

# HIGH-PERFORMANCE AND PARAMETERIZED MATRIX FACTORIZATION ON FPGAS

Ling Zhuo and Viktor K. Prasanna

Department of Electrical Engineering  
University of Southern California  
Los Angeles, USA  
{lzhuo, prasanna}@usc.edu

## ABSTRACT

FPGAs have become an attractive choice for scientific computing. In this paper, we propose a high performance design for LU decomposition, a key kernel in many scientific and engineering applications. Our design achieves the optimal performance for LU decomposition using the available hardware resources. The design is parameterized. Thus, it can be easily adapted to various hardware constraints. Experimental results show that our design achieves high performance and offers good scalability. Our implementation on a Xilinx Virtex-II Pro XC2VP100 achieves superior sustained floating-point performance over existing FPGA-based implementations and optimized libraries on the state-of-the-art processors.

## 1. INTRODUCTION

With the rapid advances in technology, FPGAs have become an attractive choice for high-performance implementations of scientific computing applications [1]. Recently, FPGAs have been used in high-end computers as application-specific accelerators for processors [2]. Example computers include SRC 6E [3] and Cray XD1 [4], among others. In these computers, an application is partitioned so that the control-intensive part is executed on the processors while the computation-intensive part runs on the FPGAs.

As FPGAs gain popularity in high-end computing, FPGA-based designs for key kernels in scientific applications have become necessary. These kernels, such as the routines in LAPACK (Linear Algebra PACKage) [5], dominate the performance of many scientific and engineering applications. Designs for them not only need to achieve high performance, but also need to consider the various hardware resource constraints on the FPGAs.

Matrix factorization is a key computational kernel in linear algebra. It is widely used to solve dense linear equations and is included in LAPACK [5]. LU decomposition is the

most used matrix factorization method. Some FPGA-based designs have been proposed for this kernel [6, 7]. However, they either do not consider the hardware constraints of the FPGA device, or do not achieve optimal performance.

In this paper, we propose a design for LU decomposition which achieves optimal performance using the available hardware resources. The design is parameterized so that it can easily adapt to the hardware constraints. The parameters include the number of PEs (Processing Elements), the storage size, and the block size. Our design employs a circular linear array of  $p + 1$  PEs. The first PE contains a floating-point divider, while each of the other PEs contains a floating-point multiplier and a floating-point adder/subtractor.  $p$  is independent of problem size  $n$ , and is determined by the available resources on the FPGA. The design requires a storage of size  $O(n^2)$  words. For larger matrices, block LU decomposition is employed where our design serves as a module. The block size  $b$  is determined by the size of the on-chip memory on the FPGA.

We implemented our algorithms using Xilinx ISE 7.1i [8], with Xilinx Virtex-II Pro XC2VP100 as our target device. Although our work is independent of the data representation, we used IEEE double-precision (64-bit) numbers in our experiments as they are used in most scientific computations. We used our own floating-point units which are partly compliant with IEEE-754 standard [9]. Experimental results show that our design achieves about twice the performance of previous work. Moreover, the performance of our design scales with the available hardware resources. Our design is device-independent and achieves about 4 GFLOPS using one XC2VP100. This performance is higher than that of the highly optimized LU decomposition algorithm from AMD Core Math Library (ACML) [10] on a 2.2 GHz Opteron processor. We also predict the performance of our design when smaller and faster floating-point units are available.

The paper is organized as follows. Section 2 discusses the background and the related work. Section 3 discusses the design issues for FPGA-based LU decomposition, and describes our proposed design. Section 4 presents our experimental results. Section 5 concludes the paper.

---

Supported by the United States National Science Foundation under grant No. CCR-0311823 and in part by No. ACI-0305763. Equipment grant from Xilinx Inc. is gratefully acknowledged.

## 2. RELATED WORK

LU factors an  $n \times n$  matrix  $A$  into an  $n \times n$  lower triangular matrix  $L$  (the diagonal entries are all 1) and an  $n \times n$  triangular matrix  $U$ . We use  $a_{xy}$ ,  $l_{xy}$  and  $u_{xy}$  to denote the elements of  $A$ ,  $L$  and  $U$ , respectively ( $0 \leq x, y \leq n - 1$ ). We assume that  $A$  is non-singular and no pivoting is needed. In [11], a sequential algorithm for LU decomposition is discussed. It consists of three main steps:

**Step 1:** The column vector  $a_{x0}$  ( $1 \leq x \leq n - 1$ ) is multiplied by the reciprocal of  $a_{00}$ . The resulting column vector is  $l_{x0}$ .  $u_{0y}$  equals  $a_{0y}$  ( $1 \leq y \leq n - 1$ ).

