2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 978-1-6654-4301-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/SBAC-PAD53543.2021.00032

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 (*e.g.*, the multifrontal algorithms [5]) are recursive and have complex data dependencies for sequentially updating nodes in an elimination tree based on the intermediate results from previous iterations. For the sake of data locality and computational performance, an algorithm-tailored buffering scheme for efficiently storing the intermediate results must be employed for computing a sparse Cholesky factorization. Unfortunately, the deep memory hierarchy and fixed hardware architecture of CPUs and GPUs can hardly be adapted to efficiently implement such an algorithm-tailored buffering scheme. Consequently, CPU- and GPU-based solutions suffer from poor cache locality and often require frequent off-chip memory access for computing sparse Cholesky factorization, greatly limiting their performance. Second, 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$\times$ and 9.7$\times$ higher performance and 10.3$\times$ and 24.7$\times$ 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$\times$ for accelerating sparse Cholesky

factorization on FPGAs over the reference method by completely eliminating off-chip memory access via software-hardware co-design.

II. Background

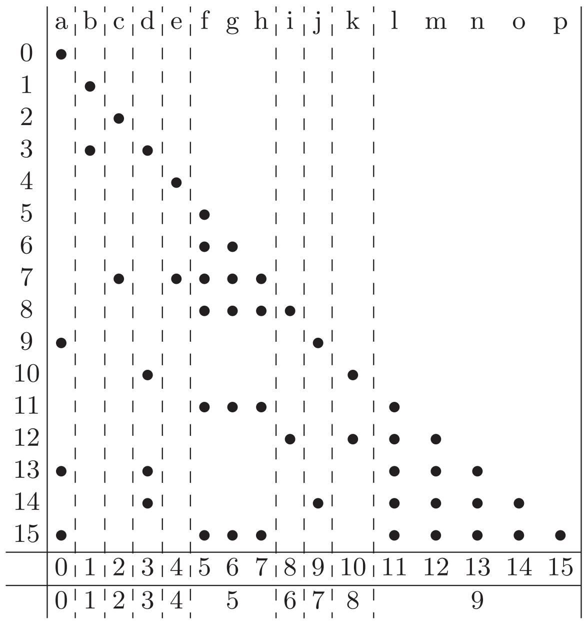
Cholesky factorization is an efficient method for decomposing a symmetric positive-definite matrix (A) into the product of a lower triangular matrix and its transpose ($L \times L^T$). In many real-life science and engineering applications, matrix A is sparse [21]. One of the significant methods for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

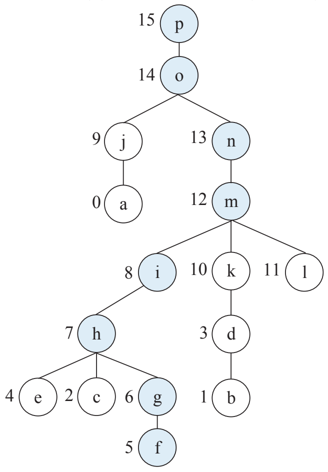
The main concepts in the multifrontal method are the elimination tree, frontal and update matrices (denoted by F and U, respectively), and the extended-add operation. The elimination tree of the matrix L is defined as a tree structure with n nodes such that node P is the parent of node C if and only 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 supenodes [5]. A supernode is a group of columns (*i.e.*, nodes in the elimination tree) if they can be treated as one computational unit in the course of sparse Cholesky factorization. If we define the sparsity structure (*i.e.*, nonzero patterns) of column j as $Struct(L(:,j))$, the set of contiguous columns $\{j, j+1, \cdots, j+t\}$ 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.

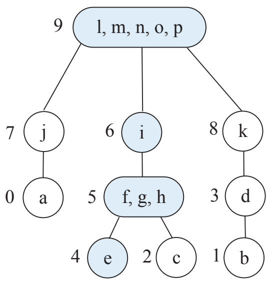
Figure 1c shows the supernodal elimination tree of Figure 1b. Algorithm 1 describes the sparse Cholesky factorization using the supernodal multifrontal method. In Algorithm 1, notion $nonzeros(V)$ is equivalent to the dense form (*i.e.*, nonzero elements) of sparse vector V. Also, notion $\oplus$ represents 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.



