# Spatula: A Hardware Accelerator for Sparse Matrix Factorization

Axel Feldmann
MIT CSAIL
Cambridge, MA, USA
axelf@csail.mit.edu

Daniel Sanchez
MIT CSAIL
Cambridge, MA, USA
sanchez@csail.mit.edu

## ABSTRACT

Solving sparse systems of linear equations is a crucial component in many science and engineering problems, like simulating physical systems. Sparse matrix factorization dominates a large class of these solvers. Efficient factorization algorithms have two key properties that make them challenging for existing architectures: they consist of small tasks that are structured and compute-intensive, and sparsity induces long chains of data dependences among these tasks. Data dependences make GPUs struggle, while CPUs and prior sparse linear algebra accelerators also suffer from low compute throughput.

We present Spatula, an architecture for accelerating sparse matrix factorization algorithms. Spatula hardware combines systolic processing elements that execute structured tasks at high throughput with a flexible scheduler that handles challenging data dependences. Spatula enables a novel scheduling algorithm that avoids stalls and load imbalance while reducing data movement, achieving high compute utilization. As a result, Spatula outperforms a GPU running the state-of-the-art sparse Cholesky and LU factorization implementations by gmean 47× across a wide range of matrices, and by up to thousands of times on some challenging matrices.

## CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**.

## KEYWORDS

Hardware accelerators, sparse linear algebra, matrix factorization, Cholesky, LU.

**ACM Reference Format:**
Axel Feldmann and Daniel Sanchez. 2023. Spatula: A Hardware Accelerator for Sparse Matrix Factorization. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3623783

## 1 INTRODUCTION

Solving sparse systems of linear equations, i.e., finding $x$ such that $Ax = b$ when $A$ is a sparse matrix and $b$ is a vector, is a fundamental problem in scientific computing [8, 22, 35, 36]. Solving sparse linear equations is the dominant computation across many applications, including simulating circuits [7, 18] and physical systems [32], computational fluid dynamics [59], and optimization [5, 31]. As a result, supercomputers spend a substantial fraction of time on solvers [4, 42].

*Matrix factorization* (Section 2) is the dominant component of direct solvers. Matrix factorization decomposes $A$ into two matrices with a particular structure that makes solving $Ax = b$ easy (e.g., $A = LU$, where $L$ is lower-triangular and $U$ is upper-triangular). When $A$ is dense, matrix factorization algorithms are regular and compute-intensive, and existing accelerators like GPUs achieve high efficiency [27]. But $A$ is often highly sparse, which causes poor performance on GPUs and CPUs. For example, on a large circuit matrix (`FullChip`) [16] that has 0.0003% nonzeros, the state-of-the-art GPU factorization algorithm STRUMPACK [22] achieves only 0.3 GFLOP/s on an NVIDIA V100 GPU—just 0.004% of its peak floating-point throughput. STRUMPACK suffers this dismal utilization *despite* using sophisticated algorithms and schedules that seek to use GPUs as best as possible [1].

Sparsity in linear solvers is unavoidable, because it arises from problem structure. For example, consider circuit simulation: a circuit may have millions of nodes, but each node is connected to only a handful of other nodes. Therefore, many problem domains will *always* yield highly sparse matrices. This is different from applications like deep learning, where sparsity is induced as an optimization (e.g., by pruning) and can be shaped or controlled [64].

To tackle this challenge, we present a hardware accelerator for sparse matrix factorization algorithms, including Cholesky and LU. These algorithms have two key properties that thwart GPUs, CPUs, and more specialized sparse accelerators [29, 45, 57, 68]:

First, sparse matrix factorization algorithms contain *long chains of dependences* among tasks, which are hard to schedule. Since GPUs lack fine-grained control over scheduling, they suffer from inefficient schedules that cause load imbalance and force excessive data movement. Prior sparse linear algebra accelerators are also insufficient: most focus on specific kernels, such as sparse matrix-sparse matrix multiplication (SpMSpM) [50, 68, 69], and even more flexible ones like ExTensor [29] and Tensaurus [57] handle loop nests that do not support these complex data dependences.

Second, sparse factorization is nonetheless dominated by *structured compute operations* on smaller matrices that can be effectively accelerated at very high throughput using systolic arrays. No prior architecture can handle this combination of sparse and structured features: the vector datapaths and tensor cores of GPUs are a good match for structured compute, but dependences cause terrible utilization. By contrast, prior sparse linear algebra accelerators focus on unstructured, memory-intensive problems [45, 67, 68] and lack the compute throughput or design to handle structured operations.

To make these challenges concrete, Section 2 presents the necessary background on sparse matrix factorization algorithms (Cholesky


*MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada*

ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3623783

and LU), and Section 3 details why current architectures handle them poorly, including a performance characterization of these algorithms on CPUs and GPUs.

Based on these insights, we present *Spatula*, a novel architecture designed to accelerate sparse matrix factorization. Spatula hardware (Section 4) combines the features needed by these algorithms: its processing elements (PEs) are systolic arrays that implement the primitive tasks in factorization workloads efficiently and at high throughput; and a programmable scheduler orchestrates execution, dispatching tasks to PEs and cheaply tracking frequent data dependences among tasks. Spatula hardware features a cache-based memory hierarchy that captures the irregular reuse in this workload and decoupled-execution mechanisms to fetch data ahead of its use and avoid memory-induced stalls.

To leverage Spatula hardware, we present a novel scheduling algorithm (Section 5) that efficiently schedules sparse matrix factorizations with a wide variety of sparsity patterns. Different sparsity patterns result in fundamentally different computation graphs, presenting different tradeoffs between parallelism and memory footprint. Our approach leverages fine-grained task scheduling, multi-level tiling, and memory-aware scheduling to achieve high utilization while minimizing data movement.

We evaluate Spatula on both sparse Cholesky factorization and sparse LU factorization using a combination of simulation and RTL synthesis (Section 6, Section 7). Spatula outperforms state-of-the-art matrix factorization algorithms on a V100 GPU by gmean 61× on Cholesky and 36× on LU, across a diverse set of matrices from many application domains including circuit simulation, structural analysis, fluid dynamics, and convex optimization. Speedups over a 32-core server CPU are gmean 213× on Cholesky 33× on LU. The evaluated configuration of Spatula has an area of 108 $mm^2$ and consumes 146 W on average when synthesized in 12-14 nm technology, significantly less than the GPU and CPU baselines.

In summary, we make the following contributions:

(1) We identify the key features of sparse matrix factorization and its inefficiencies on current architectures (Section 3).
(2) We propose the first hardware architecture that achieves high performance on sparse matrix factorization (Section 4).
(3) We design a novel scheduling algorithm enabled by this hardware that achieves high utilization (Section 5).
(4) We perform a detailed evaluation of our proposed architecture, showing order-of-magnitude improvements in performance and energy efficiency (Section 6, Section 7).

## 2 BACKGROUND

Solving systems of linear equations $Ax = b$ is a key primitive in many scientific computing applications [5, 7, 18, 31, 32, 59]. Solvers can be *direct*, if they find $x$ directly given $A$ and $b$, or *iterative*, if they start from an approximate $x$ and iterate until finding an exact solution. Efficient direct solvers rely on *factoring* matrix $A$, i.e., decomposing it into a set of matrices with a certain structure that makes $Ax = b$ easy to solve.

In this paper, we focus on *Cholesky* and *LU* factorization, the most efficient and widely used techniques for square matrices [24] (e.g., MATLAB adaptively chooses among these algorithms when solving systems of linear equations [15]).

```
1  M = lower triangle of A
2  for i in range(n):
3    M[i,i] = sqrt(M[i,i])
4    # Factor ith col: M[i+1:,i] *= (1 / M[i,i])
5    for j in range(i+1,n):
6      M[j,i] *= (1 / M[i,i])
7    # Outer prod update: M[i+1:,i+1:]-=
8    # outer(M[i+1:,i],M[i+1:,i])
9    for j in range(i+1,n):
10     for k in range(i+1,j+1):
11       M[j,k] -= M[j,i] * M[k,i]
```

**Listing 1: Cholesky factorization loop nest.**

Cholesky factorization is simpler and more efficient than LU, but it requires the $A$ matrix to have a particular structure: it needs to be symmetric (i.e., $A = A^T$), and positive-definite (i.e., for all non-zero column vectors $z$, $z^T Az$ must be positive). Cholesky finds a lower-triangular matrix $L$ such that $A = LL^T$. This enables solving $Ax = b$ via two triangular solves: $Ly = b \rightarrow L^T x = y$. Triangular matrices are simple to solve via substitution: when $A$ is a dense $n \times n$ matrix, triangular solves are $O(n^2)$, whereas factorization is $O(n^3)$ and dominates performance. Factorization also dominates performance when the matrix is sparse.

LU factorization instead finds lower-triangular matrix $L$ and upper-triangular matrix $U$ such that $A = LU$. Like Cholesky, this enables solving $Ax = b$ through triangular solves, but does not require $A$ to be symmetric positive-definite.

Real-world $A$ matrices are often extremely sparse. These matrices typically arise from discretizing equations, where each entry represents an interaction between two variables. In circuit simulations, each variable represents a circuit node [17], while for partial differential equations on meshes, each variable represents a node on the mesh [33, 58]. Each node has a small number of neighbors, irrespective of the overall system size. Consequently, matrices arising from larger simulations become increasingly sparse.

Since Cholesky factorization is simpler, in this section we use Cholesky to introduce the algorithmic techniques and optimizations in sparse matrix factorization. Section 2.4 discusses the differences between Cholesky and LU.

### 2.1 Basic Cholesky factorization

Listing 1 shows the code for a basic in-place implementation of Cholesky. $M$ is a lower-triangular matrix, initialized with the lower triangle of $A$ (because $A$ is symmetric, only its lower triangle is needed). Each outer loop iteration in Listing 1 modifies $M$ and produces a single column of the output $L$ in-place; after execution, $M$ contains the output $L$.

Specifically, the $i^{th}$ outer loop iteration in Listing 1 consists of two distinct activities. First, lines 2–5 overwrite the $i^{th}$ column of $M$, producing its final value. Second, lines 9–11 update columns $j > i$ by computing the *outer product* of the $i^{th}$ column of $M$ with itself, and subtracting this outer product from the rest of the matrix. Figure 1 shows these steps in detail for a sparse matrix.