**Step 2:**  $l_{x0}$  is multiplied by the row vector  $u_{0y}$ . The product is subtracted from the submatrix  $a_{xy}$  ( $1 \leq x, y \leq n - 1$ ).

**Step 3:** Steps 1 and 2 are recursively applied to the new submatrix generated in Step 2. During the  $k$ th iteration,  $l_{xk}$  and  $u_{ky}$  ( $k + 1 \leq x, y \leq n - 1$ ) are generated. When  $k = n - 1$ , the decomposition is complete.

There have been few works on FPGA-based matrix factorization. Choi et al [12] proposed a linear array architecture for *fixed-point* LU. The array consists of  $n$  PEs and each PE contains a multiplier, an adder/subtractor and a reciprocator. The latency of this architecture is  $n^2$ . However, using one reciprocator in every PE is very resource-consuming if the architecture is used for floating-point LU. In [13], Wang et al proposed a multiprocessor system on an FPGA device, and used it for LU decomposition of sparse block-diagonal-bordered matrices. Each processor on the FPGA is attached to a floating-point adder/subtractor, a floating-point multiplier and a floating-point divider. As a FPGA-based divider occupies a large number of slices, the number of processors in such a system is highly limited.

In [6], a circular array architecture was proposed for floating-point LU decomposition. It uses  $n$  PEs, and only the first PE,  $PE_0$ , contains a divider.  $PE_j$  ( $1 \leq j \leq n - 1$ ) contains one MAC (Multiplier and ACcumulator) and a storage of size  $n - j$ .  $PE_j$  is in charge of computations that involve column  $j - 1$ . The latency of the design is  $n^2$ , and the required storage size is  $\frac{n^2}{2}$ . However, due to the design complexity of the floating-point units, an FPGA device cannot contain  $n$  PEs when  $n$  is larger than several tens.

[7] gives a detailed description of block LU decomposition algorithm and proposes an architecture for it. The matrix is partitioned into  $b \times b$  blocks, where  $b$  is determined by the number of configurable slices. The latency of the design in [7] is approximately  $\frac{n^3}{3b}$  cycles. As discussed in Section 3, the latency of [7] is about twice the minimum latency of LU decomposition with the given hardware resources.

In this paper, we propose a design in which the number of PEs is independent of the problem size. This design also achieves higher performance than the existing work, and can serve as a module in block LU decomposition. Note that

throughout this paper, the *latency* of a design is the number of clock cycles needed to complete LU decomposition.

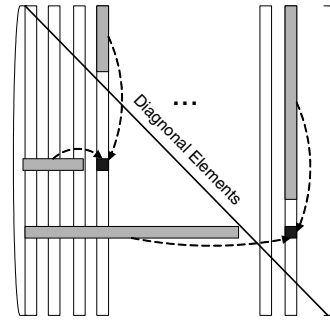
## 3. MATRIX FACTORIZATION ON FPGAS

### 3.1. Design Issues

We first propose an extension for the design in [6], which is denoted as Design 1. Based on this extension, we discuss several design issues in FPGA-based LU decomposition.

In Design 1, there are  $p + 1$  PEs connected in a circular linear array as in [6]. The first PE  $PE_0$  contains a floating-point divider, while each of the other PEs contains a floating-point multiplier and adder/subtractor. Matrix  $A$  is fed into the architecture in column major. As  $a_{x0}$  travels through  $PE_0$ , it is divided by  $a_{00}$  and the resulting  $l_{x0}$  is stored in  $PE_1$  ( $1 \leq x \leq n - 1$ ).  $a_{01}$  ( $= u_{01}$ ) multiplies with  $l_{x0}$ , and the product results are subtracted from  $a_{x1}$  ( $1 \leq x \leq n - 1$ ). The subtract results traverse the circular linear array back to  $PE_0$ , and then are divided by  $u_{11} = a_{11} - a_{01} \times l_{10}$ . The resulting  $l_{x1}$  is stored back in  $PE_1$  ( $2 \leq x \leq n - 1$ ). In particular,  $PE_j$  stores columns  $\frac{n}{p} \times (j - 1)$ ,  $\frac{n}{p} \times (j - 1) + 1$ ,  $\dots$ ,  $\frac{n}{p} \times (j - 1) + \frac{n}{p} - 1$  of  $L$  ( $1 \leq j \leq p$ ).