(a) The nonzero (circles) pattern of matrix L.



(b) The elimination tree.



(c) The supernodal elimination tree.

Figure 1: The factor matrix sparsity pattern and its corresponding elimination trees.

Additionally, since the update matrix is generated per supernode rather than a node, the number of operations and memory accesses is reduced.

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
    Let $S = \{j, j+1, \cdots, j+t\}$;
    Let $j+t, i_1, \cdots, i_r$ be the locations of nonzero elements in
      $L(:, j+t)$;
    $F_S =$
    $\begin{bmatrix} a_{j,j} & a_{j,j+1} & \cdots & a_{j,j+t} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \cdots & a_{j+1,j+t} & a_{j+1,i_1} & \cdots & a_{j+1,i_r} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{j+t,j} & a_{j+t,j+1} & \cdots & a_{j+t,j+t} & a_{j+t,i_1} & \cdots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \cdots & a_{i_1,j+t} & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{i_r,j} & a_{i_r,j+1} & \cdots & a_{i_r,j+t} & 0 & \cdots & 0 \end{bmatrix}$;
    $nchildren$ := no. of children of $S$ in supernodal elimination tree;
    for C from 1 to nchildren do
        $F_S = F_S \oplus U$;                          // update(S,C)
    end
    /* start of factorize(S)                          */
    for i from 0 to t do
        $nonzeros(L(:, j+i)) = F_S(i:, i)/\sqrt{F_S(i,i)}$;
    end
    $U_S = F_S(t+2:, t+2:$
      $) - \begin{bmatrix} l_{i_1,j+t} \\ \vdots \\ l_{i_r,j+t} \end{bmatrix} \begin{bmatrix} l_{i_1,j+t} & \cdots & l_{i_r,j+c} \end{bmatrix}$;
    /* end of factorize(S)                            */
end
```

the BLAS library for FPGAs with scalability and reusability benefits. While the existing work in [23] and [24] primarily focus on accelerating BLAS, they do not target accelerating sparse Cholesky factorization on FPGAs in the context of HPC.

B. Sparse Cholesky Factorization on CPUs and GPUs

cuSPARSE [25] is a popular CUDA sparse matrix library that can be used to approximate the sparse Cholesky factorization on Nvida GPUs. Since such an approximation algorithm is intrinsically different from the Supernodal Multifrontal algorithm adopted in this work that exactly computes the sparse Cholesky factorization, cuSPARSE is not considered as a reference method for comparison in this paper.

CHOLMOD [26] is a set of routines for factorizing sparse symmetric positive definite matrices for CPUs and GPUs [27] using multifrontal and supernodal multifrontal sparse Cholesky factorization methods. Its supernodal Cholesky factorization provides highly optimized implementations relying on LAPACK and the Level-3 BLAS.

Recursive behavior of sparse Cholesky factorization algorithms and complex data dependencies among nodes in an elimination tree result in frequent off-chip memory access and poor cache locality. Moreover, complex iterative operations (*e.g.*, inverse square root) with strong loop-carried data dependency in sparse Cholesky factorization algorithms lead to lows temporal/pipeline parallelism. Therefore, running these algorithms on CPUs and GPUs suffers low performance.

C. Accelerating Sparse Cholesky Factorization on FPGAs

