%
64	16	152.7	229.7	8	1004.8	1318/10	342.7	16	51.2	3550/32	52.3	50.0	5.3%
128	16	836.4	1282.1	8	7087.9	1318/10	1485.7	16	345.6	3550/32	368.8	355.8	4.3%
256	16	5171.3	8068.0	8	56157	1318/10	6863.5	16	2743.5	3550/32	2801.6	2717.2	3.7%
512	16	35335	55857	8	453583	1318/10	34825	16	22421	3550/32	21927	21330	3.3%
1024	16	258619	412251	8	3658982	1318/10	198275	16	182473	3550/32	173710	169236	3.2%

* n is problem size. b is block size. L is latency. Slice is area. Mult is the number of embedded multipliers.

E_{est} is the estimated energy using the performance model. E_m is the measured energy from low level simulation.

We were not aware of any prior FPGA based designs for LU decomposition. Hence, for the sake of comparison, we implemented two baseline designs: one on FPGAs using a state-of-the-art commercial compilation tool and the other, a software implementation on state-of-the-art TI DSPs. The FPGA design was implemented using Handel-C and synthesized using Celoxica DK1.1 [4]. The compilation tool can automatically exploit the parallelism of the algorithm. The synthesized design, with a frequency of 50 MHz, was then implemented with the Xilinx ISE 4.2i. Our designs dissipated 8.8x to 1.2x less energy than the Handel-C based designs.

We also compared the performance of LU decomposition on FPGAs and DSPs. FPGAs are known to be better than DSPs in terms of time and energy performance. Since many target applications for DSP devices and FPGAs are similar, comparing their time and energy performance is beneficial to designers. We chose the TI TMS320C6415 running at 600 MHz as a representative DSP. TMS320C6415 is a high performance DSP and has eight 16-bit MAC units. The LU decomposition was implemented in C and its precision was 16 bits. The matrix multiplication was performed using the function call `DSP_mat_mul` from the TI DSP library. The latency was obtained by using the TI Code Composer 2.1. To compute the energy dissipation, we assumed the 75% high / 25% low activity category of power dissipation for the function call `DSP_mat_mul` since it is a hand-optimized code [13]. The power dissipation for the rest of C code is based on the 50% high / 50% low activity category since the code is optimized by the TI compiler. For the DSP, we chose the block size b , $0 < b < \min(n, 16)$ so as to minimize the energy dissipation. As seen from the results in Table 2, our FPGA implementations perform LU decomposition faster using less energy. While we used the high performance DSP processor, TI also provides low power devices, namely the TMS320VC55xx series. Based on the datasheets, the 55xx series dissipate 150 mW at 300 MHz while the 64xx series dissipate 1500 mW at 600 MHz. The 55xx series have two MACs (600 MIPS) while the 64xx series have eight MACs (4800 MIPS). Thus the scaling factor from 64xx series to 55xx series for energy dissipation can be defined as:

$s_e = \frac{P_{55xx}}{P_{64xx}} \times \frac{MIPS_{64xx}}{MIPS_{55xx}} = 0.78$. By applying this scaling factor, the energy dissipation of our designs was determined to be 12.1x to 1.8x less than the TI 55xx series.

5 Conclusion

We developed time and energy efficient designs for LU decomposition on FPGAs. Before implementing the designs, we analyzed the architecture and algorithm to understand the design trade-offs. After pruning the design space, selected designs were implemented using VHDL in the Xilinx ISE design environment. Currently, state-of-the-art FPGAs (e.g., Virtex-II/pro) do not provide low power features such as control for multiple power states or lower static power. The proposed architectures and algorithms are parameterized based on several design parameters. Hence, when more features such as dynamic voltage scaling with dynamic frequency scaling are available, the operations *opL* and *opU* can be executed slower than the operation *opMMS*. This might provide the opportunity to use a lower frequency and lower voltage.

6 Acknowledgements

This work is supported by the DARPA Power Aware Computing and Communication Program under contract F33615-C-00-1633 monitored by Wright Patterson Air Force Base and in part by the National Science Foundation under award No. 99000613. The authors wish to thank Gokul Govindu, Ronald Scrofano, and Zack Baker for helpful discussions and contributions to the low level simulation results.

References

1. Altera Corporation. <http://www.altera.com>. 2002.
2. J. Becker, T. Pionteck, M. Glesner. DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications. *FPL*, 2002.
3. E. Casseau, D. Degrugillier. A Linear Systolic Array for LU Decomposition. *VLSI Design*, 1994.
4. Celoxica Corporation. DK1.1 Design Suite. <http://www.celoxica.com>. 2003.
5. J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, R. C. Whaley. The Design and Implementation of the Scalapack LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.
6. S. Choi, J. Jang, S. Mohanty, V. K. Prasanna. Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures. *ERSA*, 2002.
7. S. Choi, R. Scrofano, V. K. Prasanna, J.-W. Jang. Energy Efficient Signal Processing using FPGAs. *Field Programmable Gate Array*, 2003.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*, McGraw-Hill, 2nd edition, 2001.
9. S. Haykin. *Adaptive Filter Theory*, Prentice Hall, 4th edition, 2002.
10. L. Shang, A. Kaviani, K. Bathala. Dynamic Power Consumption in Virtex-II FPGA Family. *Field Programmable Gate Arrays*, 2001.
11. N. Shirazi, A. Walters, P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. *FCCM*, 1995.
12. H. Styles, W. Luk. Customising Graphics Application: Techniques and Programming Interface. *Field Programmable Custom Computing Machines*, 2000.
13. Texas Instruments. TMS320C64xx Power Consumption Summary, <http://www.ti.com>.
14. W. Tuttlebee. *Software Defined Radio: Enabling Technologies*, J. Wiley, 2002.
15. Xilinx Incorporated. <http://www.xilinx.com>.