FSCHOL: An OpenCL-based HPC Framework for Accelerating Sparse Cholesky Factorization on FPGAs

Erfan Bank Tavakoli, Michael Riera, Masudul Hassan Quraishi, Fengbo Ren
Arizona State University
Tempe, AZ, USA
Email: {ebanktav, mriera1, mquraish, renfengbo@asu}.edu

Abstract—The proposed FSCHOL framework consists of an FPGA kernel implementing a throughput-optimized hardware architecture for accelerating the supernodal multifrontal algorithm for sparse Cholesky factorization and a host program implementing a novel scheduling algorithm for finding the optimal execution order of supernodes computations for an elimination tree on the FPGA to eliminate the need for off-chip memory access for storing intermediate results. Moreover, the proposed scheduling algorithm minimizes on-chip memory requirements for buffering intermediate results by resolving the dependency of parent nodes in an elimination tree through temporal parallelism. Experiment results for factorizing a set of sparse matrices in various sizes from SuiteSparse Matrix Collection show that the proposed FSCHOL implemented on an Intel Stratix 10 GX FPGA development board achieves on average 5.5× and 9.7× higher performance and 10.3× and 24.7× lower energy consumption than implementations of CHOLMOD on an Intel Xeon E5-2637 CPU and an NVIDIA V100 GPU, respectively.

Keywords—Cholesky factorization, sparse matrix decomposition, FPGA, OpenCL, high-performance computing, reconfigurable computing.

I. INTRODUCTION

Solving large symmetric sparse linear systems using sparse Cholesky factorization plays a pivotal role in many scientific computing and high-performance computing (HPC) applications [1]–[4]. Nonetheless, the existing computational solutions to sparse Cholesky factorization based on CPUs and GPUs suffer from very limited performance due to two primary reasons. First, sparse Cholesky factorization algorithms involve complex operations (e.g., inverse square root) that are often computed using approximation algorithms (e.g., the Newton-Raphson method [6]) that are also iterative and have strong loop-carried data dependency. Unfortunately, the legacy hardware architectures of CPUs and GPUs, while being able to exploit massive spatial parallelism, lack the capability to exploit the temporal/pipeline parallelism that is critical to resolving such loop-carried data dependency, which results in long loop initiation intervals causing further reduced performance.

In addition to the limited performance issue of CPU- and GPU-based sparse Cholesky factorization solutions for HPC applications, these solutions suffer from very high energy consumption due to high runtime (i.e., low performance) and power consumption of CPUs and GPUs (e.g., 135 W thermal design power for Intel Xeon Processor E5-2637 v3 [7] and 250 W for NVIDIA V100 TENSOR CORE GPU). The high energy consumption of CPUs and GPUs in HPC data centers has received significant attention due to its high economic, environmental, and performance costs [8].

As FPGAs are being deployed as an emerging accelerator in data centers [9], [10], FPGA computing offers an alternative solution to accelerating sparse Cholesky factorization for HPC applications. An FPGA is a farm of configurable hardware resources whose functionality and interconnection can be redefined at run-time by programming its configuration memory. A state-of-the-art FPGA carries an enormous amount of fine- and coarse-grained logic, computation, memory, and I/O resources. Upon the reconfiguration of these resources, an FPGA can implement any custom hardware architecture to accelerate algorithms with both performance and energy efficiency gains [11]–[13]. Specifically, the fine-grained logic resources and the abundant on-chip memory and register resources on FPGA devices can be used to implement the customized buffering scheme tailored to a given sparse Cholesky factorization algorithm to allow efficient storage of intermediate results and data movement among and within processing elements (PEs) with no off-chip memory access required [14], [15]. Furthermore, the hardware flexibility of an FPGA allows its reconfigurable resources to compose not only spatial but also temporal/pipeline parallelism both at a fine granularity and on a massive scale to best resolve the
complex loop-carried data dependency that exists in sparse Cholesky factorization algorithms to minimize loop initiation intervals for improved performance [16]. Finally, FPGAs have lower energy consumption than CPUs and GPUs due to their lower runtime and power consumption.

So far, there has been limited work for accelerating sparse Cholesky factorization on FPGAs [17], [18]. The limitations of the existing work are three-fold. First, the existing work adopts either the left-looking [18] or the multifrontal algorithm [17] in their implementations. These algorithms are less optimized in terms of the memory access and computational complexity than the supernodal multifrontal algorithm for sparse Cholesky factorization [5], [17], [19]. Second, the existing work [18] based on the multifrontal algorithm fails to provide a scheduling algorithm for ordering and assigning the computation of different nodes in an elimination tree. The lack of a scheduling algorithm ignores the dependency among different nodes in an elimination tree, which inevitably demands frequent off-chip memory access and increases the size of on-chip memory required to load and store intermediate results. Third, the FPGA accelerator architecture proposed in [17] does not allow on-chip communication among different PEs, which enforces a large amount of off-chip memory access to occur for transferring intermediate results among PEs.