The work in [18] and [17] are based on the left-looking and multifrontal Cholesky factorization method. There is a major drawback in both of these works. The left-looking and multifrontal algorithms need more memory access and have larger computational complexity than the supernodal multifrontal algorithm [5]. Additionally, the work in [17] requires a scheduling algorithm for assigning the computation of nodes in an elimination tree to the FPGA accelerator. However, [17] did not provide any scheduling algorithm, resulting in suboptimal ordering of different nodes and, consequently frequent off-chip memory access. Moreover, their proposed hardware architecture introduces a long access latency and high overhead to store and read intermediate results to/from the off-chip memory. Since the work in [18] did not provide the runtime of their design, we compare the performance of FSCHOL with [17]. Neither of these two works provided power or energy consumption.

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

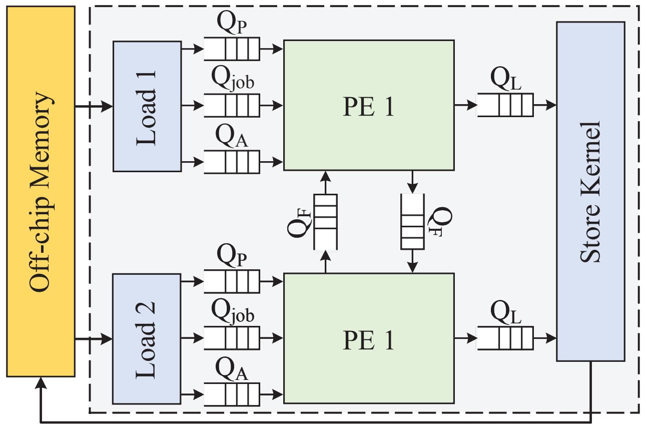


Figure 2: The high-level block diagram of the hardware architecture.

Table I: Configuration bits of a job.

Attribute	Description	Type
$job.up$	supernode is partially updated	Boolean
$job.last\_c$	supernode is going to be updated by its last child	
$job.F\_rd$	intermediate matrix F should be read from inter-module FIFO	
$job.F\_wr$	intermediate matrix F should be written to inter-module FIFO	
$job.U\_rd$	update matrix U should be read from FIFO	

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 node 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 for the update matrix, FIFO channel Q_U for the extended matrix update matrix, and FIFO channel $Q_{F,PE}$ 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_U and V_F 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.

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 ($f_{1,1}, f_{2,2}, \cdots, f_{t+1,t+1}$) and divides the superdiagonal elements of F_S by the square root value. The PE uses elements $l_{i_1,j+t}, l_{i_1,j+t}, \cdots, l_{i_r,j+c}$ for the consequent outer product operation. Moreover, update matrix U_S is computed as describe at line 10 of Algorithm 1 and stored in Mem_U.

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

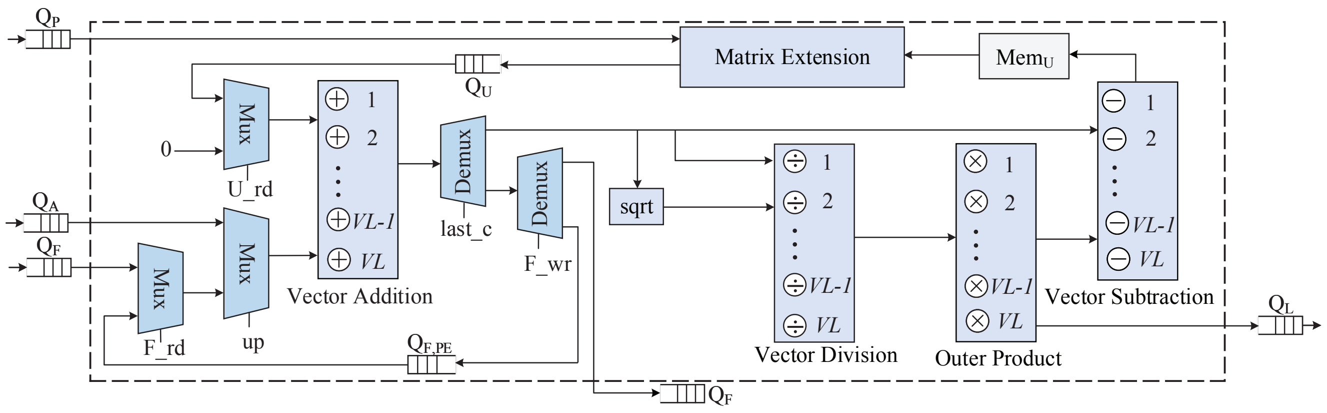


Figure 3: The high-level block diagram of a PE.

$$F = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}, U = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{bmatrix}, P = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\Rightarrow F \oplus U = \begin{bmatrix} f_{1,1}+u_{1,1} & f_{1,2} & f_{1,3}+u_{1,2} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,1}+u_{2,1} & f_{3,2} & f_{3,3}+u_{2,2} \end{bmatrix}$$

Figure 4: An example of extend-add operation.

Mem_U and a vector of elements to zeros, and loops over all elements of a Boolean pattern matrix (P) received from FIFO channel Q_P. If an element of matrix P is true, an element of the update matrix in Mem_U is stored to the vector location indexed by the counter, 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 Q_U.

4) OpenCL Implementation: FSCHOL takes advantage of the loop unrolling technique provided by OpenCL for parallel implementation of vectorized operations. The PEs are implemented as *autorun* modules. An *autorun* module starts executing automatically and does not need to be launched by a host. Therefore, the Intel OpenCL offline compiler for FPGA does not need to generate the communication logic between the host and PEs, which reduces logic utilization and enables additional performance optimizations [28]. However, load and store modules are launched explicitly by the host since they need to access off-chip memory.