This basic loop nest shows two key features of Cholesky. First, *data dependences are frequent*: each iteration updates the remainder of matrix $M$, so each output element incorporates contributions from many inputs. Second, *outer products dominate performance*: if the $i^{th}$ column of $M$ has $nnz$ nonzeros, updating the column takes $O(nnz)$ operations, whereas outer-product updates take $O(nnz^2)$.

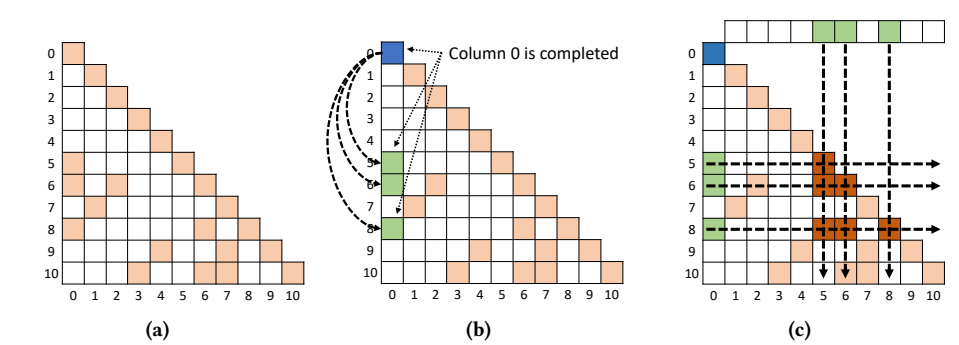


**Figure 1: Performing a single step of Cholesky factorization to a sparse matrix: (a) initial matrix $M$; (b) result of executing lines 3–6 of Listing 1; (c) result of applying the first column's outer product update to the rest of the matrix (lines 8–10 of Listing 1).**

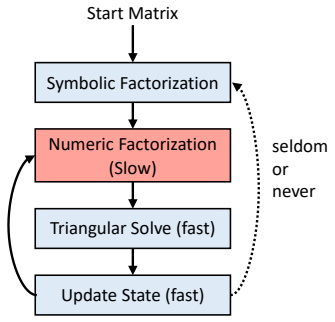


**Figure 2: Example solver application. Numeric factorization dominates performance.**

## 2.2 Challenges of sparse factorization

Sparse linear algebra algorithms must use *compressed formats* that leverage sparsity by representing only nonzero values. While they are space-efficient, compressed formats limit the operations that can be efficiently performed [9, 60]. A key challenge in sparse matrix factorization is using a compressed format that allows all necessary operations.

Typical sparse matrix formats like compressed sparse row/column (CSR/CSC) work poorly in Listing 1. Consider using CSC, which stores each column as a sorted list of nonzero coordinates and their values. CSC makes sequential traversals of columns efficient, so the updates to the $i^{th}$ column (lines 5–6) are efficient. But the outer-product updates (lines 9–11) would be extremely difficult, because they require updating individual values scattered throughout the matrix structure. These updates would require expensive searches in CSC, and often introduce new nonzeros, which would require rewriting the CSC structure.

The new nonzeros introduced by outer-product updates are known as *fill-in*. For example, in Figure 1c, $(6, 5)$ and $(8, 5)$ become nonzero. It is common for the final matrix $L$ to have substantially more nonzeros than $A$, e.g., 10–150× is typical in our experiments. But since $A$ is highly sparse (e.g., with a fraction $10^{-5}$ of nonzeros), so is $L$, and storing $M$ uncompressed would be very inefficient.

Prior work makes the key observation that *outer-product updates have substantial structure*. Figure 3 shows that the outer product of a sparse vector $v$ with itself produces nonzeros at all points in $nonzeros(v) \times nonzeros(v)$. This structure can be captured with a format that we call ***C**ompressed Cartesian **Sq**uare* (CSQ),[1] shown in Figure 3: a $k \times k$ CSQ consists of $k^2$ values and only $k$ coordinates, which denote the row (and column) coordinates of the nonzeros. When we compute the outer product of $v$ with itself, the resulting CSQ will be symmetric, so we only need to store its lower triangle.

Efficient sparse Cholesky implementations represent $M$ using *multiple* CSQ matrices, which are updated over time. Multiple matrices are needed because different outer-product updates have different sparsity patterns. Since nonzeros added by outer-product updates dominate the initial nonzeros of $A$, this representation consists almost entirely of nonzero values, leveraging sparsity. Moreover, the CSQ format enables efficient computation of outer products: since nonzeros are stored contiguously, this is equivalent to computing the outer product of a small dense vector, and can be performed efficiently with a vector processor or a systolic array.

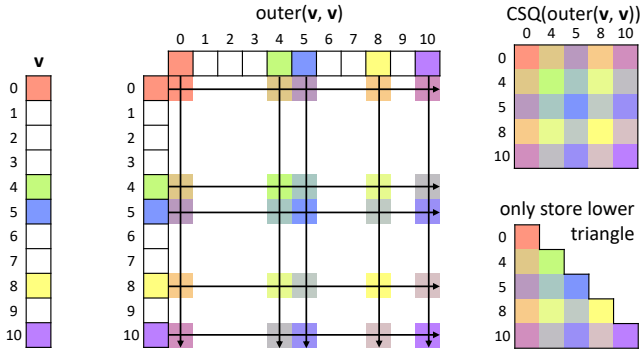


**Figure 3: Outer products can be stored in compressed format.**

## 2.3 Multifrontal sparse factorization

The widely used multifrontal algorithm [19] organizes computation as a *tree* of operations on matrices in CSQ format. This compressed format enables using dense linear algebra primitives, and is used in many factorization packages such as MUMPS [2], STRUMPACK [22], and UMFPACK [13].

**Symbolic factorization:** The multifrontal algorithm relies on a *preprocessing step* called symbolic factorization that analyzes only the matrix's *nonzero pattern* and creates helper data structures. In most applications, the nonzero pattern is static or changes very infrequently. For example, when simulating a circuit, devices do not gain new neighbors, and when simulating a car collision, most of the mesh describing the car retains the same structure. As a result, this step can be amortized across many numeric solves, making its performance costs a secondary concern [14]. To illustrate the bigger picture and where sparse matrix factorization fits within it, Figure 2 shows the general structure of many applications that use sparse matrix factorization.

[1] Prior work uses this format but does not give it a name. We give it a name for clarity. The name comes from the fact that matrix's nonzero coordinates are the Cartesian square (i.e., the Cartesian product with itself) of the nonzero coordinates of $v$.

```
for sn in postorder(etree):
  F = Fs[sn]
  # gather updates from all children
  for child in children(sn):
    gather_updates(F, Us[child])

  # factor the current supernode
  for i in range(N[sn]):
    F[i,i] = sqrt(F[i,i])
    F[i+1:,i] *= (1 / F[i,i])
    F[i+1:,i+1:] -= outer(F[i+1:,i],
                          F[i+1:,i])

  # store the rest as an update to the parent
  Us[sn] = F[N[sn]:,N[sn]:]
```

**Listing 2: Multifrontal Cholesky pseudocode.**

**Numeric factorization:** After preprocessing, the sparse factorization is described as an *elimination tree* [56] of *supernodes* $F_k$, each represented by a CSQ matrix. Figure 4 shows an elimination tree for the matrix from Figure 1a. For a supernode $F_k$, the first $N_k$ columns of $F_k$ will contain a subset of $M$'s columns that is determined by symbolic factorization. For example, in Figure 4, the nonzeros from columns $M[:, 0]$ and $M[:, 5]$ are stored in the first two columns of $F_0$. $F_k$'s remaining $U_k$ columns are used to store the results of the outer products of the first $N_k$ columns.

Concretely, in Figure 4, the outer product of $M[:, 0]$ with itself will produce nonzero values at $(5, 6, 8) \times (5, 6, 8)$. *By construction*, these values can all be stored in $F_0$ and represented efficiently using the CSQ format. For each supernode in Figure 4's elimination tree, the first $N_k$ columns are shaded and the remaining $U_k$ columns are left blank.

During the actual numeric factorization, the algorithm traverses the tree from leaves-to-root, executing the pseudocode in Listing 2. At each supernode $F_k$, the first step is gathering updates from child CSQ matrices.[2] Updates need to be accumulated *by coordinate*. For example, in Figure 4, when $sn = F_6$, $F_0[6, 6]$ and $F_2[6, 6]$ would be added to $F_6[6, 6]$, and $F_0[8, 6]$ would be added into $F_6[8, 6]$. Importantly, the same coordinate will almost always map to different *positions* (i.e., actual memory locations) in the parent and child CSQ matrices.

After all updates have been gathered, the algorithm runs $N_k$ outer-loop iterations of Cholesky factorization (Listing 1) on $F_k$. This step produces the final output columns. For example, after running two Cholesky outer-loop iterations on $F_0$, columns $M[:, 0]$ and $M[:, 5]$ will be *fully factored* and will not be updated again.

The fact that the last $U_k$ columns of any supernode $F_k$ need to be gathered into $parent(F_k)$'s CSQ imposes a *data dependence*. The algorithm cannot begin factoring $parent(F_k)$ before $F_k$ has been fully factored. As long as data dependences are respected, supernodes can be factored in parallel. Listing 2 performs a postorder traversal of the elimination tree, which ensures that all children are factored before their parents. This ordering is correct, but in Section 5, we will refine this ordering to improve performance.

This algorithm is efficient because factoring each supernode, which has *cubic* complexity, is efficient on CSQ matrices. Gather-updates have low arithmetic intensity, but there is only a quadratic number of updates, so factoring dominates.

[2]Prior work sometimes calls this operation "extend add" [2, 39].

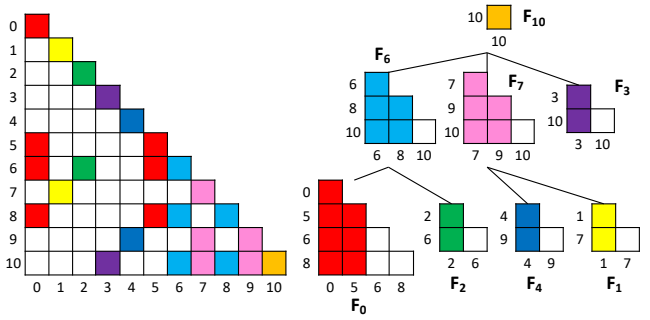


**Figure 4: Supernodal elimination tree for $L$. When Listing 2 runs on this tree, it performs 2 steps of factorization on $F_0$, 1 step of factorization on $F_2$, then adds $F_0(6, 6)$, $F_0(8, 6)$, $F_0(8, 8)$, and $F_2(6, 6)$ into their respective entries in $F_6$ before factoring $F_6$. Colored supernode cells represent where columns are factored. Note that each column is factored exactly once.**

## 2.4 LU factorization

This section has focused on Cholesky factorization, but almost all of the explanations apply to LU factorization too.