Tradition FPGA design is at the hardware description language (HDL) level, which is hard to be adopted by software and application developers in the HPC community. In recent years, high-level synthesis (HLS) technology that auto-generates HDL codes from high-level programming languages (e.g., C/C++, OpenCL, MATLAB, etc.) is becoming increasingly mainstream and makes FPGA computing a viable solution for HPC. However, new challenges also arise in the new HLS-based FPGA design flow, such as how to precisely define and optimize hardware architectures in a software-defined fashion and how to construct efficient computation pipelines for data flow processing using a memory-compute-based programming model. FPGA computing is currently an active area of research, and many questions are yet to be answered.

In this paper, we propose FSCHOL, an OpenCL-based HPC framework for accelerating sparse Cholesky factorization on FPGAs. The proposed FSCHOL framework consists of an FPGA kernel implementing an energy-efficient and throughput-optimized hardware architecture and a host program implementing a novel scheduling algorithm. We adopt the supernodal multifrontal algorithm [5], [17], [19] that requires much less memory access and features lower computational complexity than the left-looking and the multifrontal algorithm used in the existing work, which is critical to more efficient hardware mapping and improved performance.

Moreover, we propose a memory-optimized scheduling algorithm for the host program for provisioning the execution of the supernodal multifrontal Cholesky factorization, and potentially all elimination-tree-based multifrontal methods, on an FPGA device. The scheduling algorithm identifies the dependency among computation nodes in an elimination tree and correspondingly arrange their computation order on the FPGA device to avoid off-chip memory access as well as to minimize the on-chip memory requirements for storing intermediate results. This is the key to enabling data locality, thereby improving both computational performance and energy efficiency. Finally, the proposed OpenCL-based FPGA kernel architecture enables pipelined on-chip transfers of intermediate results among PEs by utilizing FIFO channels and eliminates undesired off-chip memory accesses by working in coordination with the scheduling algorithm running on the host side.

The contributions of this paper are summarized as follows.

- We propose an end-to-end OpenCL-based HPC framework for accelerating sparse Cholesky factorization on FPGAs, consisting of both an energy-efficient and deeply pipelined FPGA kernel for accelerating the supernodal multifrontal algorithm and a scheduling algorithm in the host program for eliminating the off-chip memory access and minimizing the on-chip memory requirements of the FPGA kernel for handling intermediate results.
- The proposed FPGA hardware architecture is parameterized and scalable. We propose a method for determining the optimized architectural parameters for maximizing the run-time performance given a suitable FPGA board.
- The proposed scheduling algorithm can be potentially applied to all elimination-tree-based multifrontal algorithms, including both the multifrontal and the supernodal multifrontal methods, for relaxing the memory constraints of an accelerator device (e.g., an FPGA or a GPU) in a host-accelerator computing model.
- We evaluate the performance and energy efficiency of FSCHOL based on an Intel Stratix 10 GX FPGA development board and compare it with CHOLMOD, a state-of-the-art library for sparse Cholesky factorization, running on an Intel Xeon E5-2637 CPU and an NVIDIA V100 TENSOR CORE GPU, respectively, for factorizing the nd3k, Trefethen_20000b, smt, thread, pdb1HYS, and nd24k matrices selected from SuiteSparse Matrix Collection [20]. The experimental results show that the proposed FSCHOL solution has on average 5.5× and 9.7× higher performance and 10.3× and 24.7× lower energy consumption than the CPU and GPU implementation of CHOLMOD, respectively.
- We compare FSCHOL with the state-of-the-art FPGA design [17] for running the same set of benchmarks used in [17], namely nd3k, Trefethen_20000b, and nd24k. The experimental results show that FSCHOL improves the technology-node-normalized performance on average by 11.7× for accelerating sparse Cholesky factorization.
factorization on FPGAs over the reference method by completely eliminating off-chip memory access via software-hardware co-design.

II. BACKGROUND

Cholesky factorization is an efficient method for decomposing a symmetric positive-definite matrix \( A \) into the product of a lower triangular matrix and its transpose \( L \cdot L^T \). In many real-life science and engineering applications, matrix \( A \) is sparse [21]. One of the significant methods for sparse matrix factorization was introducing the multifrontal Cholesky factorization [22].

The multifrontal method reorganizes the overall factorization of a sparse matrix into a sequence of partial factorizations of smaller dense matrices [5]. The main feature of the multifrontal method is that the update contributions from a factor column \( i \) \((L; i)\) to the remaining submatrix are computed and accumulated with contributions from other factor columns before the updates are performed. Therefore, this method reduces the number of memory accesses and operations comparing to left or right-looking algorithms [17].

The main concepts in the multifrontal method are the elimination tree, frontal and update matrices (denoted by \( F \) and \( U \), respectively), and the extended-add operation. The elimination tree of the matrix \( L \) is defined as a tree structure with \( n \) nodes such that node \( P \) is the parent of node \( C \) if and only if the first subdiagonal nonzero element at column \( C \) is located at row \( P \). Figure 1a and 1b shows the nonzero pattern of an example matrix \( L \) and its corresponding elimination tree, respectively. If we define the critical path as the longest path of nodes from the first to the top level of the elimination tree (e.g., colored nodes in Figure 1b), the number of nodes in the critical path determines the maximum amount of dependency among nodes to be resolved.