As column  $k$  of matrix  $A$  traverses the PEs, Design 1 calculates the  $k$ th columns of  $L$  and  $U$ . To generate  $u_{qk}$  ( $q < k$ ), the row vector  $l_{qy}$  is multiplied with the column vector  $a_{xk}$  ( $0 \leq x = y \leq q$ ). The result of the dot product is then subtracted from  $a_{qk}$ . To generate  $l_{qk}$  ( $q \geq k$ ), the row vector  $l_{qy}$  is multiplied with the column vector  $a_{qk}$  ( $0 \leq x = y \leq k$ ); the result of the dot product is then subtracted from  $a_{qk}$ . The resulting  $a'_{qk}$  is then divided by  $u_{kk}$ . The computation flow is shown in Figure 1.



**Fig. 1.** Computational flow for Design 1

Design 1 stores  $n - k$  words for column  $k$  of  $L$  ( $1 \leq k \leq n - 1$ ). Thus, it requires a storage of  $\frac{n^2}{2}$  words. However, as Design 1 mainly executes  $\sum_{x=0}^{\min(q,k)} l_{qx} a_{xk}$ , multiply results are generated serially and need to be accumulated in each PE. Moreover, if two neighboring PEs perform a dot product together, their intermediate results also need to be accumulated. In this case, when pipelined multipliers

and adders/subtractors are used, read-after-write data hazards may arise. To avoid such hazards, we can interleave the computations of multiple matrices, that is, use *stacking* matrices. In this case, the storage size required by the design is increased to  $\frac{n^2 s}{2}$ , where  $s$  equals the total latency of the multiplier and the adder/subtractor. Thus, the first design issue for FPGA-based LU decomposition is to avoid data hazards without greatly increasing the storage requirement.

The second design issue is to effectively utilize the parallelism provided by the architecture. In Design 1,  $PE_j$  stores  $\frac{n}{p}$  consecutive columns of  $L$ . Therefore, when the first  $\frac{n}{p}$  columns of  $A$  traverses the PEs, only  $PE_1$  is performing floating-point operations while the other PEs are idling; for the next  $\frac{n}{p}$  columns of  $A$ , only  $PE_1$  and  $PE_2$  are working, and so on. Only for the last  $\frac{n}{p}$  columns, all  $p$  PEs are working in parallel. The latency of Design 1 is approximately  $\frac{2n^3}{3p}$  cycles, when  $n > p$ . As we show later, lower latency can be achieved if the computational parallelism of the PEs is fully exploited.

### 3.2. Proposed Design

Based on the analysis in the previous section, we propose a second design, Design 2. This design parallelizes the sequential algorithm described in Section 2. In our following discussion, the pipeline latencies of the floating-point multiplier, adder/subtractor, divider are  $l_1, l_2$  and  $l_3$ , respectively.

The architecture of Design 2 is shown in Figure 2. It contains a circular linear array of  $p + 1$  PEs. The PEs are labeled as  $PE_0, PE_1, \dots, PE_p$  from left to right. Each PE has two input ports and two output ports. These ports are denoted as  $inL, outL, inU$  and  $outU$ , respectively. Each PE is only connected to its neighboring PEs, while  $PE_p$  is connected to  $PE_{p-1}$  and  $PE_0$ . Initially matrix  $A$  is stored in the external memory. When LU decomposition is complete, matrices  $L$  and  $U$  are written back to the memory.