LU factorization is similar to Cholesky factorization, but is not limited to symmetric positive-definite (SPD) matrices. Whereas Cholesky factorization finds $L$ such that $A = LL^T$, LU factorization finds a lower-triangular matrix $L$ *as well as* an upper-triangular matrix $U$ such that $A = LU$. This allows $LU$ factorization to work on non-symmetric matrices, which also means $A$'s upper triangle needs to be stored and computed on. $LU$ factorization requires $\approx 2\times$ more FLOPs than Cholesky factorization. To preserve numeric stability, we use static pivoting as a preprocessing step [37]. Using static pivoting, we are able to use a similar loop nest to Listing 1, but change the loop bounds to materialize distinct values in the upper triangle. This implementation of sparse $LU$ shares the same types of data dependences as sparse Cholesky.

# 3 SPARSE FACTORIZATION IS INEFFICIENT ON PRIOR ARCHITECTURES

We now describe why existing architectures are ill-suited to sparse factorization algorithms, motivating the need for a new accelerator. Section 3.1 and Section 3.2 quantitatively analyze the performance of state-of-the-art GPU and CPU implementations, showing that despite *extensive* hardware-aware optimizations, utilization is often dismal. Section 3.3 discusses other proposed accelerators, explaining why they lack key ingredients to handle sparse factorization.

## 3.1 GPU implementations

Given their wide availability and high peak compute throughput, GPUs have become the hardware of choice for accelerating linear algebra applications. There are many sparse matrix factorization packages that leverage GPU acceleration; we will analyze CHOLMOD (Cholesky) [54] and STRUMPACK (LU) [22], as we found that they achieved the best performance across a wide range of sparse matrices.

Figure 5 (left) reports the performance of STRUMPACK when running on an NVIDIA V100 GPU, which provides a peak double-

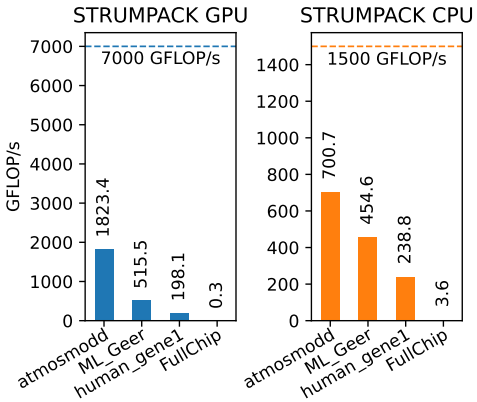


**Figure 5: Performance of STRUMPACK (GFLOP/s, higher is better) on NVIDIA V100 GPU and 32-core AMD Zen2 CPU on four representative sparse matrices.**

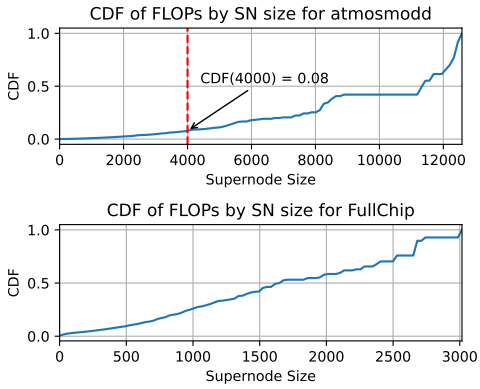


**Figure 6: CDFs of FLOPs by supernode size for the two extreme matrices in Figure 5, showing that matrices with lower utilization have more FLOPs in smaller supernodes.**

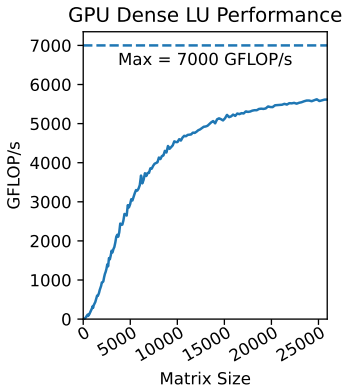


**Figure 7: Performance of *dense* LU factorization (without pivoting) across matrix sizes on an NVIDIA V100 GPU.**

precision floating-point throughput of 7000 GFLOP/s. Figure 5 reports performance in GFLOP/s, showing how well the GPU is utilized, on four representative matrices (our evaluation uses a larger set; see Section 7.1 for methodology details).

Figure 5 shows a wide range of efficiencies across matrices: while `atmosmoddd` achieves 26% of the GPU's peak throughput, `human_gene1` achieves only 2.8%, and `FullChip` achieves a dismal 0.004% of peak throughput—only 0.3 GFLOP/s!

This wide range of performance happens because *GPUs are inefficient when factoring small CSQ matrices.* As we have seen in Section 2.3, sparse factorization mainly consists of *outer-product updates* on supernodes that are much smaller than the full matrix, and are stored compactly in CSQ format.

Different matrices have different nonzero patterns, which induce different symbolic factorizations with a wide range of supernode sizes. Figure 6 shows the cumulative distribution function (CDF) of FLOPs across supernode sizes for the two matrices at the extremes of efficiency: `atmosmoddd` (top) and `FullChip` (bottom). In each graph, the $x$-axis is supernode size as the number of rows and columns (e.g., 4000 denotes a 4000×4000 supernode), and the line reports the fraction of total FLOPs (i.e., work) that happens on supernodes of size $\leq x$. In `atmosmoddd`, 8% of the work happens on supernodes of size $\leq 4000$, i.e., 92% of work happens on supernodes of size $> 4000$. By contrast, in `FullChip`, the *largest* supernode has size 3047.

To a first order, we can approximate the work in each supernode as the factorization of a dense matrix. Figure 7 shows the performance of the V100 GPU on *dense* factorization as a function of matrix size, in GFLOP/s. Performance flattens around size 20000, and drops linearly below 10000, so small matrices suffer very low throughput. This mostly explains the dismal performance of GPUs on matrices like `FullChip`: small supernodes hamper utilization.

While the above analysis is a good first-order approximation, it is not the full story, because it assumes that each supernode is factored in series. This would result in even worse SM utilization. Instead, recent work uses *batching* [1, 22, 54]: grouping small supernodes at the same depth in the elimination tree into a single kernel, as shown in Figure 8.

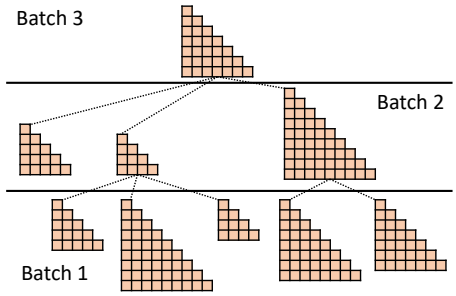


**Figure 8: Batching groups supernode factorizations.**

The GPU implementations we compare against use batching to improve utilization by exploiting parallelism across supernodes and amortizing kernel launch overheads. However, batching is a crude way to handle data dependences that misses many opportunities for parallelism. Batching also causes load imbalance because it groups supernodes of different sizes, as Figure 8 shows, causing poor SM utilization. Finally, level-by-level traversal of the elimination tree eliminates opportunities for data reuse, incurring additional DRAM traffic. It would be more efficient for the parent supernode to consume child updates immediately after they are produced, as we will see in Section 5.2. As a result of these architectural limitations, GPUs achieve poor utilization, as Figure 5 shows.

In summary, while GPUs have plentiful computational throughput, they are ultimately limited by irregular shapes and data dependences across supernodes, which destroy utilization. Spatula solves this with a more flexible organization that *(1)* achieves much higher performance on small matrices, and *(2)* gracefully handles complex data dependences to unlock much more parallelism within and across supernodes.

## 3.2 CPU implementations

Given the limited utilization of GPUs, it is important to consider CPU implementations as well. Figure 5 (right) shows STRUMPACK's performance across the same set of sparse matrices when running on a 32-core/64-thread AMD Zen2 CPU at 3.5 GHz. While the CPU

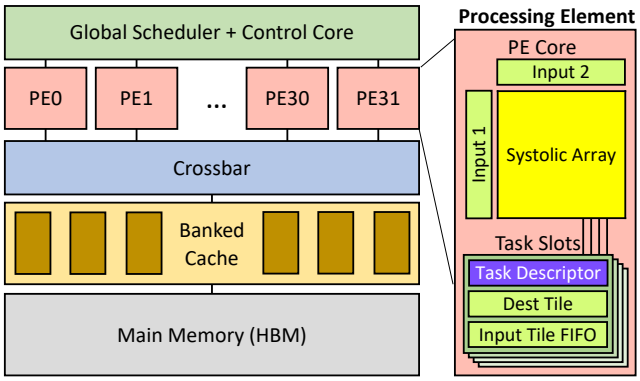


**Figure 9: Spatula architecture overview.**

implementation significantly outperforms the GPU on `FullChip` and slightly outperforms it on `human_gene1`, it still suffers from low compute utilization. CPUs overcome some of the limitations of their lower peak throughput with more flexible scheduling hardware, but they still suffer dismal utilization on many difficult matrices.

**Hybrid CPU-GPU:** Prior work has also considered hybrid CPU-GPU approaches [21, 40], However, the increasing gap in FLOP/s between CPUs and GPUs combined with costly host-accelerator communication has resulted in these approaches lagging behind GPU-only techniques [54].

### 3.3 Other accelerators

**ASIC accelerators for dense factorization:** REVEL [65] and TaskStream [10] are hardware accelerators that support Cholesky factorization of small *dense* matrices. However, they do not support the sparse case. The difficulty in sparse matrix factorization is not primarily speeding up single-supernode kernels, but rather efficiently handling differently sized supernodes without major load imbalance. Additionally, data movement is not a significant concern for a single small dense Cholesky, but becomes crucial when dealing with a large tree of supernodes, where balancing parallelism and data movement is key to achieving good performance.

**FPGA accelerators for sparse factorization:** Prior work has proposed using FPGAs to accelerate sparse factorization. But the limited arithmetic throughput of FPGAs makes them ill-suited to factorization. Nechma et al. [46] and Kapre et al. [30] describe designs with peak throughput of <10 GFLOP/s. Due to their low throughput, they underperform state-of-the-art CPU implementations. Furthermore, these designs use the Gilbert-Peierls algorithm [23] which does not efficiently scale to higher compute throughputs.