Most storage units are implemented as FIFO channels rather than RAMs, where random access is not required. However, for Mem_U there is no choice other than using RAMs. Although using shift registers would be more efficient in terms of hardware complexity, the number of elements to be stored is unknown at the compile time and depends on dimensions of F_S. Additionally, no intermediate data is written and read to/from off-chip memory.

Table II: The required size (bits) for each storage unit.

Unit	Size
$Q_U(2\times)$	$[(N+M)\times(N+M)\times VL]\times VL\times WL$
$Q_F(2\times)$	$[(N+M)\times(N+M)\times VL]\times VL\times WL$
$Q_{F,PE}(2\times)$	$[(N+M)\times(N+M)\times VL]\times VL\times WL$
$Mem_U(2\times)$	$[N\times VL]\times[N\times VL]\times WL$

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 represtend as $Depth \times Width$ and $Rows \times Columns$, respetively. 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. Parameter 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 $7\times VL+5$. According to Table II, total required size for the storage units is $(8N^2+6M^2+12M\times N)\times VL^2\times WL$ 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

$$R = \frac{N_{Ops}}{F \times (7 \times VL + 5) \times U} = \frac{\alpha}{7 \times VL + 5}, \qquad (1)$$

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 α 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

$$\begin{aligned} \text{minimize} \quad & R(VL) = \frac{\alpha}{7 \times VL + 5} \\ \text{subject to} \quad & f_1(VL) \leq C_1, f_2(VL, M, N) \leq C_2, \\ & M = \lceil \frac{C_3}{VL} \rceil, N = \lceil \frac{C_4}{VL} \rceil, \end{aligned} \qquad (2)$$

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 $M = \lceil \frac{C_3}{VL} \rceil$ and $N = \lceil \frac{C_4}{VL} \rceil$ 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 $M = \lceil \frac{C_3}{VL} \rceil$ and $N = \lceil \frac{C_4}{VL} \rceil$.

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 $[0, 0, 0, 1, 0, 2, 1, 1, 1, 3]$, 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          $push(p, Q_1)$;
10     else
11         $S = pop(Q_1)$;
12         indicate (super)node $S$ is being updated;
13         indicate vector $V_U$ should be read from $Q_U$;
14         if $nc[S]$ is equal to zero then
15             if (super)node $S$ was updated in this PE
                 then
16                 indicate vector $V_F$ should be read
                     from $Q_{F,PE}$;
17             else
18                 indicate vector $V_F$ should be read
                     from $Q_F$;
19                 update the job bits of the assigned job
                     to other PE by indicating it should
                     write the intermediate vector $V_F$ to
                     $Q_F$;
20             end
21         end
22         $nc[S] + +$;
23         if $nc[S]$ is equal to $NC[S]$ then
24             assign(p[S]) to $PE_1$;
25             $push(S, Q_1)$;
26         end
27     end
       /* similar approach for $PE_2$ */
28 end
```

