Research Article

Sparse Cholesky Factorization on FPGA Using Parameterized Model

Yichun Sun, Hengzhu Liu, and Tong Zhou

School of Computer, National University of Defense Technology, Deyu Road No. 109, Kaifu District, Changsha, Hunan 410073, China

Correspondence should be addressed to Yichun Sun; yichunsun@126.com

Received 6 March 2017; Revised 19 August 2017; Accepted 12 September 2017; Published 17 October 2017

Academic Editor: Oleg V. Gendelman

Copyright © 2017 Yichun Sun et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Cholesky factorization is a fundamental problem in most engineering and science computation applications. When dealing with a large sparse matrix, numerical decomposition consumes the most time. We present a vector architecture to parallelize numerical decomposition of Cholesky factorization. We construct an integrated analytical parameterized performance model to accurately predict the execution times of typical matrices under varying parameters. Our proposed approach is general for accelerator and limited by neither field-programmable gate arrays (FPGAs) nor application-specific integrated circuit. We implement a simplified module in FPGAs to prove the accuracy of the model. The experiments show that, for most cases, the performance differences between the predicted and measured execution are less than 10%. Based on the performance model, we optimize parameters and obtain a balance of resources and performance after analyzing the performance of varied parameter settings. Comparing with the state-of-the-art implementation in CPU and GPU, we find that the performance of the optimal parameters is 2x that of CPU. Our model offers several advantages, particularly in power consumption. It provides guidance for the design of future acceleration components.

1. Introduction

In engineering and science computations, the solution of large sparse linear system of equations is a fundamental problem in most applications. In general, two methods for solving this problem exist: the direct and iterative methods. In the direct method, when focusing on factorization algorithms, we have LU decomposition, QR decomposition, and Cholesky decomposition [1]. It should be noted that Cholesky decomposition is a special form of LU decomposition that deals with symmetric positive definite matrices. Its computational complexity is roughly half of LU algorithm. The Cholesky algorithm is widely used in applications because most matrices involved are symmetric positive definite.

The solution of large sparse linear system of equations can be adapted quite easily to parallel architectures. Supercomputers and multiprocessor systems became the main focus of this kind of research [2–4]. However these systems cost highly in making marketable products [5].

There are a lot of software- and hardware-based approaches developed to get better solution in Cholesky

decomposition. Field-programmable gate arrays (FPGAs) have unique advantages in solving such problems. First, according to the characteristics of the algorithm, a dedicated architecture could be designed [6]. Second, the computational resources, memory, and I/O bandwidth are reconfigurable. Third, the energy consumption and cost are relatively low under the premise of meeting certain performances. Fourth, it provides experimental and verification platform for future coprocessor design. Yang et al., 2010 [7, 8], presented a Cholesky decomposition based on GPUs and FPGAs. The results show that the dedicated FPGA implementation has the highest efficiency. The design in FPGAs has been able to combine parallel floating-point operations with high memory bandwidth in order to achieve higher performance than microprocessors [9].

A sparse matrix, in which the ratio of nonzero values is low, is especially suitable for direct methods. In our implementation, we chose the Cholesky factorization method over LU decomposition.

The reason why we chose the Cholesky factorization is less operation. The matrix is symmetric and half of factors

in matrix need to be calculated and suitable for parallel architectures. In parallel, there is less communication within each interprocessor. There are a lot of work in parallelizing the Cholesky factorization [3, 4]. The memory requirement is significantly lower than other direct methods, because only the lower triangular matrix is used for factorization and the intermediate and final results (the Cholesky factor L) are overwritten in the original matrix.

The three algorithms in the implementation of Cholesky factorization are row-oriented, column-oriented, and submatrix algorithms. The difference among these three algorithms is in the order of the three loops [3]. Among these three algorithms, the column-oriented algorithm is the most popular algorithm for sparse matrix. Once each column is regarded as a node, data dependency of each node in Cholesky factorization can be described as an elimination tree [10]. In the tree, the parent nodes depend on their descendants while their descendants must update them before being decomposed. Thus, the process of factorization can be taken as a process of eliminating the tree from bottom to top. Two main operations of the algorithm exist: scaling operation in columns ($cdiv()$) and updating operation between dependent columns ($cmod()$). Basing on the updating order, the researchers found that the algorithm can be classified into left-looking and right-looking algorithms.