A practical improvement to the multifrontal method is the use of supernodes [5]. A supernode is a group of columns \( l.e., \) nodes in the elimination tree \( l.\) if they can be treated as one computational unit in the course of sparse Cholesky factorization. If we define the sparsity structure \( l.e., \) nonzero patterns \( l.\) of column \( j \) as \( Struct(L(:,j)) \), the set of contiguous columns \( \{ j, j + 1, \ldots, j + t \} \) constitutes a supernode if \( Struct(L(:,k)) = Struct(L(:,k + 1)) \cup \{ k \} \) [19]. One can refer to [17] and [22] for the detailed comparison on different sparse Cholesky factorization algorithms.

Figure 1c shows the supernodal elimination tree of Figure 1b. Algorithm 1 describes the sparse Cholesky factorization using the supernodal multifrontal method. In Algorithm 1, notion \( nonzeros(V) \) is equivalent to the dense form \( l.e., \) nonzero elements \( l.\) of sparse vector \( V \). Also, notion \( \oplus \) represents the extend-add operation that adds two matrices with different dimensions by extending the smaller matrix with zeros. Comparing the critical path of the two methods, the node dependency is reduced. Moreover, Algorithm 1 introduces more parallelism in each outer loop iteration.

III. RELATED WORK

A. Accelerating Basic Linear Algebra Subprograms (BLAS) on FPGAs

The work in [23] presents parametrized implementations of dot-product and matrix-vector multiplication kernels for FPGAs and compares the performance and energy efficiency of the FPGA kernels with the CPU and GPU implementations. FBLAS [24] is an OpenCL-based, modular implementation of
Algorithm 1: The supernodal multifrontal Cholesky factorization [5]

for each supernode $S$ in increasing order of first column subscript do

1. Let $S = \{j, j+1, \ldots, j+\ell\}$;
2. Let $j + 1, j_1, \ldots, j_p$ be the locations of nonzero elements in $L(:, j + 1)$;
3. $F_S = \begin{bmatrix}
    a_{i,j} & a_{j+1,j} & \cdots & a_{j+\ell,j} \\
    a_{j+1,j} & a_{j+1,j+1} & \cdots & a_{j+1,j+\ell} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{j+\ell,j} & a_{j+\ell,j+1} & \cdots & a_{j+\ell,j+\ell} \\
  \end{bmatrix}

4. if $r \leftarrow \text{nonzeros}(L(:, j + i)) = F_S(:, :) \cap U$; then
5. $F_G = F_S \oplus U$; // update($S, C$)
6. end

for $C$ from 1 to $\text{children}$ do
7. /* start of factorize($S$) */
8. for $i$ from 0 to $\text{children}$ do
9. /* $\text{nonzeros}(L(:, j + i + 1))$ */
10. $L(:, j + i + 1)$
11. end
12. end
13. /* end of factorize($S$) */

$n\text{children} = \text{no. of children of } S \text{ in supernodal elimination tree};$

IV. FRAMEWORK DESIGN

The FSCHOL framework consists of two parts: The FPGA kernel and the host program running on the CPU. The kernel code implemented on the FPGA accelerator performs computationally intensive tasks. On the host side, the OpenCL API supports efficient management and scheduling of tasks running on the FPGA.

A. Hardware Architecture of the FPGA Kernel

1) Architectural Overview: Figure 2 shows a high-level block diagram of FSCHOL’s hardware architecture, including six modules: two processing elements (PEs), two load, and two store modules. The PEs are responsible for computations, while load and store modules read and write input and output data to/from off-chip memory, respectively. All modules process data in a pipelined and vectorized fashion. PEs are connected to load and store modules via FIFO channels. Also, PEs utilize FIFOs to send and receive intermediate results to/from each other. Separating load/store modules from PEs and connecting them using a FIFO helps compensate for the difference between the off-chip memory bandwidth and the data processing throughput.

When the data processing throughput does not match the available off-chip memory bandwidth, and loading and storing primary input and output data happen in the same module that the data are being processed, the load and store operations would be stalled for the computation units. When the depth of the FIFO channels is optimized by the offline compiler, the load and store modules are able to continuously read and write data from/to the off-chip memory and write and read them to/from the channels, respectively. For each supernode, in addition to several consecutive columns of input matrix $A$, depending on the size of the supernode, a PE needs configuration information on how to process the assigned supernode (job). The job information is set by the
Algorithm 1. Bit of Algorithm 1) updates the frontal matrix of supernode to be updated by all of its children according to lines 6-8 of Algorithm 1. Therefore, vector $V_U$ is initialized by input data from $Q_A$. If node $S$ is already updated, depending on the value of $job,F_{rd}$, $V_F$ reads its value from $Q_F$ or $Q_{F,PE}$. According to Table I, if $job,F_{rd}$ is set, it shows that the intermediate values of matrix $F$ as the result of a previous update of supernode $S$ by one of its other children are stored at the inter-module FIFO $Q_{F,PE}$; otherwise they are stored at intra-module FIFO $Q_F$. If node $S$ does not have any child, there is no update matrix to update node $S$. Therefore, $U_{rd}$ determines whether vector $V_U$ should be initialized by zeros (when there is no child) or by the values stored at $Q_U$. After loading vectors $V_U$ and $V_F$, with the corresponding values, they are added in parallel, and the output is stored at $V_F$.