update(9,8)	factorize(9)		
update(8,3)	factorize(8)	update(9,6)	send(F_9)
update(3,1)	factorize(3)	update(6,5)	factorize(6)
update(1,-)	factorize(1)	update(5,2)	factorize(5)
update(9,7)	store(F_9)	update(2,-)	factorize(2)
update(7,0)	factorize(7)	update(5,4)	store(F_5)
update(0,-)	factorize(0)	update(4,-)	factorize(4)
PE1		**PE2**	

Figure 5: The output of scheduling algorithm for the supernodal elimination tree in Figure 1c. The time increases from bottom to top.

Table III: The specification of matrices chosen from the SuiteSparse Matrix Collection.

Matrix	#Supernodes	#Rows	Density (%)
nd3k	87	9,000	4.049
Trefethen_20000b	3,678	19,999	0.139
smt	856	25,710	0.567
thread	923	29,736	0.503
pdb1HYS	1,149	36,417	0.328
nd24k	625	72,000	0.554
Fault_639	30,305	638,802	0.007
Emilia_923	43,270	923,136	0.005
bone010	44,319	986,703	0.005

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 $\frac{No.\ of\ Nonzero\ Elements}{Matrix\ Size} \times 100$, 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.

Hardware Platform	Specification
Intel Xeon E5-2637 v3 CPU	15M Cache, 3.50 GHz Clock Frequency, 4 Cores, 68 GB/s Memory Bandwidth
NVIDIA Tesla V100 GPU	16 GB HBM2, 640 Tensor Cores, 5120 CUDA Cores, 1245-1380 MHz Clock Frequency, 900 GB/s Memory Bandwidth
Intel Stratix 10 GX FPGA Board	5760 DSPs, 229 Mb M20K, 15 Mb MLAB, 15 GB/s Memory Bandwidth

Table V: Resource utilization on Intel Stratix 10 GX with $VL = 128$, $N = 4$, and $M = 2$.

Resource	ALUTs	FFs	RAMs	DSPs
Utilization	315,858	634,846	6,844	901
	17%	17%	58%	16 %

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 $VL = 128$, $N = 4$, 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 for corresponding the matrix. FSCHOL improves the performance of CPU and GPU versions of CHOLMOD by 0.8×-29.4× and 3.7×-33.7×, respectively. FSCHOL improves the performance on average by 5.5× and 9.7× over CPU and GPU implementations, respectively. For matrices where FSCHOL is less performant than the CPU implementation of CHOLMOD, the sparsity pattern is less structured and there are too

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

Matrix	CHOLMOD		FSCHOL
	CPU	GPU#	
nd3k	0.31	0.99	0.05
Trefethen_20000b	3.56	4.82	1.30
smt	0.26	1.84	0.31
thread	0.47	2.04	0.34
pdb1HYS	0.36	2.40	0.42
nd24k	10.49	12.02	0.36
Fault_639	33.00	49.26	10.89
Emilia_923	55.57	62.90	15.57
bone010	23.28	58.48	15.88

# ECC on and base clock speed.

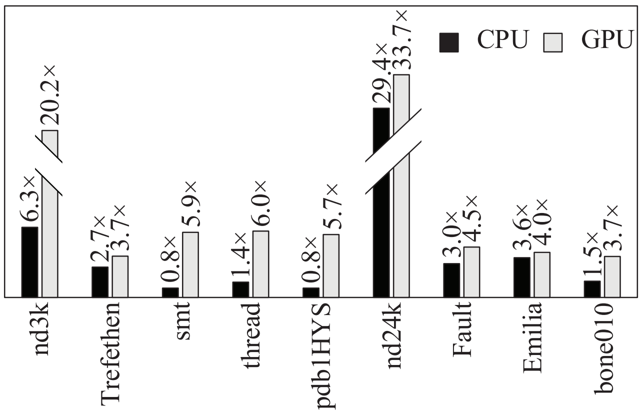


Figure 6: Runtime speedup of the FPGA implementation of FSCHOL over the CPU and GPU implementations of CHOLMOD. The GPU runs with ECC turned on at the base clock speed.