A number of studies in sparse Cholesky factorization optimized in GPU are found. Vuduc and Chandramowlishwaran mapped the submatrix tasks directly to the GPU [11], George scheduled the tasks on the GPU [12], and Lucas et al. used multiple CPU threads to take small-scale computing tasks in parallel and used GPU to speed-up large-scale computing tasks [13]. The small-scale computing tasks are in the bottom of the elimination tree whereas the large-scale computing tasks are in the top of the elimination tree. Lacoste et al. modified the task scheduling module of Cholesky factorization algorithm to provide a unified programming interface on CPU-GPU heterogeneous platforms [14]. Hogg et al. proposed a supernodal algorithm of HLS_MA87 solver for multicore CPU platforms [15]. Chen et al. introduced a GPU version of the CHOLMOD solver for CPU-GPU platforms [16]. Zou et al. improved the generation and scheduling of GPU tasks by queue-based approach and designed a subtree-based parallel method for multi-GPU system [17].

To decrease the storage and computation complexity, the sparse matrix would be compressed into a dense format (CSC format, Compressed Sparse Column Format). However, in the process of Cholesky factorization, many original zero entries will turn to be nonzeros. They increase the overhead memory and dynamically change the compressed data structure. To avoid this situation, the matrix should be preprocessed and decomposed in symbol before undergoing numerical factorization (when preprocessed and decomposed in symbol, we could generate nonzero pattern of matrix and its elimination tree. When we reorganize the data, we would get the vector array). Preprocessing exposes parallelism by reordering matrix columns. Symbol decomposition is then executed in advance to identify the sparsity pattern of the result matrix. With the symbol factorization technology being well-developed, numerical decomposition

is the most time consuming. This paper discusses and evaluates the performance of numerical decomposition based on the preprocessing and symbol factorization results.

After analyzing Cholesky algorithm when dealing with a large sparse matrix, we rewrite the algorithm in a vector version. In order to explore the relationship between parallel scale and performance, to predict the performance on field-programmable gate arrays (FPGAs) platform, we also designed corresponding vector architecture and construct a parameterized model. The performance depends on the underlying architecture and system parameters. The capacity of on-chip memory, the amount of computational resources, and the size of I/O bandwidth are critical parameters. Our model takes these factors into consideration and focused on their relationship. By using the model, we obtain the best performance when choosing the capacity of on-chip memory and the amount of computational resources.

The rest of this paper is organized as follows. Section 2 introduces the vector version of Cholesky algorithm. Section 3 describes our performance model in ideal, memory-bound, and computational-bound cases. Section 4 proves the consistency of the performance of our model and the implementation; then it analyzes the parameters and tries to find an optimal performance.

2. Cholesky Factorization

Cholesky factorization decomposes a symmetric positive definite matrix into $L \times L^T$ where L is a lower triangular matrix. The main operations of the algorithm are scale operation in columns ($cdiv(k)$) and update operation between dependent columns ($cmod(k, i)$). $cdiv(k)$ divides the subdiagonal elements of column k by the square root of the diagonal element in that column, and $cmod(k, i)$ modifies column k by subtracting a multiple of column i . The algorithm can be classified into left-looking and right-looking algorithm based on updating order. Right-looking algorithm performs all the updates $cmod(k, i)$ using column k immediately after the $cdiv(k)$. By contrast, left-looking algorithm accumulates all necessary updates $cmod(k, i)$ for column i until before the $cdiv(i)$.

From Algorithm 1 we can note that the parallelism is mainly among the nondata dependency columns and the elements in a column, corresponding to the parallel elimination among different path in the elimination tree (*internode parallelism*) and the parallel scaling among different subdiagonal elements in a node (*intranode parallelism*). Besides, because of the columns in dense matrix depending on each other strictly in order, the parallelism among columns only exists in sparse matrix decomposition.