After adding vectors $V_U$ and $V_F$, a PE decides whether to store the results or feed $V_F$ to the pipeline for the $factorize(S)$ operation. As described in Table I, bit $job,last,c$ defines whether node $S$ is ready to be factorized. If it is not set, PE stores vector $V_F$ in FIFO $Q_F$ or $Q_{F,PE}$ depending on the value of $job,F_{wr}$. If it is set, the PE takes the square root of the first $t+1$ diagonal elements of $F_S (f_{1,1}, f_{2,2}, \ldots, f_{t+1,t+1})$ and divides the superdiagonal elements of $F_S$ by the square root value. The PE uses elements $l_{i,j,t+1}, l_{i,j,t+1}, \ldots, l_{i,t+1}$ for the consequent outer product operation. Moreover, update matrix $U_S$ is computed as described at line 10 of Algorithm 1 and stored in $Mem_{U_S}$.

Figure 4 shows an example of the extend operation at line 7 of Algorithm 1 and how update matrix $U$ is extended and stored on $Q_U$. The PE initializes an index counter for scheduling algorithm in the host program. Therefore, channels $Q_A$ and $Q_{job}$ are used to send the elements of matrix $A$ and the job information, respectively. Channel $Q_P$ is used to send the Boolean elements of the pattern matrix to PEs for the extend-add operation which the details are discussed in Section IV-A3. Channel $Q_L$ sends the consecutive columns of factor matrix $L$ of the corresponding processed supernode from a PE to the store module. Channels $Q_F$ transmit the intermediate value of matrix $F$ among PEs.

2) Load/Store Modules: Each load module iterates over the number of jobs that are assigned by the scheduling algorithm. First, the load module sends a data structure containing the configuration bits of the assigned job described in Table I. When a supernode $S$ is assigned to a PE, $S$ needs to be updated by all of its children according to lines 6-8 of Algorithm 1. Bit $job,up$ determines whether $S$ is updated by any of its children. If not, the load module reads input data from off-chip memory and write them to $Q_A$ to be consumed by a PE. Store modules read the output data from PEs and send them to the off-chip memory as soon as a vector of factor matrix $L$ is ready.

3) Processing Element (PE): Based on Algorithm 1, we define two major operations in each outer loop iteration: update, and factorize. Operation $update(S,C)$ (line 7 of Algorithm 1) updates the frontal matrix of supernode $S$ ($F_S$) using the update matrix of its child supernode $C$ ($U_C$). Operation $factorize(S)$ (lines 9-12 of Algorithm 1) produces $t+1$ columns of factor matrix $L$ and the update matrix of supernode $S$ ($U_S$). Figure 3 shows the high-level block diagram of each PE. Modules Vector Addition and Matrix Extension are responsible for operation update and modules Sqrt, Vector Division, Outer Product, and Vector Subtraction perform operation factorize. For each scheduled job, a PE performs one update and one factorize operation (depending on the update status of $F_S$) in a pipelined and vectorized fashion.

When there is no available update matrix to update the frontal matrix of supernode $S$, intermediate (i.e., partially updated) values of $F_S$ are stored to complete the update process (the for loop at lines 9-11 of Algorithm 1) later. Storage units for storing intermediate results inside each PE include RAM $Mem_{U_S}$ for the update matrix, FIFO channel $Q_U$ for the extended matrix update matrix, and FIFO channel $F_{F,PE}$ for the partially updated frontal matrix. Two inter-module FIFO channels are used to send and receive the intermediate matrix $F$ ($Q_F$) among PEs whenever necessary.

For each job, PE starts by reading the configuration bits to control the multiplexers. Vectors $V_F$ and $V_O$ are defined to represent a vector with size $VL$ of matrices $F$ and $U$, respectively. If $job,up$ is not set, node $S$ is not updated by any of its children, and its factor vector $V_F$ was not initialized with the values of matrix $A$ as described in line 4 of Algorithm 1. Therefore, vector $V_F$ is initialized by input data from $Q_A$. If node $S$ is already updated, depending on the value of $job,F_{rd}$, $V_F$ reads its value from $Q_F$ or $Q_{F,PE}$. According to Table I, if $job,F_{rd}$ is set, it shows that the intermediate values of matrix $F$ as the result of a previous update of supernode $S$ by one of its other children are stored at the inter-module FIFO $Q_{F,PE}$; otherwise they are stored at intra-module FIFO $Q_F$. If node $S$ does not have any child, there is no update matrix to update node $S$. Therefore, $U_{rd}$ determines whether vector $V_U$ should be initialized by zeros (when there is no child) or by the values stored at $Q_U$. After loading vectors $V_U$ and $V_F$, with the corresponding values, they are added in parallel, and the output is stored at $V_F$.