$PE_0$  contains a floating-point divider and a storage of size  $n$  words to store  $u_{xx}$  ( $0 \leq x \leq n - 1$ ). This storage is denoted as  $S_0$ .  $PE_j$  ( $1 \leq j \leq p$ ) contains a floating-point multiplier and a floating-point adder/subtractor. There are two storages in  $PE_j$ :  $S1_j$  is of size  $\frac{(n-1)^2}{p}$  and stores the intermediate results which are denoted as  $a'_{xy}$ ;  $S2_j$  is of size  $\frac{n-1}{p}$  and stores  $u_{ky}$  in the  $k$ th iteration ( $1 \leq x \leq n - 1, y = j, j + p, \dots, j + (\frac{n-1}{p} - 1)p$ ). For simplicity, we assume  $n - 1$  is a multiple of  $p$ .

The algorithm of Design 2 contains  $n - 1$  iterations. During each iteration, the architecture goes through three stages. In **Stage 1** of the  $k$ th iteration,  $a_{k,k+1}, a_{k,k+2}, \dots, a_{k,n-1}$  traverse the linear array ( $0 \leq k \leq n - 2$ ) through ports  $inU$  and  $outU$ .  $PE_j$  performs  $u_{k,y} = a_{k,y} - a'_{k,y}$ , where  $y = j, j + p, \dots, j + (\frac{n-1}{p} - 1)p$ . The resulting elements of  $U$  is stored in  $S2_j$  ( $1 \leq j \leq p$ ) and will be used in Stage 3.

**Stage 2** is only needed when  $k > 0$ . In this stage,  $a_{k,k},$

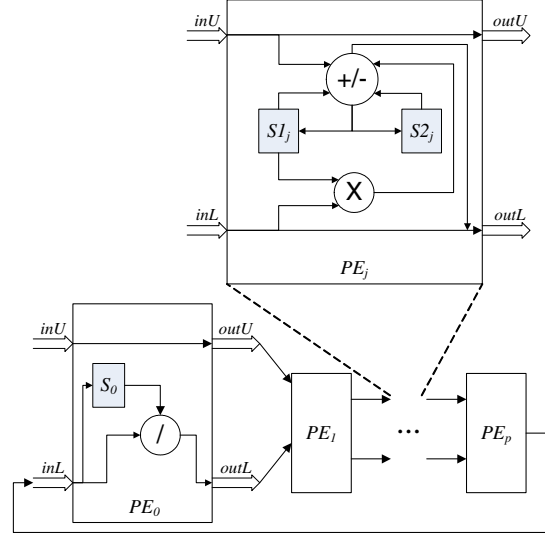


Fig. 2. Architecture of Design 2

$a_{k+1,k}, \dots, a_{n-1,k}$  enters the architecture through port  $inU$ , and  $a'_{x,k} = a_{x,k} - a'_{x,k}$  are performed ( $k \leq x \leq n - 1$ ). These  $a'_{x,k}$  traverse along the circular array through ports  $outL$  and  $inL$ . After they reach  $PE_0$ , they are divided by  $a'_{k,k}$  ( $= u_{k,k}$ ) to generate  $l_{x,k}$ . These elements of  $L$  are then used in the next stage. Note that when  $k = 0, l_{x0} = a_{x0}$  ( $1 \leq x \leq n - 1$ ).

During **Stage 3**,  $l_{k+1,k}, l_{k+2,k}, \dots, l_{n-1,k}$  are fed into  $PE_1$  through port  $inL$ . Each of them is multiplied with the elements stored in  $S2_j$  in  $PE_j$  ( $1 \leq j \leq p$ ). The intermediate results in  $S1_j$  are updated using  $a'_{xy} = a'_{xy} + l_{x,k} \times u_{k,y}$  ( $k + 1 \leq x, y \leq n - 1$ ). The computation flow of Design 2 is shown in Figure 3.

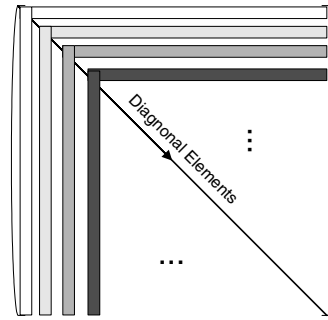


Fig. 3. Computational flow for Design 2

**Theorem 1** Design 2 completes LU decomposition in approximately  $\frac{n^3}{3p}$  clock cycles without causing any data hazard. The required storage size is  $(n - 1)^2 + n$  words.