## 4 SPATULA ARCHITECTURE

Figure 9 shows an overview of Spatula's hardware architecture. Spatula combines several unique features that enable high-performance sparse factorization. First, Spatula features *processing elements (PEs)* that achieve high performance on small matrices. Each PE features a 16×16 systolic array that executes factorization tasks at high throughput. Supernodes are processed in 16×16 tiles. This design enables high performance when factoring small supernodes, which as we have seen in Section 3, are common and performance-critical. Second, Spatula features *fine-grained hardware support for scheduling* needed to handle frequent data dependences. Each PE is double-buffered to hide latency, and a global scheduler dispatches tasks across PEs. Third, Spatula has a memory hierarchy tailored to the needs of factorization workloads: a banked on-chip cache captures irregular reuse of tiles, and high-bandwidth memory provides adequate throughput.

### 4.1 Tiles are Spatula's primitive datatype

The multifrontal algorithm (Section 2.3) runs sparse factorization as a tree of computations on CSQ matrices. The CSQ format enables processing sparse data using dense linear algebra kernels. These dense kernels can be *tiled*, which enables handling CSQ matrices of varying sizes with a fixed hardware tile size.

Spatula's primitive datatype are $T \times T$ *dense* tiles of double-precision floating-point numbers. A CSQ matrix can be divided into tiles of a fixed size using *position-based* tiling [60], as shown in Figure 10.

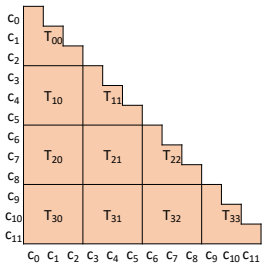


**Figure 10: Dividing a supernode CSQ into fixed-size tiles.**

Tile size is an important parameter. Larger tiles allow Spatula to achieve a particular throughput with fewer, higher-throughput PEs, allowing for a simpler on-chip network and amortizing scheduling overheads. However, they increase the granularity of computations, leading to possible under-utilization.

Sweeping tile sizes, we find that $T = 16$ achieves the highest performance across a wide range of matrices. Each tile's values are stored contiguously in memory, along with the coordinates of each row and column.

### 4.2 Task-based programming model

Spatula uses a task-based programming model. Each task takes a set of tiles as inputs and accumulates the results into a single output tile. Spatula supports several types of tasks, all shown in Table 1. Each tasks runs on one PE, and each PE can run tasks of any type.

The multifrontal algorithm (Section 2.3) is decomposed into a collection of tasks. We now present this decomposition; Section 4.3, which describes the implementation of PEs, gives more details on the internal structure of each task.

Figure 11 illustrates how the code on lines 8–12, which factors a supernode, is mapped into Spatula tasks. First, the top-left corner

| Task Type | Computation | Input Types |
|---|---|---|
| `dgemm` | `D += gemm(hcat(A), vcat(B))` | `A: list<Tile>` `B: list<Tile>` |
| `tsolve` | `D = tri_solve(D, A)` | `A: Tile` |
| `dchol` | `D = dense_chol(D)` | |
| `dlu` | `D = dense_lu(D)` | |
| `gather_updates` | `for T in A: D += T` | `A: list<Tile>` |

**Table 1: Spatula task types. Note that `D` is always a `Tile`, and it is both an input to the task and its sole output.**

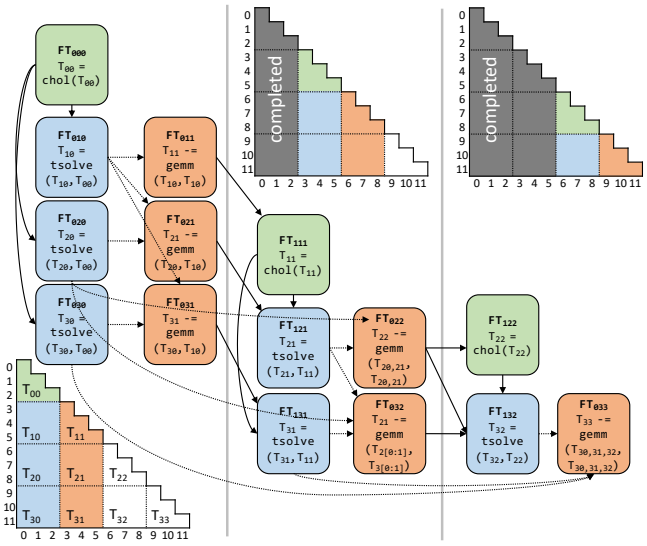


**Figure 11: Data-dependence graph of tasks needed to perform 9 outer-loop iterations of factorization on a tiled CSQ supernode.**

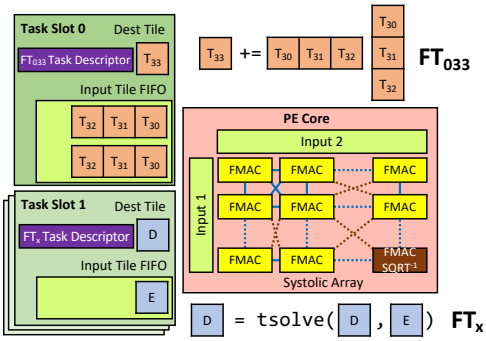


**Figure 12: A snapshot of a PE's state. $FT_{033}$ is drawn from Figure 11, and $FT_x$ is a task from a different supernode. Vertical and horizontal links are part of the basic systolic array, and diagonal links are added to support `dchol` and `dlu`.**

tile is factored using a `dchol` (dense Cholesky) task. This task's output tile, $FT_{000}$, is then consumed by a set of `tsolve` (triangular solve) tasks that process the leftmost column of tiles. These tasks' outputs, $FT_{010}$, $FT_{020}$, and $FT_{030}$, are then consumed by a set of `dgemm` tasks that process the second column of tiles. Once the `dgemm` tasks in the second column complete, tasks $FT_{111}$, $FT_{121}$, and $FT_{131}$ are able to execute, and the algorithm continues like this. Note that `dgemm` tasks in the third column take inputs from both the first and second columns. For example, $T_{022}$'s input $T_{1[0:1]}$ is produced by $FT_{121}$. These deep chains are due to the dependences discussed in Section 2: each `dgemm` task computes and accumulates all the outer-product updates for its tile.

Finally, the last task type, `gather_updates`, is used to gather updates across supernodes (lines 4–5 of Listing 2).

## 4.3 Processing elements

As shown in Figure 9, Spatula has 32 processing elements (PEs). Each PE features a systolic array that can execute all types of tasks. Systolic arrays are ideally suited to `dgemm`, the most common task type. The key design choice is whether to have homogeneous PEs that can handle all tasks, or heterogeneous PEs specialized to each task. We opt for a homogeneous approach for two reasons. First, the mix of different task types *varies across matrices*. For example, `Serena` has 99.4% of its FLOPs in `dgemm` tasks, whereas `G3_Circuit` has just 85% of its FLOPs in `dgemm` tasks. Second, other types of tasks can also benefit from systolic arrays, and these tasks are often in the critical path (e.g. `dchol` in Figure 11), so running them quickly helps overall performance.

Prior work has already proposed *different* systolic arrays for each task [6, 34]; we adapt and combine these ideas to produce a single systolic array that can handle all. This design adds < 5% area overhead vs. an array that can only run `dgemm`. Below, we explain how we build up the array task by task.

**Basic systolic array for `dgemm`:** `dgemm` multiplies $n$ pairs of input tiles, interpreted as matrices $A$ and $B$ of sizes $T \times nT$ and $nT \times T$, and accumulates the result into a $T \times T$ tile $D$. Figure 12 (top) shows task $FT_{033}$ from Figure 11 as an example.

We start with a standard systolic array, shown in Figure 12, which consists of a grid of $T \times T$ FMAC units with pipelined horizontal and vertical connections. The destination tile $D$ is first loaded into the systolic array. Then, a column of $A$ and row of $B$ are fed to the array each cycle, and the array computes and accumulates partial products following an output-stationary (inner-product) dataflow. Finally, the output tile $D$ is read out row by row. The array is double-buffered (detailed later) to hide startup latencies, so its throughput is one `dgemm` per $nT$ cycles.

**Handling dense factorization (`dchol`/`dlu`) tasks:** Tasks corresponding to top-left tiles, such as $FT_{000}$ in Figure 11, are small dense Cholesky factorizations (`dchol`), and require additional hardware support to execute on a systolic array.

We augment the systolic array to implement `dchol` using Brent et al.'s algorithm [6]. We extend the ALU at one of the corners of the array to support a division and square-root operation, as Figure 12 shows. We also add diagonal links between ALUs, and augment the PE's finite state machine (FSM) to support a different dataflow: input tile $D$ is streamed into the array row by row (as in `dgemm`), but values cycle through the array so that all elements in the diagonal pass through the inverse-square-root ALU. Outputs are streamed into the PE's accumulator, then written back to the cache.

Systolic dense Cholesky factorization tasks are *latency-bound*, each having a critical path of $T$ inverse-square root operations. These tasks under-utilize the array's FMAC units, but they are uncommon, as each supernode has a linear number of these tasks, vs. a cubic amount of `dgemm` tasks. `dlu` tasks also leverage the same hardware modifications.

**Handling triangular solve (`tsolve`) tasks:** `tsolve` tasks need a much smaller set of extensions to implement Kung et al.'s algorithm [34]. Values of the read-only $A$ input are streamed downwards through the array, while values in each row of the read-write $D$ input are cycled through a row of ALUs. These tasks do not require any additional ALU hardware or links between ALUs over the above extensions, just additional control FSM logic to enable this dataflow.