Figure 2: The high-level block diagram of the hardware architecture.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.up</td>
<td>supernode is partially updated</td>
<td>Boolean</td>
</tr>
<tr>
<td>job.last_c</td>
<td>supernode is going to be updated by its last child</td>
<td></td>
</tr>
<tr>
<td>job,F_{rd}</td>
<td>intermediate matrix $F$ should be read from inter-module FIFO</td>
<td></td>
</tr>
<tr>
<td>job,F_{wr}</td>
<td>intermediate matrix $F$ should be written to inter-module FIFO</td>
<td></td>
</tr>
<tr>
<td>job,U_{rd}</td>
<td>update matrix $U$ should be read from FIFO</td>
<td></td>
</tr>
</tbody>
</table>
written and read to/from off-chip memory.

and the index counter is increased by 1. Otherwise, the corresponding element of the vector is skipped and kept as zero. Once VL iterations have been passed, the vector is stored on QV. As mentioned before, the datapath has VL single-precision floating-point numbers, and the arithmetic submodules perform VL operations in parallel. Table II shows the size of storage units for each PE. Dimensions of FIFOs and RAM blocks are represented as Depth × Width and Rows × Columns, respectively. Parameter WL is the word length of the data format (e.g., 32 bits for single-precision floating-point).

There are three tunable parameters: VL, N, and M. Parameters M and N are used to scale the design to support arbitrary sizes for frontal and update matrices based on the maximum supernode size (i.e., the maximum number of consecutive columns with the same sparsity pattern) and the maximum number of nonzero elements among all columns of the factor matrix, respectively. The total number of required DSPs for the FPGA kernel with two PEs is equal to 7VL+5. According to Table II, total required size for the storage units is (8N^2 + 6M^2 + 12M × N) × VL^2 × WL bits.

5) Performance-optimized Model for Determining Design Parameters: It is important to provide a guideline to derive design parameters (VL, N, and M) to minimize the runtime subject to the available on-chip resources and the characteristics of the input matrices.

We formulate the runtime (R) as 

$$ R = \frac{N}{F \times P \times U}, $$

where N, F, P, and U are the number of floating-point operations (FLOPs) to factorize a matrix using Algorithm 1, the clock frequency, the computational parallelism, and the spatial-temporal utilization factor, respectively. U is a statistical metric that measures the average occupation (in both space and time) of the available computation resources in a
computing device for performing the FLOPs of a given algorithm. Therefore, $U$ is the average ratio of effective computational parallelism per clock cycle ranging from 0 to 1. $N$ is an algorithmic parameter and depends on dimensions and the sparsity structure of the input matrix. $P$ is determined by the total number of utilized DSP units calculated as $7 \times VL + 5$. Therefore, $R$ can also be expressed as

$$R = \frac{N_{ops}}{F \times (7 \times VL + 5) \times U} = \frac{\alpha}{7 \times VL + 5},$$

(1)

where $\alpha = \frac{N_{ops}}{F \times U}$ is a empirical term that lumps the algorithm- and implementation-specific terms $N_{ops}$, $F$, and $U$. Note that $\alpha$ is introduced mainly to simply the formulation and can be treated as a constant when determining the optimal architectural parameters.

We define the constrained optimization problem as

$$\min R(VL) = \frac{\alpha}{7 \times VL + 5}$$

subject to

$$f_1(VL) \leq C_1, f_2(VL, M, N) \leq C_2,$$

$$M = \left\lceil \frac{C_3}{VL} \right\rceil, N = \left\lceil \frac{C_1}{VL} \right\rceil,$$

(2)

where $f_1(VL) = 7 \times VL + 5$ and $C_1$ are the total number of required and available DSP blocks, respectively, $f_2(VL, M, N) = (8N^2 + 6M^2 + 12M \times N) \times VL^2 \times WL$ and $C_2$ are the total required and available RAM size, respectively, $C_3$ is the maximum supernode size (i.e., maximum number of consecutive columns of the factor matrix with the same sparsity pattern), and $C_4$ is the maximum number of nonzero elements among all columns of the factor matrix. For the total available RAM size ($C_2$) one must consider a margin from the values reported in the FPGA device datasheet to consider for RAM resources used for the glue and control logics.

Based on $R(VL) = \frac{\alpha}{7 \times VL + 5}$, to minimize the runtime, $VL$ should be maximized. The constrained optimization problem defined as Equation 2 has an analytical solution as following.

1) Derive the first constraint for $VL$ from $f_1(VL) \leq C_1$.
2) Derive the second constraint for $VL$ by using $M = \left\lceil \frac{C_3}{VL} \right\rceil$ and $N = \left\lceil \frac{C_1}{VL} \right\rceil$ in $f_2(VL, M, N) \leq C_2$.
3) Now we have two ranges of values for $VL$. The maximum value of $VL$ is determined from the tighter constraint.
4) Using calculated $VL$, the values of $N$ and $M$ are found from $M = \left\lfloor \frac{C_3}{VL} \right\rfloor$ and $N = \left\lfloor \frac{C_1}{VL} \right\rfloor$.

B. Scheduling Algorithm

1) Concepts: The main challenges involved in co-designing the scheduling algorithm with the the proposed hardware architecture for the FPGA kernel stem from two perspectives. On the one hand, as the objective of the scheduling algorithm is to minimize the amount of off-chip memory access on the FPGA device, the impact of workload scheduling on the amount of off-chip memory access must be accurately modeled based on the specific architecture of the FPGA kernel. On the other hand, the scheduling algorithm needs to generic enough to be able to adapt to the FPGA kernel implementations in various computational parallelism on different FPGA devices.

We propose and implement a scheduling algorithm that intelligently assigns the update and factorization operations (update and factorize) to PEs to minimize on-chip memory requirements with no off-chip memory access for the storage of intermediate results. The main goal of the algorithm is to pipeline the dependency among a child supernode $C$ and its parent supernode $S$ to avoid storing the update matrix of supernode $C$. Therefore, the algorithm assigns operation update($S, C$) immediately after operation factorize($C$). In this order, when supernode $C$ is factorized and its update matrix is produced, the update matrix is consumed by supernode $S$ at the same PE to reduce the on-chip storage requirements and reduce communications among different PEs. We provide the pseudo-code only for PE1 as it is the same for PE2. Note that the algorithm works for both supernodal and potentially multifrontal Cholesky factorization.

2) Details: In Algorithm 2, list $nc$ is used to keep track of the number of children that each supernode is already updated by. Line 3-6 works on supernodes with no child. Since they do not have any dependency, it is only needed to initialize the frontal matrix $F$ with elements of the matrix $A$ and immediately factorize the supernode. Once a supernode with no dependency is factorized in a PE, its update matrix is ready to update its parent supernode. Therefore, the supernode ID is pushed to PE’s queue. Then, the supernode ID is popped from the queue, and the corresponding parent supernode is updated. Suppose a supernode is updated by all of its children. In that case, it is ready to be factorized, and the resulting update matrix is used for updating the parent supernode in the consequent job assigned to this PE. Otherwise, the queue would be empty, and the algorithm starts with lines 3-6. During this process, if a parent supernode needs to be updated in a different PE than it was updated, the configuration bits of the job assigned to the other PE is updated to let the PE know that it has to send the intermediate frontal matrix to the inter-module FIFO ($Q_F$) as shown by crossed-out functions in Figure 5.

The scheduling algorithm is implemented as a part of the host code. As an example, we apply the algorithm to the elimination tree illustrated in Figure 1c. The ordered list of supernodes is $T = [0, 4, 1, 2]$. Lists $P$ and $NC$ are $[7, 3, 5, 8, 5, 6, 9, 9, 9, -1]$ and $[0, 0, 0, 0, 1, 0, 2, 1, 1, 3]$, respectively. Value $-1$ in list $P$ shows that the last node is reached. The output of this example is shown in Figure 5.
Algorithm 2: The scheduling algorithm

**Input:** The list of (super)nodes with no child \((T)\), the list of parents for each (super)node \((P)\), and the list of the no. of children of each (super)node \((NC)\).

**Output:** The assignment of update and factorize operations to each PE.

1. initialize list \(nc\) to zeros to keep track of the no. children that each (super)node is updated by;
2. initialize two single-element queues \(Q_1\) and \(Q_2\) as empty to store the ready (super)node to be updated or factorized;
3. while the top (super)node is not factorized do
   4. if \(Q_1\) is empty then
      5. \(S = T[0]\);
      6. assign update\((S,-)\) to \(PE_1\);
      7. remove \(T[0]\) from \(T\);
      8. \(p = P[S]\);
      9. push\((p,Q_1)\);
   else
      10. \(S = pop(Q_1)\);
      11. indicate (super)node \(S\) is being updated;
      12. indicate vector \(V_U\) should be read from \(Q_U\);
      13. if \(nc[S]\) is equal to zero then
         14. if (super)node \(S\) was updated in this PE
             15. indicate vector \(V_F\) should be read from \(Q_F,PE\);
             16. else
                 17. indicate vector \(V_F\) should be read from \(Q_F\);
                 18. update the job bits of the assigned job to other PE by indicating it should write the intermediate vector \(V_F\) to \(Q_F\);
         19. end
      20. end
      21. \(nc[S]++\);
      22. if \(nc[S]\) is equal to \(NC[S]\) then
         23. assign\((p[S])\) to \(PE_1\);
         24. push\((S,Q_1)\);
      25. end
   end
   /* similar approach for \(PE_2\) */
end

V. EVALUATION

A. Setup

We evaluate the performance and energy efficiency of FSCHOL in terms of runtime \((s)\) and energy consumption \((J)\). To evaluate the design, we select a set of matrices from the publicly available SuiteSparse Matrix Collection [20] (formerly known as the University of Florida Sparse Matrix Collection), a set of sparse matrices in real applications. The characteristics of the matrices are summarized in Table III including matrix dimensions (the number of rows and columns are equal), the density percentage calculated from \(\text{No. of Nonzero Elements} \times 100\), and the number of supernodes.

The data format in our design is single-precision floating-point (32-bit). We develop and implement FSCHOL using Intel FPGA SDK for OpenCL with Quartus Prime Pro 20.1. We compare FSCHOL with CPU and GPU versions of CHOLMOD [26].

As mentioned before, the supernodal multifrontal method decomposes the sparse Cholesky factorization into a series of dense factorizations. These dense factorizations rely on dense BLAS and LAPACK libraries. Therefore, to improve the performance of the CHOLMOD library on CPU, we use Intel Math Kernel Library (Intel MKL) [29] instead of single-threaded BLAS and LAPACK routines. Intel MKL is a set of highly optimized, threaded, and vectorized math functions that maximize the performance of Intel’s processors.