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-7FPGA 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 $delay = 1/S$, where $S = 28/14nm$, 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 VII: Runtime (*second*) comparison between the FPGA implementation of FSCHOL and the reference work in [17].

Matrix	[17]	[17]#	FSCHOL (Speedup)
nd3k	1.96	0.98	0.05 (19.6×)
Trefethen_20000b	3.94	1.97	1.30 (1.5×)
nd24k	10.12	5.06	0.36 (14.1×)

# Technology scaling to 14nm: $delay = 1/S$ 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). Figure 7 show the energy consumption reduction factor of FSCHOL over the CPU and GPU implementations for corresponding the matrix. FSCHOL reduces the energy consumption of the CPU and GPU implementations on by 1.6×-54.7× and 8.5×-92.1×, respectively. FSCHOL reduces the energy consumption on average by 10.3× and 24.7× over CPU and GPU implementations, respectively. Since the work in [17] did not provide any result on power or energy consumption, we can not compare FSCHOL in terms of energy consumption with [17].

Table VIII: Energy consumption (J) comparison between the FPGA implementation of FSCHOL and the CPU and GPU implementations of CHOLMOD.

Matrix	CHOLMOD		FSCHOL
	CPU	GPU#	
nd3k	13	39	1
Trefethen_20000b	146	244	29
smt	11	72	7
thread	19	80	8
pdb1HYS	15	95	9
nd24k	430	723	8
Fault_639	1353	3523	240
Emilia_923	2279	8665	342
bone010	954	5108	349

# ECC on and base clock speed.

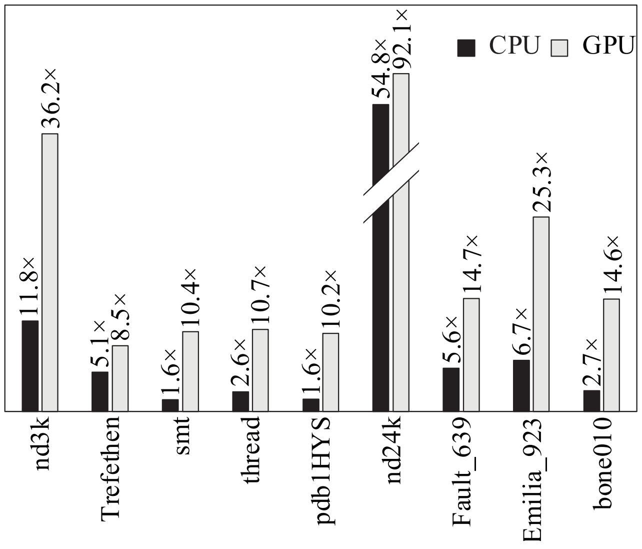


Figure 7: Energy consumption reduction of the FPGA implementation of FSCHOL compared to the CPU and GPU implementations of CHOLMOD. The GPU runs with ECC turned on at the base clock speed.

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 FPGA University Program.

REFERENCES

[1] B. A. Hussein and A. A. Shabana, "Sparse matrix implicit numerical integration of the stiff differential/algebraic equations: Implementation," *Nonlinear Dynamics*, vol. 65, no. 4, pp. 369–382, 2011.

[2] J. Georgii and R. Westermann, "A streaming approach for sparse matrix products and its application in galerkin multigrid methods," *Electronic Transactions on Numerical Analysis*, vol. 37, no. 263-275, pp. 3–5, 2010.

[3] U. Zwick, "All pairs shortest paths using bridging sets and rectangular matrix multiplication," *Journal of the ACM (JACM)*, vol. 49, no. 3, pp. 289–317, 2002.

[4] C. Bouvier, "The filtering step of discrete logarithm and integer factorization algorithms," 2013.

[5] J. W. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM review*, vol. 34, no. 1, pp. 82–109, 1992.

[6] J.-A. Pineiro and J. D. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1377–1388, 2002.