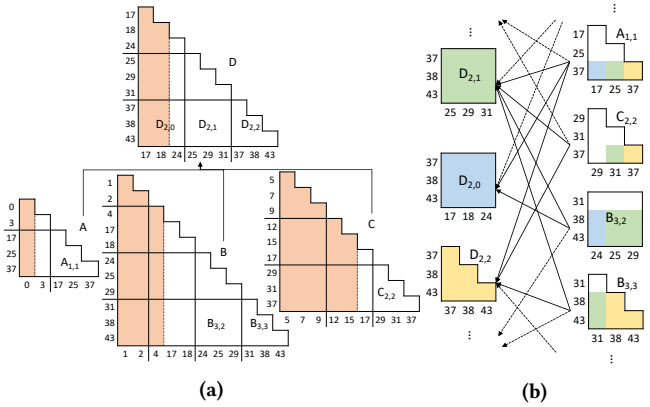


**Figure 13: Given the elimination tree with tiled CSQ supernodes shown in (a), (b) illustrates the the many-to-many `gather_update` dependence structure.**

**Handling gather-update tasks:** `gather_update` tasks use the PE just for addition. These tasks are responsible for adding values with matching coordinates from a tile of a child's CSQ matrix into a tile of the parent's CSQ matrix. Figure 13a shows a detailed example, where updates (the unshaded region of the CSQ matrix) from matrices $A$, $B$, and $C$ need to be accumulated into to $D$. However, at a tile level, the coordinates of each tile *do not match*. As shown in Figure 13a, tile $D_{2,1}$ requires updates from tiles $B_{3,2}$, $B_{3,3}$, $C_{2,2}$, and $A_{1,1}$. Coordinates on both axes in each tile are guaranteed to be strictly increasing, allowing addition to be implemented by shifting input rows into the correct position.

**Task and data orchestration:** To achieve full PE utilization, PEs employ two different latency-hiding mechanisms. First, as shown in Figure 9, each PE has multiple task slots (four in our implementation) to decouple data accesses from execution. Each slot can hold a different task, and the scheduler dispatches tasks to PE slots. When a task arrives at a slot, the PE starts loading the task's input tiles while the PE is executing a task in another slot. This lets the PE hide the latency of memory accesses (from cache hits or misses).

Second, the systolic array *double-buffered*: each array element has two sets of input and accumulator registers. While one task is executing, the PE can load the next task's operands into the array's accumulator registers. This lets the PE hide any startup latency when moving from one task to the next.

Tasks become runnable when all of their input operand loads are completed and data is available. If none of the PE's slots contains a runnable task, the PE stalls. As the PE executes a task, it will draw inputs from the input tile FIFO associated with the current task. Upon finishing a task, the PE writes the updated destination tile back to Spatula's cache.

## 4.4 Hardware scheduler

Efficiently executing sparse matrix factorizations requires dynamic scheduling hardware. There are two sources of parallelism: fine-grained *intra-supernode* parallelism, where independent tasks from a single supernode can run in parallel, and coarser-grained *inter-supernode* parallelism, where tasks from independent supernodes can run in parallel.

Due to the wide range of supernode sizes, achieving good utilization requires exploiting *both* intra- and inter-supernode parallelism. Figure 14 shows this by comparing Spatula's performance on several matrices under three scheduling policies. **Inter** dispatches each supernode to a different PE, exploiting only inter-supernode parallelism. By exploiting coarse-grain parallelism, **Inter** is very simple to implement and needs minimal hardware support. Unfortunately, it achieves terrible utilization. Recall from Figure 6 that most matrices have large supernodes with ample intra-supernode parallelism. Binding a single supernode to each PE results in running these large supernodes serially and bottlenecking performance. For example, when factoring the root supernode, there are no other available supernodes, so only one PE would be utilized.

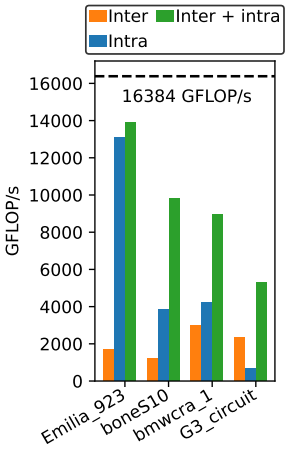


**Figure 14: Comparing scheduler designs.**

By contrast, **Intra** runs one supernode at a time across all PEs, exploiting only intra-supernode parallelism. **Intra** requires a hardware scheduler, as with 32 PEs, it must dispatch a task every few cycles, and must negotiate complex inter-task dependences. **Intra** works well when large supernodes dominate, e.g., in `Emilia_923`, but works poorly when small supernodes are frequent. For instance, **Intra** suffers from 4.0% compute utilization in `G3_circuit`, as its small supernodes cannot use all the PEs. This is the same reason why GPU implementations use batching (Section 3).

**Intra+inter** is Spatula's scheduling policy, which exploits both intra- and inter-supernode parallelism. This policy first exploits fine-grain parallelism within a single supernode to keep data footprint low, but overlaps multiple small supernodes when more parallelism is necessary. Figure 14 shows that **Intra+inter** achieves high utilization.

We now discuss the hardware support that enables Spatula's scheduling policy; Section 5 presents the policy itself.

Figure 15 shows Spatula's scheduler, which follows a *two-level* design: a *supernode scheduler* feeds ready supernodes to a *task scheduler*, which produces the fine-grained tasks of each supernode and dispatches them to PEs.

**Supernode scheduler:** The supernode scheduler determines the coarse-grain schedule by controlling the processing order of supernodes. This unit requires limited throughput, producing at most one supernode per 100 cycles (and often much less). Thus, for flexibility, we implement this unit using a RISC-V control core. The core feeds the task scheduler through a queue of ready supernodes, and consumes supernode completion notifications from the task scheduler.

**Task scheduler:** The task scheduler determines the fine-grain schedule of computation. This scheduler is implemented using dedicated hardware, because it requires high throughput (producing one task every 3 to 20 cycles) and it must handle tight data dependencies with low latency.

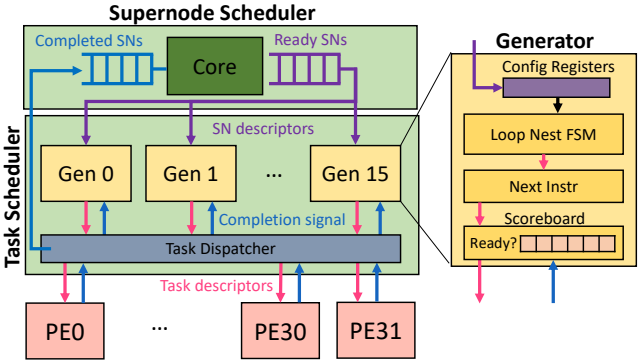


**Figure 15: Spatula scheduler design, with 16 generators.**

The task scheduler is built around a set of *generator* units (16 in our implementation). A generator is a simple, configurable FSM that produces all the tasks to process one supernode. Each generator is first configured with the information of the supernode, including its location in memory and dimensions. Then, the generator emits a sequence of tasks that process the supernode. Once the generator finishes producing tasks for its current supernode, it can be reused for a different supernode. A *task dispatcher* consumes this sequence of tasks and dispatches them to PEs, filling their task slots greedily.

Each generator releases a task to the dispatcher only when the task is ready, i.e., when all its inputs have been computed. To this end, each generator keeps a *completion scoreboard* that tracks which inputs are available. Due to the structure of the computation, this information is easy to track, requiring $k$ $log_2 k$-bit entries for a $k \times k$-tile supernode (this encodes the last available column tile for each row tile). Because we use multi-level tiling (Section 5), $k$ is limited to 80, and this scoreboard takes about 500 bits of state, with simple wakeup logic similar to that of a scalar scoreboarded core [61].

## 4.5 Memory hierarchy

In our design, we opt for a cache instead of scratchpad memory. As discussed in Section 4.4, the scheduler dynamically interleaves tasks from different supernodes depending on the readiness of their inputs. The resulting access pattern cannot be known at compile-time, requiring the use of a cache.

We use an LRU cache with large cache lines. As dense $16 \times 16$ tiles are our primitive datatype, we can utilize large tile-sized cache lines (2KB in our implementation). We have relatively few PEs and relatively large data transfers, so a full crossbar is practical. Every cycle, we have 32 PEs, each of which consumes 32 double-words of data per cycle, resulting in a total of 8 TB/s bandwidth needed to feed our PEs when they are all active. This network configuration consumes relatively little area and energy, as we will see in Section 7. Our scheduler issues memory accesses ahead of time, when a task group is first scheduled onto a PE, achieving a high degree of decoupling and limiting memory stalls.

# 5 SCHEDULING

Section 4.4 showed that exploiting intra- and inter-supernode parallelism is necessary, and presented the scheduling hardware needed to do so. We now present our scheduling algorithm, showing how we negotiate parallelism and footprint to achieve high utilization. To reduce data footprint, our general strategy is to exploit finer-grain parallelism first. Section 5.1 describes how Spatula schedules within a supernode, and Section 5.2 describes how Spatula dynamically overlaps supernodes when intra-node parallelism is insufficient.

## 5.1 Intra-supernode scheduling

Effectively scheduling a single supernode is important. Though Figure 14 shows that running a single supernode at a time is insufficient, exploiting available intra-supernode parallelism lets Spatula achieve the same degree of parallelism with fewer concurrent supernodes, reducing cache footprint.

**Breadth-first task order:** Intra-supernode schedules are implemented in hardware by the generator FSMs described in Section 4.4. For simplicity, generators produce tasks in a *fixed order*, and dispatch tasks in this order to PEs (i.e., out-of-order completion is possible, but not out-of-order dispatch).

Due to frequent dependences, different task orders produce vastly different performance. However, we observe that due to the structure of the computation, a *breadth-first task order* produces a nearly optimal schedule. This is simply a loop nest that corresponds to a breath-first traversal of the task dependence graph shown in Figure 11.

Simpler orders, like following a fixed-dimension order, fail to exploit the available parallelism and are multiple times slower on small supernodes with frequent dependences. Conversely, we explored an aggressive dataflow scheduler that issues tasks out-of-order, and found negligible overall performance gains overall, and less than 10% in all cases. Thus, we opt for this simple breadth-first order.

**Multi-level tiling:** Some matrices have supernodes whose size vastly exceeds on-chip storage. For instance, `atmosmodd`'s largest supernode (Figure 6) is 12709×12709, *over 1 GB!*

Handling these large supernodes efficiently requires additional levels of tiling. In Section 4, we saw factorization is amenable to tiling, and Spatula uses small $T \times T$ tiles (16×16 in our implementation) as its main primitive. This structure is fractal, and admits further tiling. We introduce level-2 *supertiles* of configurable size, $S \times S$ small tiles each. $S$ is configurable, and simply adds loop nest levels to the generator FSM, which produces tasks to compute outputs supertile by supertile.

We size each supertile to fit on-chip. For example, with $S = 70$, each supertile is 10 MB, with fits within Spatula's 16 MB cache. Most reuse happens within a supertile, allowing us to adopt the breadth-first task order without incurring additional data movement. Generators also process supertiles in breadth-first order, but this is to simplify logic: we have evaluated other supertile orders and performance is insensitive to ordering.