**Proof.** The size of the storage can be derived from the description of the architecture. The latency is calculated as follows:

In Stage 1 of iteration 0,  $u_{0,n-1}$  arrives at  $PE_p$  at clock cycle  $(n - 1 + p)$ . Thus, Stage 1 finishes at clock cycle

$(n - 1 + p + l_2)$ , when  $u_{0,n-1}$  is generated. In Stage 3 of iteration 0,  $l_{x0}$  needs to be multiplied with  $\frac{(n-1)}{p}$  elements in each PE ( $1 \leq x \leq n - 1$ ). Thus,  $l_{n-1,0}$  reaches  $PE_p$  after  $\frac{(n-1)^2}{p}$  clock cycles, and  $a'_{n-1,n-1}$  is generated after  $(l_1 + l_2)$  more clock cycles.

Similarly, in iteration  $k$ , Stage 1 takes  $(n - k - 1 + p + l_2)$  cycles and Stage 2 takes  $(n - k - 1 + p + l_2 + l_3)$  cycles. Note that there is no dependency between stages 1 and 2. Thus, Stage 2 can start right after  $a_{k,n-1}$  enters the architecture. In this case, these two stages altogether need  $(2(n - k - 1) + p + l_2 + l_3)$  cycles. In Stage 3 of iteration  $k$ ,  $l_{xk}$  needs to be multiplied with at most  $\lceil \frac{(n-k-1)}{p} \rceil$  elements in each PE ( $k + 1 \leq x \leq n - 1$ ). Thus, Stage 3 takes  $(\lceil \frac{(n-k-1)}{p} \rceil)(n - k - 1) + l_1 + l_2$  cycles.

Therefore, the total latency of Design 2 =  $(n - 1 + p + l_2) + \sum_{k=1}^{n-1} (2(n - k - 1) + p + l_2 + l_3) + \sum_{k=0}^{n-1} (\lceil \frac{(n-k-1)}{p} \rceil)(n - k - 1) + l_1 + l_2$ . Suppose  $\lceil \frac{(n-k-1)}{p} \rceil = \frac{n-k-1}{p} + 1$  for all  $k$ . The latency is at most  $\frac{n}{6p}(n - 1)(2n - 1) + (n - 2)(n - 3) + (l_1 + 2l_2 + l_3 + p + 2)n + l_2 - l_3 \approx \frac{n^3}{3p}$  cycles, when  $n > p$ .

Note that in Design 2 each stage only starts when all the results of the previous stage are generated (except stages 1 and 2). Thus, there is no data hazard in the design. ■

By overlapping the computations in Stage 3 and Stage 1, we can further reduce the latency of Design 2. However, when  $(\lceil \frac{(n-k-1)}{p} \rceil)(n - k - 1)$  is smaller than  $l_1 + l_2$ ,  $a'_{k+1,k+2}$  may be read before it is correctly updated. To avoid such data hazard, zero padding is needed so that Stage 3 at least takes  $l_1 + l_2$  clock cycles. In this case, the latency of Design 2 can be reduced but is still about  $\frac{n^3}{3p}$  cycles. As the design needs to read  $n^2$  words from and write  $n^2$  words to the memory, the required memory bandwidth is approximately  $\frac{6p}{n}$  words/clock cycle.

**Theorem 2** *Design 2 achieves the optimal performance of LU decomposition with  $p$  multipliers and  $p$  adders/subtractors, when  $p < n$ .*

**Proof.** According to the algorithm description in Section 2,  $(n - 1 - k)^2$  multiplications and subtractions and  $(n - 1 - k)$  divisions are performed during the  $k$ th iteration. Therefore,

LU decomposition requires totally  $\sum_{k=0}^{n-1} (n - 1 - k) \approx \frac{n^2}{2}$  divisions, and  $\sum_{k=0}^{n-1} (n - 1 - k)^2 \approx \frac{n^3}{3}$  multiplications and additions/subtractions each.

When  $p \leq n$ , the latency of any design for LU decomposition is bounded by the time needed to complete the multiplications and additions/subtractions. Suppose  $p$  multipliers and  $p$  adders/subtractors work in parallel during each clock cycle, the lower bound on the latency of LU decomposition

is approximately  $\frac{n^3}{3p}$  cycles. Thus, our design achieves the minimum latency, and hence the optimal performance of LU decomposition, when  $p < n$ . ■

### 3.3. Block LU Decomposition

In Design 2, the required storage size  $m = (n - 1)^2 + n$  words. Suppose the available size of the on-chip memory on the FPGA device is  $M$  words. When  $m > M$ , block LU decomposition algorithm needs to be used. In this algorithm, matrix  $A$  is partitioned into blocks of size  $b \times b$ . The computation flow of the algorithm is similar to the algorithm described in Section 2, except that the basic computational unit is now a block instead of a word.

Design 2 can be used in the architecture for block LU decomposition proposed in [7]. In this architecture, there are one set of  $p_1$  PEs for  $b \times b$  LU decomposition, one set of  $p_2$  PEs for  $b \times b$  matrix multiplication and a single PE for matrix subtraction. Both the PE sets for LU and matrix multiplication require a storage of size  $b^2$  [2]. Therefore,  $b = \sqrt{M/2}$ .

During block LU decomposition, the PEs for LU are used for about  $\frac{n^2}{b^2}$  times and the PEs for matrix multiplication are used for about  $\frac{n^3}{3b^3}$  times. Thus, the latency of block LU decomposition is determined by the latency of matrix multiplication, and is approximately  $\frac{n^3}{3p_2}$  cycles. In [7],  $p_1 = p_2 = b$ . The required memory bandwidth is  $\frac{6p_2}{n}$  words/clock cycle. When Design 2 is used for LU in [7],  $p_1$  and  $p_2$  are independent of  $b$ .

### 3.4. Algorithm Analysis

Design 2 is characterized by various parameters. The number of PEs  $p + 1$  is determined by the number of available configurable slices. The storage size  $m$  depends on the matrix size, while the block size  $b$  is determined by the size of on-chip memory on the device. Thus, by changing the values of the parameters, Design 2 can adapt to various matrices and various hardware constraints.

In Table 1, we compare the performance of Design 1, Design 2 and the existing work. In the table,  $s = l_1 + l_2$ . The value of  $b$  in [7] and the value of  $p$  in Design 2 are both determined by the number of slices on the device. As the design in [7] needs  $2b$  PEs, the latency of Design 2 is approximately half of that of [7]. For larger  $n$ , block LU algorithm using Design 2 can also achieve higher performance because the value of  $p_2$  can be larger than the value of  $b$  in [7].

## 4. EXPERIMENTAL RESULTS

Our target device is Xilinx Virtex-II Pro XC2VP100, which contains 44,096 slices and about 1 MB on-chip memory. We used Xilinx ISE 7.1i and Mentor Graphics ModelSim 6.0a

**Table 1.** Algorithm Comparison

	Design 1	Design 2	[7]	Block LU with Design 2
No. of PEs	$p + 1$	$p + 1$	$b + b$	$p_1 + p_2$
Latency	$\frac{2n^3}{3p}$	$\frac{n^3}{3p}$	$\frac{n^3}{3b}$	$\frac{n^3}{3p_2}$
Stacking	Yes	No	Yes	No
Storage	$\frac{sn^2}{2}$	$n^2$	$\frac{(s+1)b^2}{2}$	$2b^2$

development tools [8, 14]. In our experiments, we used our own 64-bit floating-point adders and multipliers that comply with IEEE-754 format. Their implementation details can be found in [15]. Table 2 gives the characteristics of the floating-point units.

**Table 2.** 64-bit Floating-Point Units

	Adder	Multiplier	Divider
No. of Pipeline Stages	11	8	58
Area (slices)	892	835	3213
Clock Speed(MHz)	170	170	140

We have implemented Design 2 on the target device. The characteristics of the PEs are shown in Table 3.  $PE_0$  contains a floating-point divider and a control unit that generates the control signals for the other PEs. Thus, the area of  $PE_0$  is much larger than the other PEs. The clock speed of  $PE_0$  is bounded by that of the divider.  $PE_j$  ( $1 \leq j \leq p$ ) consists of one floating-point multiplier and adder/subtractor, as well as multiple 64-bit multiplexers that select the inputs to the adder/subtractor. The local storage in the PEs are implemented using Block RAMs (BRAMs). Therefore, the size of the storage does not affect the area of the PEs.

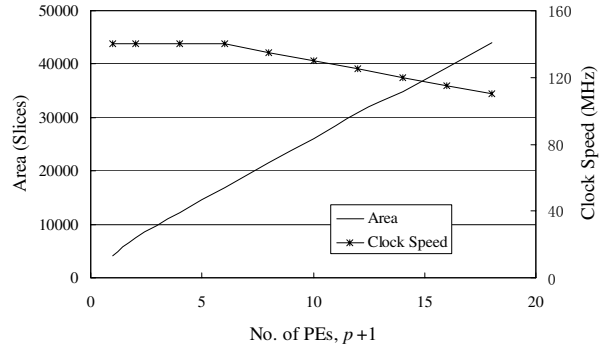
**Table 3.** Characteristics of PEs in Design 2

	$PE_0$	$PE_j$ ( $1 \leq j \leq p$ )
Area (slices)	4112	2136
Clock Speed(MHz)	140	160

The scalability of our design is shown in Figure 4. The number of PEs that can be configured increases almost linearly with the available resources. On the other hand, the clock speed of the design decreases with the number of PEs due to the increased routing complexity. However, the total degradation in clock speed is less than 25% when  $p + 1$  increases from 1 to 18 (Note that the speed of the entire design is bounded by that of the divider when  $p + 1 < 6$ ). Therefore, the performance of our design scales with the available hardware resources.

#### 4.1. Performance Comparison

We compare the performance of Design 2 with the design in [7], as shown in Table 4. The size of the matrices ranges from  $100 \times 100$  to  $1000 \times 1000$ . In our implementations,

**Fig. 4.** Scalability of Design 2

$p + 1 = 18$  and the clock speed is 110 MHz. In the implementations of [7],  $b = 9$  because at most 9 PEs of LU and matrix multiplication each can fit into the device. The clock speed is also 110 MHz. We see that when  $n$  is larger than 500, our work achieves about 2X speedup over the design [7]. This is because our work utilize the computational parallelism provided by the PEs more effectively. Moreover, using our design, there is no blocking overhead.

**Table 4.** Latency comparison of various designs

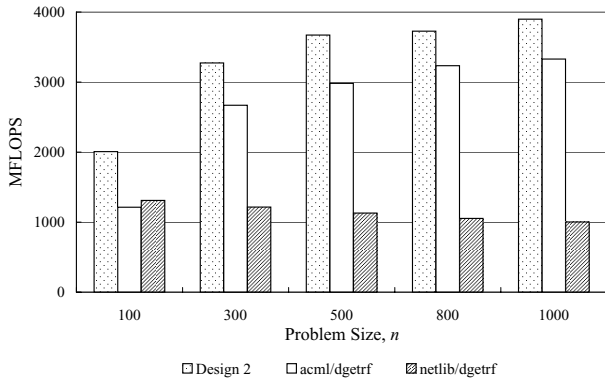
$n$	Design 2	[7]	Speedup
	Latency (ms)	Latency (ms)	
100	0.33	0.46	1.40
300	5.5	8.8	1.60
500	22.7	40.9	1.80
800	91.6	169.3	1.85
1000	171.0	331.7	1.94

We also compare our design with the general-purpose processor based designs. We use the sustained MFLOPS as the metric. It is calculated as:

$$MFLOPS = \frac{\text{Total number of floating-point operations}}{\text{Latency of the design}}$$

We consider a 2.2 GHz AMD Opteron processor with a L1 cache of 64 KB and a L2 cache of 1 MB. We executed two versions of the LU factorization routine *dgetrf*. One is from netlib LAPACK, and the other is from ACML [10]. ACML is a set of numerical routines tuned specifically for AMD64 platform processors (including Opteron and Athlon 64). The MFLOPS performance of both versions and our work is shown in Figure 5.

Figure 5 shows that our Design 2 achieves about 4X speedup over netlib *dgetrf*. Moreover, Design 2 achieves better performance than ACML *dgetrf* on AMD Opteron. Note that ACML *dgetrf* exploits optimizations specific to the architecture of Opteron processor while our design is device independent. Also, we did not utilize any manual placement to increase the performance.

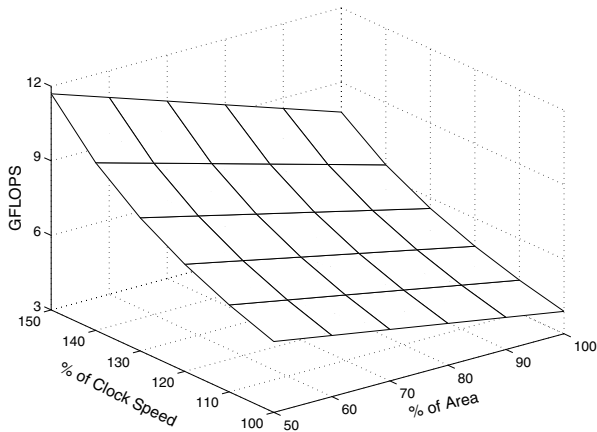


**Fig. 5.** Comparison of sustained MFLOPS performance of FPGA-based design and general-purpose processor based designs

## 4.2. Performance Projection

In Design 2, the implementations of the floating-point units have no effect on the architecture or the control logic. Therefore, when improved floating-point units are available, they can be plugged into the design easily. On the other hand, most of the area of Design 2 is occupied by the floating-point units, and its clock speed is bounded by that of the floating-point divider. Thus, if the performance of the units is improved, the performance of the design will improve accordingly. For example, we could use the floating-point units of Xilinx [8] to achieve higher performance.

Figure 6 shows the projected sustained performance of Design 2 as the area of the floating-point units reduces to 50% and the clock speed increases 50%. In this prediction, 25% of the performance is deducted to account for the degradation of the clock speed caused by the routing. We see that with XC2VP100, Design 2 can achieve up to 11.7 GFLOPS for LU decomposition.



**Fig. 6.** Projected performance of Design 2,  $n = 1000$

## 5. CONCLUSION

In this paper, we proposed a parameterized design for FPGA-based floating-point LU decomposition. By changing the values of the parameters, such as the number of PEs and the block size, we can adapt our design to various matrices and various hardware constraints. Our design achieves the minimum latency that can be reached by any LU decomposition design. The performance of the design scales with the available hardware resources. Implemented on one Xilinx Virtex-II Pro FPGA, our design achieves about 4 GFLOPS, which is higher than the sustained performance of a LU decomposition algorithm specifically optimized for a 2.2 GHz AMD Opteron processor. In the future, we are going to include partial pivoting in our design, and also work on designs that can utilize both FPGAs and processors in the reconfigurable computers for more linear algebra kernels.

## 6. REFERENCES

- [1] W. Chen and P. Kosmas and M. Leeser and C. Rappaport, "An FPGA Implementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm," in *Proc. of the FPGA 2004*, February 2004.
- [2] L. Zhuo and V. K. Prasanna, "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proc. of SuperComputing 2005*, November 2005.
- [3] SRC Computers, Inc., "<http://www.srccomp.com/>."
- [4] Cray Inc., "<http://www.cray.com/>."
- [5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK user's guide third edition," August 1999.
- [6] G. Govindu, S. Choi, and V. K. Prasanna, "Efficient Floating-Point Based Block LU Decomposition on FPGAs," in *Proc. of ERSA 2004*, June 2004.
- [7] V. Daga, G. Govindu, S. Gangadharpalli, V. Sridhar, and V. K. Prasanna, "Efficient Floating-point based Block LU Decomposition on FPGAs," in *Proc. of ERSA 2004*, June 2004.
- [8] Xilinx Incorporated, "<http://www.xilinx.com/>."
- [9] Institute of Electrical and Electronics Engineers, *IEEE 754 Standard for Binary Floating-Point Arithmetic*. IEEE, 1984.
- [10] AMD Core Math Library, "<http://developer.amd.com/acml.aspx>."
- [11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.
- [12] S. Choi and V. K. Prasanna, "Time and Energy Efficient Matrix Factorization using FPGAs," in *Proc. of FPL 2003*, September 2003.
- [13] X. Wang and S. G. Ziavras, "Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations," in *Proc. of FPT 2003*, December 2003.
- [14] Mentor Graphics Corp., "<http://www.mentor.com/>."
- [15] G. Govindu, R. Scrofano, and V. K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," in *Proc. of ERSA 2005*, June 2005.