We measure the performance of the CPU implementation of CHOLMOD on a dual-socket Intel Xeon E5-2637 v3 CPU [7] with an effective bandwidth of 51 GB/s per socket, and the GPU implementation of CHOLMOD on an NVIDIA Tesla V100 GPU, one of the most powerful data center GPUs for accelerating HPC [30]. Our work is evaluated on an Intel Stratix 10 GX FPGA Development Board [31]. Table IV...
Table IV: Specifications of the CPU system, the GPU device, and the FPGA board used in the evaluation.

<table>
<thead>
<tr>
<th>Hardware Platform</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5-2637 v3 CPU</td>
<td>15M Cache, 3.50 GHz Clock Frequency, 4 Cores, 68 GB/s Memory Bandwidth</td>
</tr>
<tr>
<td>NVIDIA Tesla V100 GPU</td>
<td>16 GB HBM2, 640 Tensor Cores, 3120 CUDA Cores, 1245-1380 MHz Clock Frequency, 900 GB/s Memory Bandwidth</td>
</tr>
<tr>
<td>Intel Stratix 10 GX FPGA Board</td>
<td>3760 DSPs, 225 MB M20K, 15 MB MLAB, 15 GB/s Memory Bandwidth</td>
</tr>
</tbody>
</table>

Table V: Resource utilization on Intel Stratix 10 GX with \( V_L = 128 \), \( N = 4 \), and \( M = 2 \).

<table>
<thead>
<tr>
<th>Resource Utilization</th>
<th>ALUTs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>315,858</td>
<td>634,846</td>
<td>6,844</td>
<td>901</td>
</tr>
<tr>
<td></td>
<td>17%</td>
<td>17%</td>
<td>58%</td>
<td>16%</td>
</tr>
</tbody>
</table>

summarizes the specifications of the CPU system, the GPU device, and the FPGA board used in the evaluation.

B. Experiment Results

1) Performance Comparison with CPU and GPU Implementations: The architectural parameters for implementing the FPGA kernel are \( V_L = 128 \), \( N = 4 \), and \( M = 2 \). The FPGA kernel runs at 236 MHz. Table V shows the resource utilization for the implemented FPGA kernel.

Table VI shows the performance comparison of FSCHOL in terms of runtime in seconds with the GPU version of CHOLMOD and the CPU version of CHOLMOD enhanced with Intel MKL library for implementing the supernodal multifrontal Cholesky factorization algorithm. The lower runtime shows higher performance. According to Table VI, the CPU implementation outperforms the GPU version of CHOLMOD since algorithms and applications with low arithmetic computation and complex memory handling are more efficient to be mapped on CPUs than on GPUs [9]. Also, the GPU runs with the error correction code (ECC) turned on at the base clock speed. One can further boost the GPU performance using a boost clock speed with ECC turned off [32] at the cost of much higher power consumption and random bit errors. We choose the base clock speed with ECC to evaluate the sustainable performance achievable in a scientific computing environment. The GPU can not work at a boost clock speed permanently or for a long time. Moreover, turning off ECC compromise the results probabilistically where accurate results are necessary for scientific computing.

Figure 6 shows the performance improvement of FSCHOL over the CPU and GPU implementations of CHOLMOD. The GPU runs with ECC turned on at the base clock speed.

Figure 6: Runtime speedup of the FPGA implementation of FSCHOL over the CPU and GPU implementations of CHOLMOD. The GPU runs with ECC turned on at the base clock speed.

many supernodes that cause performance degradation, while for matrices with a very structured sparsity pattern and a small number of nonzero supernodes compared to the matrix dimension, FSCHOL significantly improves the performance as the performance scales with the number of supernodes and the matrix size.

2) Performance Comparison with the State-of-the-art FPGA Implementation: The work in [17] implemented the multifrontal Cholesky factorization algorithm on a Xilinx Virtex-7 FPGA VC709 evaluation board with a 28nm FPGA device [33]. Differently, FSCHOL is implemented on an Intel Stratix 10 GX FPGA development board with a 14nm FPGA device [34]. For a fair comparison, we normalize the performance numbers of [17] to the 14nm technology node using the scaling factor of \( \text{delay} = 1/S \), where \( S = 28/14 \text{nm} \), commonly used in the existing literature [35]. In addition, the performance of the FSCHOL solution is evaluated based on the same matrices used in the reference work [17].

Table VII shows the performance comparison of FSCHOL

Table VI: Runtime (second) comparison between the FPGA implementation of FSCHOL and the CPU and GPU implementations of CHOLMOD.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CHOLMOD</th>
<th>FSCHOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>nd3k</td>
<td>0.31</td>
<td>0.09</td>
</tr>
<tr>
<td>Trefethen 20000b</td>
<td>3.56</td>
<td>4.82</td>
</tr>
<tr>
<td>thread</td>
<td>0.26</td>
<td>1.84</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>0.36</td>
<td>2.40</td>
</tr>
<tr>
<td>nd24k</td>
<td>10.49</td>
<td>12.02</td>
</tr>
<tr>
<td>Fault_609</td>
<td>32.00</td>
<td>49.26</td>
</tr>
<tr>
<td>Emilia_20k</td>
<td>55.57</td>
<td>62.90</td>
</tr>
<tr>
<td>bone010</td>
<td>23.28</td>
<td>58.48</td>
</tr>
</tbody>
</table>

\( ^{\#} \) ECC on and base clock speed.
Table VII: Runtime (second) comparison between the FPGA implementation of FSCHOL and the reference work in [17].

<table>
<thead>
<tr>
<th>Matrix</th>
<th>[17]</th>
<th>[17]#</th>
<th>FSCHOL (Speedup)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nd3k</td>
<td>1.96</td>
<td>0.98</td>
<td>0.05 (19.6×)</td>
</tr>
<tr>
<td>Trefethen_20000b</td>
<td>1.94</td>
<td>1.97</td>
<td>1.30 (11.3×)</td>
</tr>
<tr>
<td>nd24k</td>
<td>10.12</td>
<td>3.06</td>
<td>0.36 (14.1×)</td>
</tr>
</tbody>
</table>

# Technology scaling to 3nm: delay = V/S where S = L/14nm.

in terms of normalized runtime in seconds with the work in [17] for implementing the multifrontal Cholesky factorization algorithm. The original performance results of [17] are also shown in Table VII. Similarly, the lower runtime shows the higher performance, and the numbers in parentheses show the performance improvement of FSCHOL with respect to the corresponding matrix. The experimental results show that the proposed FSCHOL solution outperforms the reference design in terms of performance on average by 11.7× across the three different benchmarks.

The performance improvement of FSCHOL over the work in [17] is primarily due to the elimination of off-chip memory access for buffering intermediate results as a result of our software-hardware co-design methodology. Specifically, the scheduling algorithm implemented in the host program is tailored to the proposed hardware architecture of the FPGA kernel to offload the computational workloads of different supernodes in an optimized order that maximizes the data reuse of intermediate results, thus avoids unnecessary off-chip memory access.

3) Energy Efficiency Comparison with CPU and GPU Implementations: We measure the average power consumption of the Intel Stratix 10 GX development board using the Power Monitor tool in the Board Test System (BTS) application provided by Intel [36] during FPGA kernel execution. BTS measures the supply voltage and the drawn current of the entire FPGA board by reads values from on-board sensors.

For the power measurement of the CPU, we utilize likwid-powermeter tool from Likwid [37] to access the Running Average Power Limit (RAPL) counters on the Intel CPU. The RAPL interface is controlled through MSR registers [38]. For the power measurement of the GPU, we utilize the POWER query option [39] of NVIDIA System Management Interface (nvidia-smi) [40] tool.

Table VIII summarizes the energy consumption (J) of different implementations calculated from multiplying the runtime (s) and the power consumption (W).

<table>
<thead>
<tr>
<th>Matrix</th>
<th>CHOLMOD</th>
<th>FSCHOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>GPU</td>
<td></td>
</tr>
<tr>
<td>nd3k</td>
<td>13</td>
<td>39</td>
</tr>
<tr>
<td>smt</td>
<td>11</td>
<td>72</td>
</tr>
<tr>
<td>thread</td>
<td>19</td>
<td>80</td>
</tr>
<tr>
<td>pdb1HYS</td>
<td>15</td>
<td>95</td>
</tr>
<tr>
<td>nd24k</td>
<td>430</td>
<td>723</td>
</tr>
<tr>
<td>Fault_639</td>
<td>1353</td>
<td>3523</td>
</tr>
<tr>
<td>Emilia_923</td>
<td>2279</td>
<td>8665</td>
</tr>
<tr>
<td>bone010</td>
<td>954</td>
<td>5108</td>
</tr>
</tbody>
</table>

# ECC on and base clock speed.

we can not compare FSCHOL in terms of energy consumption with [17].

VI. CONCLUSION

In this paper, we present FSCHOL, an OpenCL-based HPC framework for FPGA acceleration of sparse Cholesky factorization. FSCHOL includes a deeply pipelined and scalable FPGA kernel that accelerates supernodal multifrontal Cholesky factorization algorithm and a scheduling algorithm for efficient assignment of computational nodes for potentially all elimination-tree-based multifrontal methods.

We propose a performance-optimized model to derive architectural parameters for the FPGA kernel subject to the available on-chip resources (DSPs and RAMs) and input matrix characteristics (the maximum supernode size and the maximum number of nonzero elements among all columns of
the factor matrix) to map the design into suitable data-center grade FPGAs.

The experimental results based on the Intel Stratix 10 GX FPGA development board for accelerating the Cholesky factorization of a set of sparse matrices from SuiteSparse Matrix Collection show on average one order of magnitude higher performance and lower energy consumption compared to the state-of-the-art implementations of sparse Cholesky factorization on CPU, GPU, and the other FPGA work [17].

ACKNOWLEDGMENT

This work is supported by an unrestricted gift (CG#1490376) from Cisco Research Center. The FPGA development kits used for this research is donated by Intel to the state-of-the-art implementations of sparse Cholesky factorization on CPU, GPU, and the other FPGA work [17].

REFERENCES