## 5.2 Inter-supernode scheduling

As discussed in Section 4.4, achieving high compute utilization requires running multiple supernodes concurrently. However, exploiting inter-supernode parallelism too aggressively risks blowing up the algorithm's cache footprint.

Spatula's general-purpose scheduling core enables flexibility when designing supernode ordering algorithms. To balance parallelism with memory footprint, Spatula opts for an algorithm that is based on a post-order traversal of the tree. A post-order traversal

| | Area ($mm^2$) |
|---|---|
| **PEs:** 32, 16×16 double-buffered systolic PEs, 1 GHz | 43.5 |
| **Scheduler:** 16-generator and RISC-V control core | 0.05 |
| **Cache:** 16 MB, 32 banks, 16-way, 2 KB lines, LRU, write-allocate, up to 256 concurrent misses | 17.6 |
| **NoC:** 5 32×32 (4 TB/s) crossbars | 16.7 |
| **Main memory:** 2 HBM2E PHYs (1 TB/s) | 29.8 |
| **Total** | **107.7** |

**Table 2: Configuration and area of Spatula as evaluated.**

minimizes footprint by visiting a parent supernode immediately after all its children.

However, Spatula's algorithm allows dynamic reordering to unlock inter-supernode parallelism. Specifically, code running on the scheduling core maintains a min-heap of ready supernodes keyed by their position in the post-order traversal. Whenever a new supernode is needed, the scheduler core yields the supernode at the root of the min-heap.

To minimize footprint, Spatula's task dispatcher follows a *biased* order: it tries to fill PEs with tasks from the generators with older supernodes, and uses more recent ones only when older supernodes have no ready tasks.

This policy automatically balances parallelism and footprint: when processing a large supernode, internal parallelism is plentiful, and the whole system focuses on that supernode, processed in cache-fitting supertiles. Other generators may have younger supernodes, but their tasks are not issued because the large supernode fills all PEs. Conversely, when processing many small supernodes with limited internal parallelism, the dispatcher overlaps the execution of these supernodes to keep high utilization. But this is fine footprint-wise, as each supernode is much smaller than the on-chip cache.

# 6 IMPLEMENTATION

We implement Spatula's components in RTL and synthesize it using Synopsys Design Compiler on commercial 12/14nm technology processes. We target a 1 GHz frequency. We assume 512 GB/s bandwidth per PHY, similar to the NVIDIA A100 GPU, which has 2.4 TB/s with 6 HBM2E PHYs [48]. We rely on prior work to estimate the PHY's area [11] and power [20].

Table 2 shows Spatula's configuration and its area breakdown by component. Spatula's design is balanced between computation and communication, with functional units taking up 43.5 $mm^2$ out of the total 108 $mm^2$ of area.

# 7 EVALUATION

## 7.1 Experimental methodology

**Evaluated systems:** We compare Spatula with two baselines: an NVIDIA V100 GPU, and an AMD Ryzen Threadripper PRO 3975WX CPU. Note these systems have higher area and TDP than Spatula at similar or more advanced nodes (specifically, the V100 is 815 $mm^2$ in TSMC 12nm) [47].

**Simulation:** We evaluate Spatula using a cycle-level simulator. Our simulator is based on a simulator for accelerators that has been used in prior work, including Gamma [68] and ISOSceles [67], but with customized timing models for PEs, scheduler, and memory system. The simulator is cycle-driven [41]: every hardware component is modeled as an object; every cycle, each object is ticked and its activity is simulated appropriately.

Spatula's simulator uses synthesis-derived detailed timing models for PEs, scheduler, caches, on-chip network, and main memory. Caches are banked, pipelined, and implement lookups with serial tag and data accesses, and we model bank access latency and contention. Banks are shared across PEs. The NoC connecting PEs and cache banks is modeled using bit-sliced crossbars as described by Passas et al. [51]. We model HBM2E's structure using Micron's specifications [43]. Each cache bank issues accesses to a single HBM2E channel. Because each cache line is 2KB, the size of the row buffer, memory accesses achieve high utilization.

As an optimization, PEs are simulated at a task granularity, but we do not model the cycle-by-cycle execution of each task (e.g., the timing of each individual ALU). Since tasks are executed systolically, once started, each task incurs a fixed latency that depends solely on tile size parameters encoded in the task descriptor. This enables us to fully simulate large matrices instead of depending on sampling-based approaches. We model all dynamic timings of PEs, e.g., each task does not start until all its operands have been fetched.

Finally, we check functional correctness against the baselines, and we compute power by combining activity factors from simulations with synthesis results.

**Selected configuration:** Spatula's default configuration uses the parameters in Table 2. We determined this configuration by sweeping the number of PEs, cache banks, HBM PHYs, and primitive tile size. We select a Pareto-optimal configuration with reasonable area and power. We use RTL synthesis to find the area and power of components other than main memory; we use prior work to estimate HBM2E power [20, 52] and PHY area [12, 52]. Section 7.3 evaluates larger and smaller Spatula designs, showing good scalability.

**Factorization algorithms:** We compare against state-of-the-art implementations of sparse Cholesky and sparse LU.

For sparse LU, we compare against STRUMPACK [22], a widely used sparse LU package optimized for both CPUs and GPUs. For sparse Cholesky, we compare against CHOLMOD [8]. CHOLMOD is widely used and is the standard sparse Cholesky implementation used in Matlab. We compare against the most recent versions (STRUMPACK 6.3 and CHOLMOD 7), except for CHOLMOD on GPU, where we use version 4.6.0-beta, which incorporates batching [54] and outperforms more recent versions. These packages use MKL on the CPU, and cuBLAS and cuSolver on the GPU.

**Input matrices:** We select a representative set of matrices from SuiteSparse [16]. We select symmetric positive-definite matrices for Cholesky, and leave the others for LU. For both Cholesky and LU, we select the 20 relevant matrices with the longest GPU execution time. Hardware acceleration is most needed for matrices that are time-consuming to factor. We find that among these top matrices, some have extremely similar structures, such as `Long_Coup_dt0` and `Long_Coup_dt6`. Additionally, some matrices are reduced versions of others, such as `bone010` and `bone010_M`. To achieve a more diverse set, we select only one matrix from each of these groups.

The resulting matrix sets have diverse structures and display a wide range of utilizations in CPU and GPU baselines. Our methodology produced a set of matrices that contains almost all the matrices evaluated by our baselines [22, 54].

| Matrix | Spatula TFLOP/s | vs. GPU | vs. CPU |
|---|---|---|---|
| Serena | 14.5 | 12.4 | 46.8 |
| Geo_1438 | 14.0 | 14.4 | 60.7 |
| Emilia_923 | 13.9 | 15.2 | 61.4 |
| Fault_639 | 13.8 | 18.3 | 67.0 |
| Hook_1498 | 13.5 | 20.4 | 141.2 |
| nd24k | 13.2 | 14.4 | 57.6 |
| audikw_1 | 13.2 | 25.9 | 90.8 |
| PFlow_742 | 12.9 | 200.4 | 93.9 |
| bone010 | 12.7 | 31.5 | 129.1 |
| StocF-1465 | 12.6 | 96.4 | 181.4 |
| Flan_1565 | 12.1 | 29.2 | 155.0 |
| consph | 10.6 | 239.2 | 179.0 |
| boneS10 | 9.8 | 121.0 | 664.1 |
| apache2 | 9.6 | 156.3 | 837.6 |
| offshore | 9.5 | 147.8 | 710.6 |
| inline_1 | 9.2 | 121.0 | 464.0 |
| bmwcra_1 | 8.9 | 120.2 | 399.9 |
| BenElechi1 | 8.3 | 242.7 | 628.4 |
| af_0_k101 | 8.1 | 169.5 | 651.6 |
| G3_circuit | 5.3 | 206.1 | 2273 |
| **gmean** | **11.0** | **61.3** | **212.7** |

**Table 3: Spatula performance on sparse Cholesky and speedups over GPU and CPU.**

| Matrix | Spatula TFLOP/s | vs. GPU | vs. CPU |
|---|---|---|---|
| cage13 | 14.4 | 11.4 | 17.7 |
| Long_Coup0 | 14.1 | 9.4 | 18.3 |
| nlpkkt80 | 13.7 | 8.2 | 16.5 |
| Ge87H76 | 13.5 | 6.9 | 19.0 |
| atmosmodd | 13.4 | 7.7 | 17.6 |
| Transport | 12.8 | 10.5 | 17.5 |
| language | 12.4 | 18.9 | 25.5 |
| ML_Geer | 11.6 | 21.6 | 23.6 |
| appu | 11.2 | 38.2 | 36.4 |
| dielFilterV3real | 11.2 | 32.9 | 25.1 |
| CoupCons3D | 11.1 | 26.4 | 29.1 |
| kkt_power | 11.0 | 29.7 | 24.0 |
| ASIC_680k | 10.6 | 294.9 | 45.6 |
| torso3 | 10.3 | 29.1 | 40.4 |
| ohne2 | 9.9 | 30.5 | 32.1 |
| F1 | 9.6 | 32.8 | 33.4 |
| human_gene1 | 8.7 | 37.8 | 31.5 |
| FullChip | 6.5 | 22030 | 1645 |
| TSOPF_b2383 | 4.9 | 105.8 | 30.3 |
| rajat31 | 4.8 | 99.2 | 45.8 |
| **gmean** | **10.4** | **36.2** | **32.5** |

**Table 4: Spatula performance on sparse LU and speedups over GPU and CPU.**

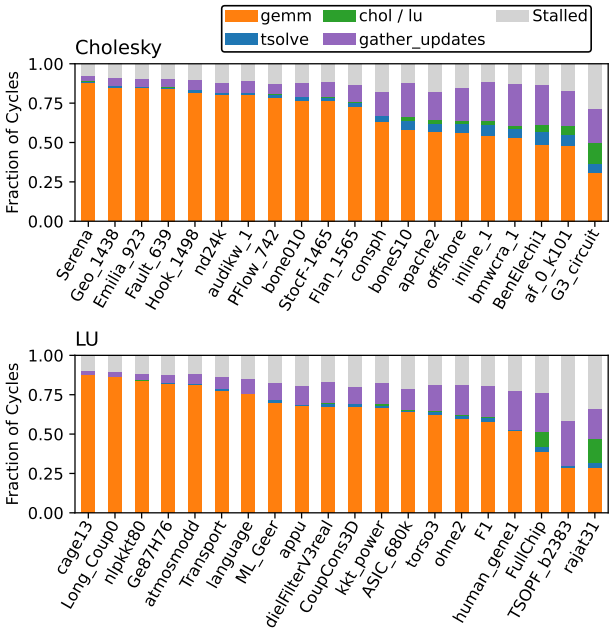


**Figure 16: Spatula cycle breakdown.**

## 7.2 Performance

Table 3 and Table 4 compare Spatula's performance to the CPU and GPU baselines. Speedups are measured in terms of end-to-end execution time, and Spatula's performance is given in TFLOP/s, which also conveys overall utilization (Spatula has a peak throughput of 16.384 TFLOP/s).

Overall, Spatula achieves large speedups: on Cholesky, it is gmean 61.3×/213.7× faster than the GPU/CPU; on LU, it is gmean 36.2×/32.5× faster. Spatula's gmean speedups over Cholesky and LU are 47.1× over the GPU and 83.1× over the CPU. Spatula's large speedups stem from its uniformly high utilization, gmean 10.7 TFLOP/s, 65% of peak throughput.

Performance for Spatula and the baselines depends on matrix structure. In matrices dominated by large supernodes, such as `atmosmodd`, all systems achieve high utilization, and Spatula's speedups over the GPU and CPU are more muted (e.g., 7.7× over the GPU on `atmosmodd`). But in matrices with a more diverse mix of supernodes, GPUs and CPUs falter, whereas Spatula still achieves good utilization. This causes large speedups. For example, Spatula is 22,000× faster than the GPU and 1,600× faster than the CPU on `FullChip`, because Spatula achieves 6.5 TFLOP/s on this matrix while the baselines have terrible utilization (Section 3).

## 7.3 Architectural analysis

**Utilization:** Spatula achieves high PE utilization across matrices. Figure 16 shows a breakdown of activity in PEs, showing the cycles that they spend on tasks of each type and stalled (due to limited parallelism or memory access stalls). Stalls are rare (typically 5-15% of cycles), showing that Spatula's latency-hiding mechanisms and memory-aware scheduling are effective. As discussed in Section 4.3, most cycles are spent in `dgemm` tasks, where Spatula achieves full utilization of ALUs. `gather_update` tasks are the second most-common type, and the other types are rare overall, but can consume significant cycles in some matrices (like `G3_circuit` or `rajat31`).

**Data movement:** Figure 17 reports Spatula's main-memory traffic per matrix. The left of each bar shows the total data transferred (bottom) and average memory bandwidth (top); each bar is broken down by type of traffic. Loads are broken down into three categories: **compulsory loads** needed to read inputs, **noncompulsory loads** initiated by `gather_update` tasks, and **noncompulsory loads** initiated by other task types. Store traffic is split between writing **results** back to main memory and **spilling** intermediates.

Figure 17 shows that Spatula achieves high bandwidth utilization: the average is 40% of the maximum 1 TB/s, with remarkably little deviation across matrices (27% to 87%) given their diversity. This is because Spatula's memory-aware scheduling achieves similar operational intensity across matrices.

Figure 17 also shows that Spatula avoids needless data movement. This can be seen from the ratio of non-compulsory loads to store spills, which is close to 1:1 across all matrices. This means that each spilled value is read back only once most of the time, avoiding thrashing.

**Power consumption:** Figure 18 shows a breakdown of Spatula's power consumption. On almost all matrices, more than half of total power (including main memory) goes to PEs, showing that thanks to Spatula's memory-aware scheduling, these algorithms are compute-bound.

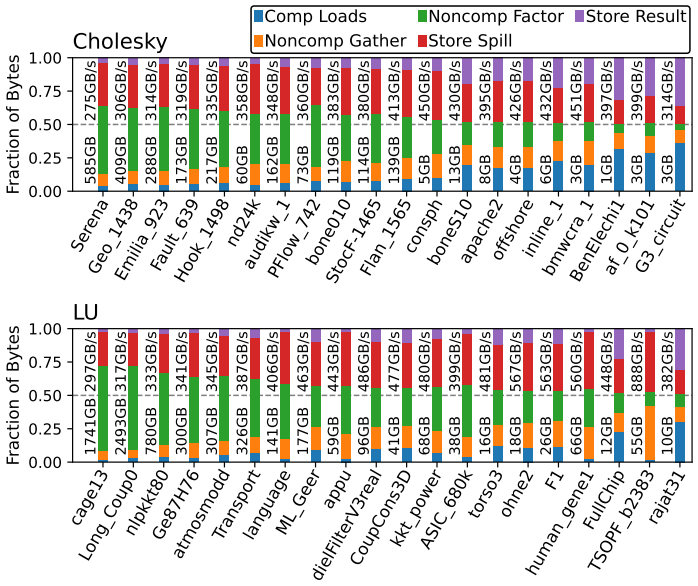


**Figure 17: Spatula data movement.**

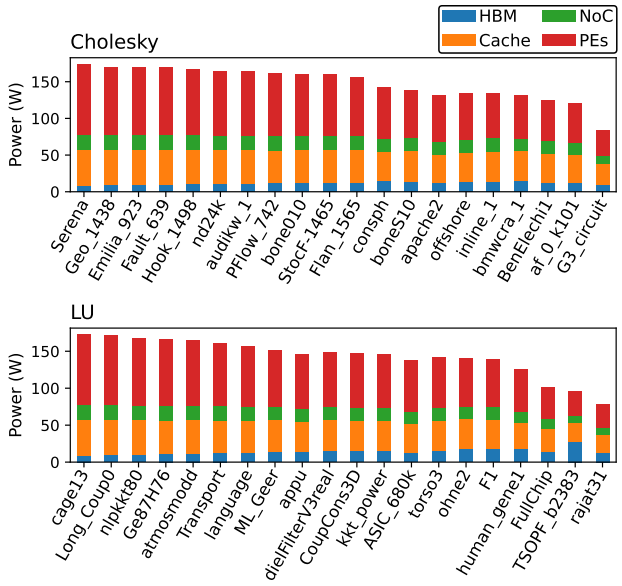


**Figure 18: Spatula power breakdown.**

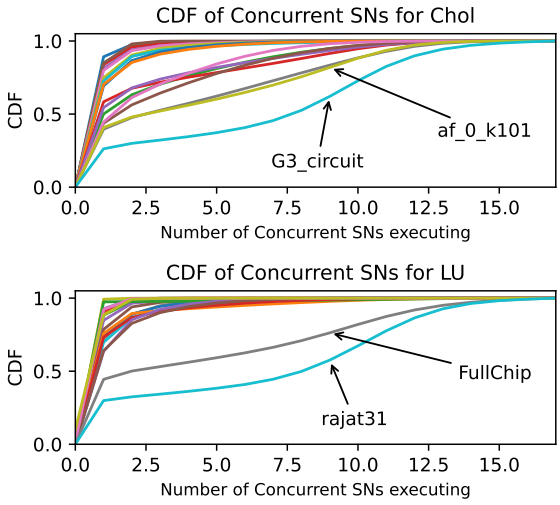


**Figure 19: CDFs of concurrently executing supernodes.**

**Scheduling:** Figure 19 shows CDFs for the distributions of concurrently executing supernodes for Cholesky (top) and LU (bottom). Each line reports results for a single matrix, and each point shows the fraction of time ($y$-axis) that Spatula spends executing at most the given number of supernodes ($x$-axis) concurrently.

Figure 19 shows two important points. First, different matrices need different levels of concurrency, requiring a flexible scheduler. Second, for many matrices, a lot of the time is spent processing a single large supernode. This is because unlike GPU implementations, Spatula can factor small supernodes at high throughput, removing them as bottlenecks.

**Scalability:** We explore the design space of Spatula implementations by sweeping the number of PEs, the primitive tile size, the number of HBM PHYs, and the cache size.

Figure 20 shows the performance of each design ($y$-axis) as a function of area ($x$-axis). Spatula's design scales gracefully to both larger and smaller configurations, with linear performance along the Pareto frontier.

**Comparison across GPU generations:** We have compared with the V100 GPU because it is built on the same technology we synthesize Spatula on, but more recent GPUs exist.

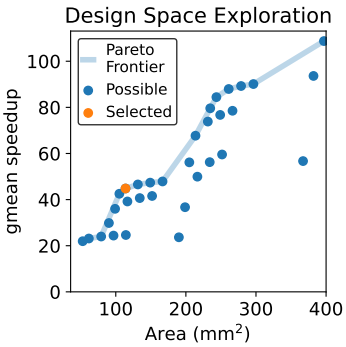


**Figure 20: Scalability**

Table 7.3 shows the gmean performance of STRUMPACK on the NVIDIA V100, A100, and H100 GPUs, and their utilization as percentage of peak throughput (7/19.5/51 FP64 TFLOP/s, respectively). Newer GPUs improve throughput but utilization is low across the board: the A100 improves V100's utilization, reaching 4.8% (likely due to its larger cache and FP64 tensor cores); meanwhile, the H100 barely outperforms the A100 and suffers the lowest utilization, 2.0%.

These results show that the latest GPUs still suffer from poor utilization, and Spatula still outperforms them by a wide margin. Moreover, if built on newer technologies, Spatula can also be scaled to provide higher throughputs. For example, Spatula as configured achieves a 11× speedup over the A100 (just on LU), but the A100 has 2.6× more transistors than the V100. As Figure 20 shows, Spatula scales effectively with area.

| | V100 PCIe | A100 PCIe | H100 PCIe |
|---|---|---|---|
| gmean GFLOP/s | 272 | 962 | 1024 |
| gmean util % | 3.9% | 4.8% | 2.0% |

**Table 5: Performance of STRUMPACK on other GPUs.**

## 8 ADDITIONAL RELATED WORK

Section 3 described prior work that seeks to accelerate matrix factorization. In this section, we discuss sparse linear algebra accelerators designed for other workloads.

**Accelerators for memory-intensive sparse kernels:** Many accelerators target specific vector, matrix, or tensor products, such as sparse matrix-sparse matrix multiplication (SpMSpM) or sparse-matrix vector multiplication (SpMV) [3, 38, 45, 49, 50, 53, 55, 66, 68, 69]. These computations differ greatly from matrix factorization: they have low arithmetic intensity, and require significant work to traverse and manipulate (e.g., intersect or merge) sparse data. These accelerators focus on these aspects, which are not problems for sparse factorization.

**Sparse systolic arrays:** Some accelerators extend systolic arrays to perform sparse matrix multiplication [28, 63]. The systems are orthogonal to Spatula's PEs, which mulitiply *dense* matrices, but extend systolic arrays to other kernels.

**Sparse-dense accelerators:** ExTensor [29] and Tensaurus [57] are programmable accelerators that support a wide range of tensor products, including kernels with one sparse and one dense input. These operations often have higher compute intensity, but tensor products lack the challenging data dependences that arise in sparse matrix factorization. These accelerators are not designed to handle these dependences and have a substantially different structure from Spatula.

**CPU-CGRA systems:** Systems like Tartan [44], DySER [25], BERET [26], and C-Cores [62] integrate spatial reconfigurable arrays into general-purpose cores. In principle, some of these arrays could be configured to execute Spatula tasks. However, the general-purpose core adds large area costs and limits task dispatch throughput. In Section 3.2 and Figure 5, we show that existing CPU implementations already have low utilization. Giving CPUs DySER-style execution units would not fix the dispatch problems, leaving these extremely underutilized.

## 9 CONCLUSION

Sparse matrix factorization dominates performance in many scientific computing applications. Existing architectures are ill-equipped to handle the challenging data dependences and load imbalance induced by different sparse matrix structures. We have presented Spatula, an architecture designed to accelerate sparse LU and Cholesky factorization, achieving gmean 47× speedup over state-of-the-art GPU implementations. By working across the hardware-software interface, Spatula's efficiency gains make a large class of numeric algorithms practical.

Spatula relies on several novel techniques that are applicable beyond our specific implementation. Spatula avoids the parallelism, load imbalance, and data movement bottlenecks of GPUs by *(1)* using short tasks that process small tiles as the basic unit of work; *(2)* adopting flexible scheduling hardware that handles data dependences and dispatches these short tasks at high throughput; and *(3)* leveraging a novel scheduling algorithm that balances parallelism and memory footprint. GPUs or other domain-specific accelerators could also adopt these techniques to achieve high throughput on sparse factorization as well as on applications with similar features.

## ACKNOWLEDGMENTS

We thank Yifan Yang, Quan Nguyen, Victor Ying, Hyun Ryong Lee, Nithya Attaluri, Shabnam Sheikha, Fares Elsabbagh, Alex Krastev, Joel Emer, our anonymous reviewers, and our anonymous shepherd for their feedback on the paper. We would also like to thank Theo Diamandis, Shiv Sundram, and Peter Feldmann for their help in understanding factorization algorithms and motivating the work as well as Mark Hamilton for his help with benchmarking. This research was funded in part by the National Science Foundation under grant CCF-2217099, and by a Wistron research grant.

## REFERENCES

[1] Ahmad Abdelfattah, Pieter Ghysels, Wajih Boukaram, Stanimire Tomov, Xiaoye Sherry Li, and Jack Dongarra. 2022. Addressing irregular patterns of matrix computations on GPUs and their impact on applications powered by sparse direct solvers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[2] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. MUMPS: a general purpose distributed memory sparse solver. In *Applied Parallel Computing*.

[3] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. 2020. Alrescha: A lightweight reconfigurable sparse-computation accelerator. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*.

[4] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report NAS-95-020. NASA Ames Research Center.

[5] Shane Barratt and Stephen Boyd. 2022. Covariance prediction via convex optimization. *Optimization and Engineering* (2022).

[6] Richard P Brent and Franklin T Luk. 1982. *Computing the Cholesky factorization using a systolic architecture.* Technical Report 82-521. Cornell University.

[7] Xiaoming Chen, Yu Wang, and Huazhong Yang. 2013. NICSLU: An adaptive sparse matrix solver for parallel circuit simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)* 32, 2 (2013).

[8] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008).

[9] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018).

[10] Vidushi Dadu and Tony Nowatzki. 2022. TaskStream: Accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*.

[11] Sal Dasgupta, Teja Singh, Ashish Jain, Samuel Naffziger, Deepesh John, Chetan Bisht, and Pradeep Jayaraman. 2020. Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

[12] Sal Dasgupta, Teja Singh, Ashish Jain, Samuel Naffziger, Deepesh John, Chetan Bisht, and Pradeep Jayaraman. 2020. Radeon RX 5700 Series: The AMD 7nm Energy-Efficient High-Performance GPUs. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

[13] Timothy A Davis. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)* 30, 2 (2004).

[14] Timothy A Davis. 2006. *Direct methods for sparse linear systems.* SIAM.

[15] Timothy A Davis. 2013. Algorithm 930: FACTORIZE: An object-oriented linear system solver for MATLAB. *ACM Transactions on Mathematical Software (TOMS)* 39, 4 (2013).

[16] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011).

[17] Timothy A Davis and E Palamadai Natarajan. 2011. Sparse matrix methods for circuit simulation problems. In *Scientific Computing in Electrical Engineering (SCEE 2010)*.

[18] Timothy A Davis and Ekanathan Palamadai Natarajan. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)* 37, 3 (2010).

[19] Iain S Duff and John K Reid. 1983. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)* 9, 3 (1983).

[20] Wei Ge, Mengnan Zhao, Cheng Wu, and Jun He. 2011. The Design and Implementation of DDR PHY Static Low-Power Optimization Strategies. In *Communication Systems and Information Technology*.

[21] Thomas George, Vaibhav Saxena, Anshul Gupta, Amik Singh, and Anamitra R Choudhury. 2011. Multifrontal factorization of sparse SPD matrices on GPUs. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

[22] Pieter Ghysels and Ryan Synk. 2022. High performance sparse multifrontal solvers on modern GPUs. *Parallel Comput.* (2022).

[23] John R Gilbert and Tim Peierls. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.* 9, 5 (1988).

[24] Gene H Golub and Charles F Van Loan. 2013. *Matrix computations*. JHU press.

[25] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of the 17th IEEE international symposium on High Performance Computer Architecture (HPCA-17)*.

[26] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th annual IEEE/ACM international symposium on Microarchitecture (MICRO-44)*.

[27] Azzam Haidar, Ahmad Abdelfatah, Stanimire Tomov, and Jack Dongarra. 2017. High-performance Cholesky factorization for GPU-only execution. In *Proceedings of the General Purpose GPUs (GPGPU-10)*.

[28] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. 2020. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS)*.

[29] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*.

[30] Nachiket Kapre and André DeHon. 2009. Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*.

[31] Kshitij Khare, Sang-Yun Oh, Syed Rahman, and Bala Rajaratnam. 2019. A scalable sparse Cholesky based approach for learning high-dimensional covariance matrices in ordered data. *Machine Learning* 108, 12 (2019).

[32] Seid Koric and Anshul Gupta. 2016. Sparse matrix factorization in the implicit finite element method on petascale architecture. *Computer Methods in Applied Mechanics and Engineering* (2016).

[33] Seid Koric, Qiyue Lu, and Erman Guleryuz. 2014. Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes. *Computers & Structures* (2014).

[34] Hsiang Tsung Kung and Charles E Leiserson. 1979. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*.

[35] Jean-Yves L'Excellent. 2012. *Multifrontal methods: parallelism, memory usage and numerical aspects*. Ph. D. Dissertation. Ecole Normale Supérieure de Lyon.

[36] Xiaoye S Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005).

[37] Xiaoye S Li and James Demmel. 1999. A Scalable Sparse Direct Solver Using Static Pivoting.. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*.

[38] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*.

[39] Joseph WH Liu. 1992. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev.* (1992).

[40] Robert F Lucas, Gene Wagenbreth, John J Tran, and Dan M Davis. 2012. *Multifrontal sparse matrix factorization on graphics processing units*. Technical Report ISI-TR-677. USC Information Sciences Institute.

[41] Carl J. Mauer, Mark D. Hill, and David A. Wood. 2002. Full-system timing-first simulation. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

[42] Paul Messina. 2017. The Exascale Computing Project. *Computing in Science & Engineering* (2017).

[43] Micron. 2018. High Bandwidth Memory with ECC. `https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/hbm2e/8gb_and_16gb_hbm2e_dram.pdf`.

[44] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*.

[45] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*.

[46] Tarek Nechma and Mark Zwolinski. 2014. Parallel sparse matrix solution for circuit simulation on FPGAs. *IEEE Trans. Comput.* (2014).

[47] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[48] NVIDIA. 2020. NVIDIA DGX Station A100 System Architecture. `https://images.nvidia.com/aem-dam/Solutions/Data-Center/nvidia-dgx-station-a100-system-architecture-white-paper.pdf`.

[49] Subhankar Pal, Aporva Amarnath, Siying Feng, Michael O'Boyle, Ronald Dreslinski, and Christophe Dubach. 2021. SparseAdapt: Runtime control for sparse linear algebra on a reconfigurable accelerator. In *Proceedings of the 54th annual IEEE/ACM international symposium on Microarchitecture (MICRO-54)*.

[50] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*.

[51] Giorgos Passas, Manolis Katevenis, and Dionisios Pnevmatikatos. 2012. Crossbar NoCs are scalable beyond 100 nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)* (2012).

[52] Rambus Inc. 2020. White paper: HBM2E and GDDR6: Memory Solutions for AI.

[53] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: Stream ISA and processor specialization for sparse computation. In *Proceedings of the 27th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*.

[54] Steven C Rennich, Darko Stosic, and Timothy A Davis. 2016. Accelerating sparse Cholesky factorization on GPUs. *Parallel Comput.* (2016).

[55] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. 2019. Efficient SpMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*.

[56] Robert Schreiber. 1982. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software (TOMS)* 8, 3 (1982).

[57] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*.

[58] Matthew L Staten, Steven J Owen, Suzanne M Shontz, Andrew G Salinger, and Todd S Coffey. 2012. A comparison of mesh morphing methods for 3D shape optimization. In *Proceedings of the 20th International Meshing Roundtable*.

[59] Yuzhi Sun, ZJ Wang, and Yen Liu. 2007. Efficient implicit non-linear LU-SGS approach for viscous flow computation using high-order spectral difference method. In *Proceedings of the 18th AIAA Computational Fluid Dynamics Conference*.

[60] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2020. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture* (2020).

[61] James E Thornton. 1964. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*.

[62] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*.

[63] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side sparse tensor core. In *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48)*.

[64] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. *Advances in neural information processing systems (NeurIPS)* (2016).

[65] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*.

[66] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. In *Proceedings of the 27th IEEE international symposium on High Performance Computer Architecture (HPCA-27)*.

[67] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2023. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *Proceedings of the 29th IEEE international symposium on High Performance Computer Architecture (HPCA-29)*.

[68] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*.

[69] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient architecture for sparse matrix multiplication. In *Proceedings of the 26th IEEE international symposium on High Performance Computer Architecture (HPCA-26)*.