In Algorithm 1, lines (2), (7), and (9) are a progressive cycle of three layers. The amount of memory access per layer to the array l is $O(n)$ (n is the scale of the matrix). The complexity of memory access is $O(n^3)$ whether it is left-looking or right-looking. For the scale of matrix increasing, the complexity is unacceptable when a platform lacks cache space and the on-chip memory is small [18]. Duff and Reid [19] proposed a multifrontal Cholesky algorithm. The method

```

Input: A
Output: L
(1) L = A
(2) for  $i = 1; i \leq n; i++$  do
(3)    $l(i, i) = \text{sqrt}(l(i, i));$                                 /* cdiv(k) */
(4)   for  $j = 1; j \leq n; j++$  do
(5)      $l(j, i) = l(j, i)/l(i, i);$ 
(6)   end
(7)   for  $k = i + 1; k \leq n; k++$  do
(8)     if  $l(k, i) \neq 0$  then
(9)       for  $p = k; p \leq N; p++$  do
(10)         $l(p, k) = l(p, k) - l(k, i) \times l(p, i);$           /* cmod(k, i) */
(11)      end
(12)    end
(13)  end
(14) end

```

ALGORITHM 1: Right-looking Cholesky algorithm.

```

Input: A
Output: L
(1) L = A
(2) for  $i = 1; i \leq N; i++$  do
(3)    $i, i_1, \dots, i_r$  are the abscissas of non-zeros  $L(:, i)$ ;
(4)    $c_1, \dots, c_s$  are the child nodes of node  $i$ ;
(5)    $U = U_{c_1} \oplus \dots \oplus U_{c_s};$  /* construct the update matrix of node  $i$  */
(6)    $F_i = \begin{pmatrix} a_{i,i} & a_{i_1,j} & \dots & a_{i_r,j} \\ a_{i_1,j} \\ \vdots \\ a_{i_r,j} & \dots & 0 \end{pmatrix} \oplus U;$  /* update front matrix: cmod */
(7)    $F_i \rightarrow \begin{pmatrix} l_{i,i} & 0 \\ l_{i_1,i} \\ \vdots & I \\ l_{i_r,i} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U_i \end{pmatrix} \begin{pmatrix} l_{i,i} & l_{i_1,i} & \dots & l_{i_r,i} \\ 0 & I \end{pmatrix}$  /* decompose  $F_i$ : cdiv */
(8) end

```

ALGORITHM 2: Multifrontal Cholesky algorithm.

reorganizes the overall factorization of a sparse matrix into a sequence of partial factorizations of dense smaller matrices. The multifrontal algorithm accumulates the update factors into a matrix. In this way, the algorithm reduces the memory access time [20] and the update operation.

Algorithm 2 shows its content. The decomposition of every column only needs to read its front matrix F and its corresponding update matrix U . The descendants updating effects are accumulating in the matrix U , thus, avoiding the access of its descendants. In Algorithm 2, $\oplus$ means the extended addition operation between two sparse matrices.

While decomposing matrix F_i , the update matrix U_i is also generated; specifically, the following equation presents the calculation of matrix U_i .

$$U_i = F_i(2:r+1, 2:r+1) - \begin{pmatrix} l_{i,i} \\ l_{i_2,i} \\ \vdots \\ l_{i_r,i} \end{pmatrix} \times (l_{i_1,i} \quad l_{i_2,i} \quad \dots \quad l_{i_r,i}). \quad (1)$$

```

Input: A
Output: L
(1) for  $j = 1; j \leq n; j++$  do
(2)    $F_j = \begin{pmatrix} a_{1,j} & \cdots & a_{i_r,j} \\ \vdots & & \\ a_{i_r,j} & & 0 \end{pmatrix}$ 
(3)   for  $i = 1; i \leq M(j); i++$  do
(4)      $F_j = F_j \oplus \mathcal{U}_{(j)} \cdot U_i;$ 
(5)   end
(6)    $L(*, j) = F_j(*, 1) / \text{sqrt}(F_j(1, 1));$ 
(7)    $M(p(j))++;$ 
(8)   for  $k = j + 1; k \leq n; k++$  do
(9)     for  $t = k; t \leq n; t++$  do
(10)       $\mathcal{U}(p(j)) \cdot U_{M(p(j))}(t, k) = F_j(t, k) - L(t, j) \times L(k, j);$ 
(11)    end
(12)  end
(13) end

```

ALGORITHM 3: Multifrontal Cholesky algorithm in a vector version.

3. Vector Version and Architecture

This section chiefly introduces our vector version of the Cholesky algorithm and its implementation on FPGA.

3.1. Vector Version of Cholesky Algorithm. In Algorithm 3, we rewrite multifrontal algorithm in a more detailed fashion. $\mathcal{U}_{(j)}$ is a storage structure, the information of a series j th node in all the update matrices $\mathbf{U}$. $M(j)$ counts the number of its update matrices, which means the amount of child nodes of the j th node. p is an array that records the corresponding parent node. For instance, $p(j)$ is the parent node of the j th node. In line (6) of Algorithm 3, the j th column of $\mathbf{L}$ is calculated by the first column of $\mathbf{F}_j$. Lines (7) to (12) work on the generation of update matrix to its parent node and store it in $\mathcal{U}_{(p(j))}$. Notably, the update matrix is symmetric; thus, only a lower part of the matrix is generated and stored.

The execution of lines (2) to (3) in Algorithm 3 is illustrated in a vector fashion. In Algorithm 4 we exploited the strip-mined technology. z is the size of $\mathbf{F}_j$ while m is the amount of lanes in our vector architecture. k is the number of child nodes of this processed node. Furthermore, $\mathbf{U}(i; *, t)$ is the t th column of update matrix from the i th child. Each updating of a column is assigned to a lane at one time. The calculation of $\mathbf{L}$ and the updating of matrix $\mathbf{U}$ are in a similar vector processing method.

3.2. Architecture. The primary architecture of vector machine contains k -independent processing elements (PEs). We divide cdiv and cmmod modules for each PE. Figure 1 gives the details of the implementation. It is mainly composed of 3 computation modules and a series of storage components. The computation modules are **Fet_num**, **Sqrt_Div**, and **Mul_Sub**. The storage components include 5 FIFOs (First Input First Output, implemented by RAM) and RAMs for increasing buffer when data is transforming. **Fet_num** gets

```

Input:  $F_j, U$ 
Output:  $F_j$ 
(1) for  $i = 1; i \leq km; i++$  do
(2)    $\text{low} = 0;$ 
(3)    $VL = \text{mod}(z, m);$ 
(4)   for  $p = 1; p \leq z/m; p++$  do
(5)     for  $t = \text{low}; t \leq \text{low} + VL - 1; t++$  do
(6)        $F_j(*, t) = F_j(*, t) + U(i; *, t);$ 
(7)     end
(8)      $\text{low} = \text{low} + VL;$ 
(9)      $VL = m;$ 
(10)  end
(11) end

```

ALGORITHM 4: Updating algorithm of front matrix F_j .

data from **F_FIFO** (it stores frontal matrix which is $\mathbf{F}_j$ in Algorithm 3) and **Uc_FIFO** (it stores update matrix which is the output $\mathbf{F}_j$ in Algorithm 4) and combines with the data and the index that is located in data to finish extended addition operation. Then the operation in **Sqrt_Div** module starts up. The operation in **Sqrt_Div** is to finish the decomposition of column, which includes division and extraction of a root. The implementation of division and extraction of a root are based on SRT algorithm (it is usually called Digit Recurrence algorithm) and they share the implementation of SRT algorithm. **Lt_FIFO** stores the results of $L(t, j)$. The output data from **Sqrt_Div** is divided into two parts: $L(*, j)$ is the final results and it is stored in **L_FIFO**. $L(k, j)$ is stored in **LRAM** for updating. Data in **LRAM** is for **Mul_Sub** to generate update matrix. **Mul_Sub** includes multiplication and subtraction operation. Algorithm 4 is implemented in this module. The initial data in Cholesky factorization is obtained

FIGURE 1: The implementation of vector architecture.

FIGURE 2: The architecture of cmod and cdiv.

from DDR memory and the results after factorization are also stored in DDR.

The primary architecture of vector machine contains k -independent processing elements (PEs). We divide cdiv and cmod modules for each PE. Figures 2(a) and 2(b) show the two architectures. Each module has m lanes. From the two figures, we could conclude the data flow in one PE as described in Algorithm 3. For the extended addition between F and U , each lane processes one column at one time. More concretely, Figure 2(a) shows when $\text{mod}(z, m) = 2$, the first two columns of F and corresponding columns in U are processed first. After that, all lanes are fully parallel most of the time. In Figure 2(b), only the first column of F needs to be processed when calculating the corresponding column of L . The rest of the columns in F need to be used when generating the updating matrix for its parent node.

4. Performance Model

Basing on the analysis of the Cholesky algorithm, we construct a performance model on FPGA platform to predict the performance of our designed vector architecture. The performance is related to the underlying architecture and its parameters. When ignoring the implementation architecture,

we can predict the peak performance with on-chip memory, computational resources, and I/O bandwidth.

Figure 3 depicts the nonzero pattern of L and its elimination tree. The elimination tree of the matrix L is defined to be the structure with $\text{size}(L, 1)$ nodes and node p is the parent of j if and only if the following is established.

$$p = \min \{i > j \mid l_{ij} \neq 0\}. \quad (2)$$

In the elimination tree, each column in the matrix is presented as a node. In Figure 3 the numbers on the right side of the elimination tree represent the levels of the corresponding nodes. The path with a dashed line is the critical path. Given the existing dependency between parent node and its child node, the amount of nodes in the critical path, which is determined by the algorithm and the nonzeros pattern of the matrix, has the least number of nodes that should be processed sequentially.

To construct the model on FPGA, we define these variables and their descriptions as listed in Table 1.

Assuming that data are initially stored in DDR3, the result will finally be written back to DDR3. Therefore, RW contains the input data, output data, and intermediate values. The intermediate values are forced to store and load in DDR3 when needed because of the small on-chip memory size. Thus

FIGURE 3: Nonzeros pattern of L and its elimination tree.

TABLE 1: Variables that we define and their descriptions.

AOs	The amount of operations for decomposing matrix A
RW	The data volume transferred between DDR3 and FPGA on-chip memory
BW	The available bandwidth between DDR3 and FPGA
OPV	Byte/Flop ratio, that is AOs/RW
T_{cdiv}	The total computing time for $cdiv$ of a PE
T_{cmod}	The total computing time for $cmod$ of a PE
T_{node}	Computing time of decomposing one node
n	The amount of nonzeros in matrix L
S_i	The amount of nodes in the i th level of the elimination tree, $i = 1, 2, \dots, n$
nz	The amount of operations for decomposing matrix A
$\eta(L_{*j})$	The amount of the nonzero entries in j th column of L
$\eta(L)$	The amount of the nonzero entries of L

$RW = \text{Input} + \text{output} + 2 \times \text{temp}$. For a given problem, input and output data volumes are fixed, only temp is determined by the problem itself and FPGA on-chip memory.

The amount of operations (AO) is determined by the amount of nonzero entries, specifically, for $N \times N$ matrix. It takes one square operation and $\eta(L_{*j}) - 1$ division operation to scale one column ($cdiv$). Given that the update and front matrices are symmetric, only the lower part of the matrix should be taken into consideration. The amount of multiplication operations is $(1/2)\eta(L_{*j}) \times (\eta(L_{*j}) + 1)$, and the AOs for addition in $cmod$ are the same. As presented in the following equation, AO is the sum of all these operations.

$$AOs = \sum_{j=1}^N \left(1 + \eta(L_{*j}) - 1 + 2 \times \frac{1}{2} \times \eta(L_{*j}) \right) \times (\eta(L_{*j}) + 1) = 2\eta(L) + \sum_{j=1}^N \eta(L_{*j})^2. \quad (3)$$

The chip resources contain on-chip memory and computational resources. The on-chip memory is the size of the memory that could be loaded immediately. It is a critical factor in RW. Moreover, computational resources determine the extent of parallel execution. In the next two subsections, we construct two different performance models for both memory- and computational-bounded cases.

4.1. Memory-Bound. Assuming FPGA owns enough computational resources and the system is totally memory-bound, the performance is determined by the memory access time; thus, the time of decomposition can be expressed as

$$T_M = \frac{RW}{BW}. \quad (4)$$

For DDR3, when the data storage format and the access pattern are determined, BW will be determined. RW is

FIGURE 4: The parallel and pipelining execution.

a function of on-chip memory and the matrices. When memory bounded, the peak performance is described as

$$P_M = \frac{\text{AOs}}{T_M} = \frac{\text{AOs}}{\text{RW}} \times \text{BW}. \quad (5)$$

4.2. *Computational-Bound.* Figure 4(a) demonstrates the decomposition process of the matrix. The matrix is shown in Figure 3 with two PEs. We note that the bottleneck is shown in the first PE and we conclude that module `cddiv` and `cmmod` can be executed in parallel; the total time is $T = \max\{T_{\text{cddiv}}, T_{\text{cmmod}}\}$.

From Figure 4(b), $(\text{clk}_{\text{sqr}} + \text{clk}_{\text{div}} + \lceil (\eta(L_{*j}) - 2)/m \rceil)$ is needed to finish cdiv when the operations are pipeline executed. Similarly, cmul will take $(\text{clk}_{\text{mult}} + \text{clk}_{\text{add}} + \lceil (\eta(L_{*j}) + 1) \times (\eta(L_{*j})/2)/m \rceil)$ clocks. clk_{sqr} , clk_{div} , clk_{mult} , and clk_{add} are the clocks that square root, division, multiplication, and addition operation take. In our implementation, our operation modules of square root, division, multiplication, and addition need 16, 27, 1, and 8 clocks, respectively. Generally, when $\eta(L_{*j}) > 8\sqrt{m}$, $\text{clk}_{\text{cmul}} > \text{clk}_{\text{div}}$, then we have

$$T = \max \{T_{\text{cdiv}}, T_{\text{cmod}}\} = T_{\text{cmod}}. \quad (6)$$

Assuming that resources are enough, T_{node} will be independent from the scale of nonzeros. They are executed in parallel and the time it takes to finish a square root, division, multiplication, and addition operation sequentially is the same. Besides that, only the time of nodes in the critical path will be counted into the amount of time. In this ideal case, the total time is $T_{\text{Cl}} = \sum_{j=1}^n \text{clk}_{\text{cmod}}(i_j)$ and the corresponding performance reaches the roofline value [21].

To be more realistic, we assume one PE contains m lanes. A total of m nonzeros exist, executed in parallel after the diagonal element being processed in one node. In this case, we assume that one PE requires b units of resources. When the computational resources are bound and FPGA available resources are v units, one FPGA can contain $k = \lfloor v/b \rfloor$ PEs. It

means that it supports $k = \lfloor v/b \rfloor$ nodes processed mostly in parallel.

Assuming that the PE, which reaches bottleneck, is assigned to process $i_1, i_2, \dots, i_r$ nodes, we indicate that the amount of clocks is $\text{clks} = \sum_{j=1}^r \text{clk}_{\text{cmod}}(i_j)$ clocks. If the FPGA working frequency is f , the total process time is $T = \text{clks}/f$.

Given that the largest number of nodes processed in parallel is k , decomposing the i th level needs time for processing $\lceil S_i/k \rceil$ nodes. For the whole elimination tree, a total of $\sum_{i=1}^n \lceil S_i/k \rceil (N_{\text{node}})$ nodes are processed sequentially at most. Consequently, when computational resources are bounded, the computation time is shown as

$$T_{C2} = \sum_{j=1}^{N_{\text{node}}} \frac{\text{clk}_{\text{cmod}}(i_j)}{f}. \quad (7)$$

$N_{\text{node}} \geq n$, which means that $T_{C2} \geq T_{C1}$. So when computational resources and data dependency are taken into consideration, the process time is described as

$$T_C = \max(T_{C1}, T_{C2}) = T_{C2}. \quad (8)$$

Furthermore, the performance is described as

$$P_C = \frac{\text{AOs}}{T_C}. \quad (9)$$

4.3. Summary. When considering I/O bandwidth and computational resources, we can conclude that the following is the performance equation according to (4), (5), (6), (7), (8), and (9).

$$\begin{aligned} P &= \min(P_C, P_M) = \min\left(\frac{\text{AOs}}{T_C}, \frac{\text{AOs}}{T_M}\right) \\ &= \text{AOs} \times \min\left(\frac{1}{T_C}, \frac{1}{T_M}\right). \end{aligned} \quad (10)$$

TABLE 2: The analysis of sparse Cholesky factorization.

Condition	Bottleneck	Performance	Conclusion
$T_C < T_M$	Chip memory, I/O bandwidth	$OPV \times BW$	Performance is proportional to I/O bandwidth and Flop/Byte ratio
$T_C > T_M$	Computational resources	$AOs \times \frac{f}{\sum_{j=1}^{N_{node}} clk_{cmod}(i_j)}$	Performance is negative correlation to N_{node} and clk_{cmod}

TABLE 3: The analysis of factors in computational-bound cases.

Factor	Calculation	Best value	Ultimate value
N_{node}	$N_{node} = \sum_{i=1}^n \left\lceil \frac{S_i}{k} \right\rceil$	$k = \max(S_i)$	$N_{node} = n$
clk_{cmod}	$clk_{mult} + clk_{add} + \left\lceil (\eta(L_{*j}) + 1) \times \frac{\eta(L_{*j})/2}{m} \right\rceil$	$m = \max \left\{ (\eta(L_{*j}) + 1) \times \frac{\eta(L_{*j})}{2} \right\}$	$T_{node} = clk_{mult} + clk_{add} + 1$

The analysis and summary are demonstrated in Table 2. Once the parameters of FPGA and matrix are ascertained, we can obtain the peak performance and bottleneck.

Specifically, when the system is computational-bound, the performance is determined by N_{node} and clk_{cmod} . Table 3 discusses the influence of these two factors. From the table, these two factors are subject to the number of PE(k) and the number of lanes(m) in PE. Greater values of k and m do not guarantee a better performance. When $k = \max(S_i)$ and $m = \max\{(\eta(L_{*j}) + 1) \times \eta(L_{*j})/2\}$, factor N_{node} and clk_{cmod} will reach their ultimate values.

5. Experimental Evaluation

The matrices used in this paper are part of the University of Florida matrix collection, which is involved in the field of electronics-magnetics, structure design, and so on. The data format is double floating-point (64 bits), needing 8 bytes per entry when using Compressed Sparse Column format and ignoring the index.

5.1. Accuracy of Performance Modeling. This paper implements a double floating-point version of sparse Cholesky algorithm on Xilinx Virtex-7 FPGA VC709 evaluation board (xc7vx485t-2ffg1761), which tightly integrates a high-performance ARM cortex with programmable logic devices. We implement this version in 4 different scales (1 PE, 1 module in 1 PE. 1 PE, 2 modules in 1 PE. 2 PE, 1 module in 1 PE. 2 PE, and 2 modules in 1 PE) to verify our performance module. The memory chip in our platform is 6.7 MB and the frequency is 200 MHz.

We adopt the following steps in our experiments: (1) predicting the performance of our model using commonly used matrices, (2) measuring the actual performance with the matrices in our implementation, and (3) assessing the consistency of predicted and measured values. The results are shown in Figure 5 for accuracy evaluation and Figure 6 for comparison of performance different rates. In Figure 5, m is the number of PEs and k is the number of modules in one PE. The percentage of relative difference between a predicted value and measured value is calculated by $(E -$

$V)/V \times 100$, where E is the predicted value and T is the measured value. There are many optimizations in hardware implementation (like DDR storage access mode, hardware configuration parameters). After optimizing the implementation and selecting some general class of matrices so they can get a better agreement, we have the following observations from our experimental data.

The accuracy of model is critical since we need to use this model to infer large-scale parallel performance. We focus on performance and the difference rate of the performance. Figure 5 shows the comparisons between the measured performance and predicted performance by our model of four classes matrix sets (shipsec1, nd24k, Trefethen_2000b, and nd3k) at different parallel scales (1 PE 1 module, 1 PE 2 module, 2 PE 1 module, and 2 PE 2 module). The measured performances are relatively stable and accurate. From [22–24] this established literature on performance model, we can conclude that the predicted and measured values have good consistency. Figure 6 presents the performance difference rates. As the scale of parallelism increased, the difference rates dropped from 10 percent to less than 5 percent.

As shown in Figure 5, the estimated performance is higher than the tested performance. We simplify data processing in the computational-bound model. Furthermore in the memory-bound model, a deviation in the bandwidth is found between DDR3 and FPGA.

5.2. Optimization. To obtain a balance in resources in FPGA and the performance, we set the memory chip to 64 MB, with m PEs and each PE having k modules.

We compare the performance of our optimum implementation with four state-of-the-art supernode algorithms. One is the HSL_MA87 solver for multicore CPU platforms [15] while the other is the GPU version of CHOLMOD solver for CPU-GPU platforms [16]. The rest are subtree-based parallel methods for CPU-GPU platforms [17]. And the factorization time obtained is shown in Table 4. We have the following conclusions in the performance comparison.

The frequency of HSL_MA87, CHOLMOD, and GPU is 2.5 GHz, 3.2 GHz, and 2.93 GHz (CPU is 2.93 GHz, GPU is 1.15 GHz in [16]). Our frequency is 200 MHz. After testing

FIGURE 5: Accuracy evaluation on different scales.

TABLE 4: Performance comparison with HSL_MA87, GPU version of CHOLMOD, and supernode.

Matrix	HSL_MA87 times (s)	CHOLMOD times (s)	One-GPU times (s)	Two-GPU times (s)	Ours
nd3k	2.02	2.92	1.27	1.12	1.96 ($m = 2, k = 256$)
nd24k	28.56	22.17	14.63	10.12	10.08 ($m = 8, k = 32$)
Trefethen_20000b	12.63	8.49	5.47	3.94	3.58 ($m = 4, k = 64$)

FIGURE 6: Comparison of performance difference rates.

matrix sets nd3k, nd24k, and Trefethen_20000b, our performance is on the same level as GPU performance. When our work is implemented as a coprocessor (i.e., when the frequency increases, our model can guide the design of coprocessor), our implementation has a greater advantage in performance.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (no. 61602496).

References

- [1] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 5, pp. 502–520, 1997.
- [2] K. Gallivan, R. J. Plemmons et al., *Parallel Algorithms for Matrix Computations*, vol. 4, SIAM, Philadelphia, PA, USA, 1990.
- [3] A. George, M. T. Heath, and J. Liu, "Parallel Cholesky factorization on a shared-memory multiprocessor," *Linear Algebra and its Applications*, vol. 77, no. C, pp. 165–187, 1986.
- [4] M. T. Heath, E. Ng, and B. W. Peyton, "Parallel algorithms for sparse linear systems," *SIAM Review. A Publication of the Society for Industrial and Applied Mathematics*, vol. 33, no. 3, pp. 420–460, 1991.
- [5] S. G. Haridas and S. G. Ziavras, "FPGA implementation of a Cholesky algorithm for a shared-memory multiprocessor architecture," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 19, no. 4, pp. 211–226, 2004.
- [6] L. Zhou, H. Liu, and J. Zhang, "Loop acceleration by cluster-based CGRA," *IEICE Electronics Express*, vol. 10, no. 16, 2013.
- [7] D. Yang, G. D. Peterson et al., "High performance reconfigurable computing for cholesky decomposition," in *Symposium on Application Accelerators in High Performance Computing (SAAHPC '09)*, 2009.
- [8] D. Yang et al., "Performance comparison of cholesky decomposition on GPUs and FPGAs," in *Proceedings of the Symposium on Application Accelerators in High Performance Computing (SAAHPC '10)*, 2010.
- [9] C. Y. Lin, H. K.-H. So, and P. H. W. Leong, "A model for matrix multiplication performance on FPGAs," in *Proceedings of the 21st International Conference on Field Programmable Logic and Applications, FPL 2011*, pp. 305–310, grc, September 2011.
- [10] J. W. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.
- [11] R. Vuduc and A. Chandramowlishwaran, "On the limits of GPU acceleration," in *Proceedings of the the USENIX Workshop Hot Topics in Parallelism ('10)*, 13 pages, USENIX Association Berkeley, Berkeley, CA, USA, 2010.
- [12] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal factorization of sparse SPD matrices on GPUs," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, pp. 372–383, Anchorage, AK, USA, May 2011.
- [13] R. F. Lucas, G. Wagenbreth, D. M. Davis, and R. Grimes, "Multifrontal computations on GPUs and their multi-core hosts," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 6449, pp. 71–82, 2011.
- [14] X. Lacoste, P. Ramet, M. Faverge, I. Yamazaki, and J. Dongarra, "Sparse direct solvers with accelerators over DAG runtimes," *HAL-INRIA*, 2012.
- [15] J. D. Hogg, J. K. Reid, and J. A. Scott, "Design of a multicore sparse Cholesky factorization using DAGs," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3627–3649, 2010.
- [16] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software*, vol. 35, no. 1, 2008.
- [17] D. Zou, Y. Dou, S. Guo, R. Li, and L. Deng, "Supernodal sparse Cholesky factorization on graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2713–2726, 2014.
- [18] O. Maslennikov, V. Lepekha, A. Sergiyenko, A. Tomas3, and R. Wyrzykowski, *Parallel Implementation of Cholesky LLT-Algorithm in FPGA-Based Processor*, vol. 496, Springer Verlag Berlin Heidelberg, Heidelberg, Germany, 2008.
- [19] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear equations," *ACM Transactions on Mathematical Software*, vol. 9, no. 3, pp. 302–325, 1983.
- [20] J. W. Liu, "The multifrontal method for sparse matrix solution: theory and practice," *SIAM Review. A Publication of the Society for Industrial and Applied Mathematics*, vol. 34, no. 1, pp. 82–109, 1992.
- [21] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

- [22] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-accelerated sparse LU factorization for circuit simulation with performance modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 786–795, 2015.
- [23] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for SpMV on GPU using probabilistic modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 196–205, 2015.
- [24] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2014.