[7] "Intel xeon processor e5-2637 v3 83358." [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/xeon/e5-processors/e5-2637-v3.html

[8] Z. Zhou, J. H. Abawajy, F. Li, Z. Hu, M. U. Chowdhury, A. Alelaiwi, and K. Li, "Fine-grained energy consumption model of servers based on task characteristics in cloud data center," *IEEE access*, vol. 6, pp. 27 080–27 090, 2017.

[9] F. A. Escobar, X. Chang, and C. Valderrama, "Suitability analysis of fpgas for heterogeneous platforms in hpc," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, 2015.

[10] M. H. Quraishi, E. B. Tavakoli, and F. Ren, "A survey of system architectures and techniques for fpga virtualization," *arXiv preprint arXiv:2011.09073*, 2020.

[11] A. Rafique, G. A. Constantinides, and N. Kapre, "Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 24–34, 2014.

[12] M. Farhadi, M. Ghasemi, and Y. Yang, "A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on fpga," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[13] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Polar: A pipelined/overlapped fpga-based lstm accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 838–842, 2019.

[14] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 28, no. 1-2, pp. 7–27, 2001.

[15] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based implementation of signal processing systems*. Wiley Online Library, 2017.

[16] D. Marković and R. W. Brodersen, *DSP architecture design essentials*. Springer Science & Business Media, 2012.

[17] Y. Sun, H. Liu, and T. Zhou, "Sparse cholesky factorization on fpga using parameterized model," *Mathematical Problems in Engineering*, vol. 2017, 2017.

[18] M. Soltaniyeh, R. P. Martin, and S. Nagarakatte, "Synergistic cpu-fpga acceleration of sparse linear algebra," *arXiv preprint arXiv:2004.13907*, 2020.

[19] E. G. Ng and B. W. Peyton, "Block sparse cholesky algorithms on advanced uniprocessor computers," *SIAM Journal on Scientific Computing*, vol. 14, no. 5, pp. 1034–1056, 1993.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[21] G. Lindfield and J. Penny, "Chapter 2 - linear equations and eigensystems," in *Numerical Methods (Fourth Edition)*, fourth edition ed., G. Lindfield and J. Penny, Eds. Academic Press, 2019, pp. 73 – 156. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128122563000117

[22] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear," *ACM Transactions on Mathematical Software (TOMS)*, vol. 9, no. 3, pp. 302–325, 1983.

[23] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI*. IEEE, 2010, pp. 288–293.

[24] T. De Matteis, J. de Fine Licht, and T. Hoefler, "Fblas: Streaming linear algebra kernels on fpga," in *presented at the Int. Conf. HPC Netw. Stor. Anal.*, 2019, pp. 17–22.

[25] "cusparse :: Cuda toolkit documentation." [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[26] 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 (TOMS)*, vol. 35, no. 3, pp. 1–14, 2008.

[27] S. Rennich, T. Davis, and P. Vandermersch, "Gpu acceleration of sparse matrix factorization in cholmod," in *GPU Technology Conference*, 2014.

[28] "Intel fpga sdk for opencl pro edition: Programming guide." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[29] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[30] "Nvidia v100 tensor core gpu." [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[31] "Intel stratix 10 gx/sx product table." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf

[32] "Cholmod." [Online]. Available: https://developer.nvidia.com/cholmod/

[33] "Virtex-7 fpga family." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html

[34] "Intel stratix 10 fpga features." [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10/features.html

[35] F. Ren and D. Marković, "18.5 a configurable 12-to-237ks/s 12.8 mw sparse-approximation engine for mobile exg data aggregation," in *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*. IEEE, 2015, pp. 1–3.

[36] "Intel stratix 10 gx fpga development kit user guide." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-s10-fpga-devl-kit.pdf

[37] T. Gruber, J. Eitzinger, G. Hager, and G. Wellein, "Likwid 5: Lightweight performance tools," 2019.

[38] "Intel 64 and ia-32 architectures software developer's manual: Volume 3." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[39] "nvidia-smi documentation." [Online]. Available: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf

[40] "Nvidia system management interface." [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface