Accelerating Sparse Cholesky Factorization on Sunway Manycore Architecture

Mingzhen Li, Yi Liu, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang and Depei Qian

Abstract—To improve the performance of Sparse Cholesky factorization, existing research divides the adjacent columns of the sparse matrix with the same nonzero patterns into supernodes for parallelization. However, due to the various structures of sparse matrices, the computation of the generated supernodes varies significantly, and thus hard to optimize when computed by dense matrix kernels. Therefore, how to efficiently map sparse Choleksy factorization to the emerging architectures, such as Sunway many-core processor, remains an active research direction. In this paper, we propose *swCholesky*, which is a highly optimized implementation of sparse Cholesky factorization on Sunway processor. Specifically, we design three kernel task queues and a dense matrix library to dynamically adapt to the kernel characteristics and architecture features. In addition, we propose an auto-tuning mechanism to search for the optimal settings of the important parameters in *swCholesky*. Our experiments show that *swCholesky* achieves better performance than state-of-the-art implementations.

Index Terms—Sunway architecture, Sparse Cholesky factorization, Performance Optimization

1 Introduction

Sparse Cholesky factorization is an important and indispensable building block of modern scientific applications, which is widely used in the fields such as computational fluid dynamics [1], finite element analysis [2], geophysics [3] and electromagnetism [4]. Applications in these fields usually rely on large numerical models, which requires solving the sparse linear systems such as $Ax = b$. Suppose A is a sparse symmetric positive definite (SPD) matrix, x and b are dense vectors, methods that solve the above sparse linear system can be classified into direct methods [5] and iterative methods [6]. In direct methods, Cholesky factorization is the first step, which factorizes the sparse SPD matrix A into the form of $A = LL^T$, where L is a sparse lower triangular matrix, and the elements on its diagonal are positive. To derive the solution of dense vector x, one needs to solve two linear systems of $Ly = b$ and $L^T x = y$. Cholesky factorization is the core computation in the above methods, and it dominates the execution time. In iterative methods, such as pre-conditioned Conjugate Gradient (PCG), an approximate form of Cholesky factorization, named incomplete Cholesky factorization, is commonly used as a preconditioner to reduce the number of iterations.

To accelerate sparse Cholesky factorization, George and Liu [7] first proposed the concept of *supernode*, which enables the acceleration of Cholesky factorization using high-performance dense linear algebra libraries. In the process of solving column j of matrix L, the sparse pattern of column j can affect all columns corresponding to the nonzero elements in column j. Thus, there are usually columns with similar sparse patterns. By appropriate reordering, the number of columns with similar sparse patterns can be increased. These columns are collectively referred as *supernode*, and the basic computation unit of Cholesky factorization changes from one column to one *supernode* (several columns). In addition, the computations for Cholesky factorization are converted into dense matrix operations. In supernodal left-looking Cholesky algorithm, when calculating *supernode* i, the corresponding columns in *supernode* i $(j < i)$ are packaged in the form of dense matrix and then added to i after several transformations, then *supernode* i itself is solved in the dense form. In this process, four linear algebra kernels, *syrk*, *gemm*, *trsm* and *potrf* are mainly used.

Sunway TaihuLight is the first supercomputer to achieve more than 100PFlops peak performance around the world. It is equipped with 40,960 pieces of Sunway SW26010 many-core processors. Each Sunway processor has four core groups (CG). The performance in double-precision is 3.06TFlops. Each CG consists of a Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs). Both MPE and CPEs are 64-bit RISC cores. The MPE is used for computation management, whereas the CPEs are used for computation acceleration. In addition, each CPE has 64KB Local Device Memory (LDM), whose bandwidth and latency is similar to L1 cache, but requires explicit control by the program. The CPEs can access the main memory through Direct Memory Access (DMA) for continuous accesses. In advance, the CPEs also support DMA broadcast (BCAST) mode. In BCAST mode, the data accessed by one CPE through DMA is broadcasted to the rest CPEs within a CG, which achieves higher memory bandwidth on CPE.

In order to efficiently run sparse Cholesky factorization on Sunway many-core architecture, we re-design the supernodal Cholesky factorization by taking advantage of the architecture features of Sunway. Specifically, we need to

- *M. Li, Y. Liu, H. Yang, Z. Luan and D. Qian are with Sino-German Joint Software Institute, the School of Computer Science and Engineering, Beihang University, Beijing, China, 100191. L. Gan and G. Yang are with Department of Computer Science and Technology, Tsinghua University, Beijing, China, 100084.*
  Email: {lmzhhh, yi.liu, hailong.yang, 07680, depeiq}@buaa.edu.cn and {lingan, ygw}@tsinghua.edu.cn

address the following challenges:

- *Inefficient computation of small-scale kernels*: The width of each *supernode* is strongly correlated to the sparse structure of matrix A, therefore existing approach may generate a large number of small *supernodes*, which rely on the computation of a large number of linear algebra kernels with small input size (denoted by small-scale kernels). However, these kernels are too small to fully utilize the massive computing resources as well as the precious memory bandwidth of Sunway.
- *Lack of essential math library*: There lacks a flexible and efficient dense linear algebra library that supports sparse Cholesky factorization on Sunway architecture. Currently, the *BLAS* and *LAPACK* libraries only support the computation on MPE, which cannot utilize the computation power of CPEs. Although the *xMath* library supports the kernel acceleration using CPEs, it parallelizes each kernel independently without acknowledging the kernel relationships in Cholesky factorization, and thus fails to exploit the opportunity of performance improvement by increasing the data reuse.
- *Various sparse matrice structures*: The structures of sparse matrices in scientific applications vary significantly. In Cholesky factorization, it requires to apply the dense kernels to sub-matrices from matrices with varying structures. These sub-matrices are different in various aspects such as size, shape and storage format, which makes it difficult to optimize the performance when applying the dense kernels in Cholesky factorization.

To address the above challenges, we design and implement *swCholesky*. We optimize the dense kernels required for sparse Cholesky factorization, and propose the design of multiple task queues to schedule the kernels of different scales. In addition, we propose a performance auto-tuning mechanism to further optimize the performance of *swCholesky*. We compare *swCholesky* with the-state-of-the-art implementations of sparse Cholesky factorization. Our experiment results demonstrate that *swCholesky* can utilize the architecture features of Sunway effectively, and achieve promising performance speedup across representative datasets. Specially, this paper makes the following contributions:

- We propose a kernel scheduling mechanism based on multiple kernel task queues (*KTQs*), including Instant Queue (*InstQ*), Task Parallel Queue (*TPQ*) and Data Parallel Queue (*DPQ*). The *KTQs* schedule the kernels to MPE and CPEs based on the computation scale of each kernel as well as accelerate the kernel on CPEs in either task parallel or data parallel manner.
- We propose *swCHOLBLAS*, a dense linear algebra library for sparse Cholesky factorization, including *syrk*, *gemm*, *trsm* and *potrf* kernels. The *swCHOLBLAS* achieves comparable performance to the existing xMath library on a single CG. More importantly, the *swCHOLBLAS* enables the computation of Cholesky factorization scale to four CGs on one Sunway processor, which provides more performance opportunity to accelerate the computation of sparse Cholesky factorization on Sunway.
- We build a performance model for *swCholesky* on Sunway, and propose a performance auto-tuning mechanism to search for the optimal parameter settings in *swCholesky*. The auto-tuning mechanism eliminates the drawbacks of manual and empirical tuning, which is difficult to achieve optimal performance for sparse matrices with varying structures.

The remaining part of this paper is organized as follows. Section 2 describes the background of Cholesky factorization as well as the existing optimization techniques. Section 3 presents our methodology to re-design sparse Cholesky factorization to adapt to Sunway architecture. Section 4 presents the evaluation results of *swCholesky* by comparing to the-state-of-the-art implementations. Section 5 discusses the related work and section 6 concludes this paper.

2 Background

2.1 Cholesky Factorization

In this subsection, we present the basic concepts about Cholesky factorization, including the mathematical expression, the generation of fill-in, supernodal method, and serial/parallel implementations using left looking and right-looking approaches.

Assuming matrix $A \in \mathbb{R}^{n\times n}$ is a sparse SPD matrix. The goal of sparse Cholesky factorization is to find a sparse lower triangular matrix $L \in \mathbb{R}^{n\times n}$ or sparse upper triangular matrix $U \in \mathbb{R}^{n\times n}$, so that $A = LL^T$ or $A = U^TU$. Because matrix U can be derived by transposing matrix L, we focus on illustrating the process of deriving matrix L, which is known as the Cholesky factor.

Algorithm 1 Sparse Cholesky Factorization.

Input: a: sparse SPD matrix. (Left-looking)
1: **for** $j = 0$ **to** $n - 1$ **do**
2: // pull the updated matrix and update j
3: **for** $k \in Ancestors[j]$ **do**
4: $cmod(j, k)$
5: **end for**
6: $cdiv(j)$
7: **end for**

Input: a: sparse SPD matrix. (Right-looking)
1: **for** $j = 0$ **to** $n - 1$ **do**
2: $cdiv(j)$
3: // push the updated k of the matrix to $Childs$
4: **for** $k \in Childs[j]$ **do**
5: $cmod(k, j)$
6: **end for**
7: **end for**

The computation of Cholesky factorization contains two parts, $cdiv(j)$ and $cmod(j, k)$. The $cdiv(j)$ is responsible for calculating the square root of the diagonal element of column j and dividing other elements by this square root. The $cmod(j, k)$ is responsible for updating column j according to the computation results of column k. The

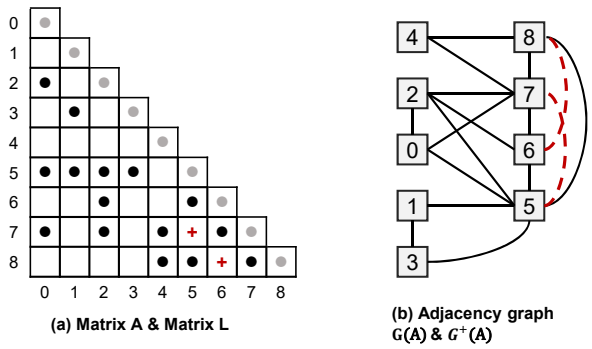


Fig. 1. The illustration of Matrix A, Cholesky factor L (a) and their adjacency graphs (b). • and $+$ refer to nonzeros of A and fill-in of L.

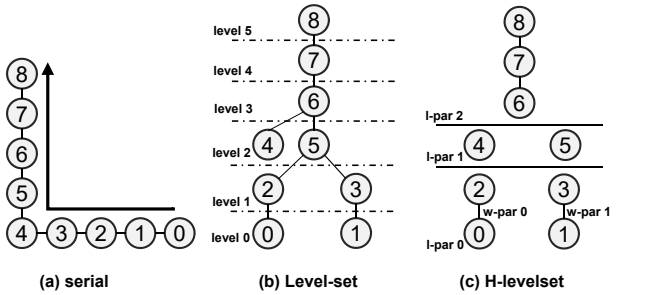


Fig. 2. The serial (a), Level-set (b) and H-levelset (c) methods for sparse Cholesky factorization.

widely used left-looking and right-looking Cholesky factorizations shown in Algorithm 1 have different computation orders of $cdiv$ and $cmod$ when processing the dependencies, where $Ancestors[j]$ represents column indices of all non-zero elements of row j, and $Childs[j]$ represents row indices of all non-zero elements of column j. For instance, when performing the computation on column j, left-looking algorithm pulls the data from dependent columns and performs $cmod(j,.)$, before performing $cdiv(j)$. Whereas, right-looking algorithm first performs $cdiv(j)$, and then pushes the value of column j to columns that depend on column j, before performing $cmod(.,j)$.

Fill-in is inevitably generated during the computation of sparse Cholesky factorization, as shown in Figure 1. Specifically, there are elements in matrix A, where $A(i,j)=0$, but $L(i,j)\neq 0$. To explain this phenomenon, we first introduce some basic concepts in graph theory. Assuming the adjacency graph of $A\in\mathbb{R}^{n\times n}$ is $G(A)=(V,E)$, where $G(A)$ has n vertices, each vertex $v\in V$ corresponds to column v in matrix A (also row v because A is symmetric), each edge $(v,u)\in E$ corresponds to nonzero element $A(v,u)$ (also $A(u,v)$ due to symmetry). We define F as the set of edges representing all *fill-in* in Cholesky factor L. Then the adjacency graph of L can be defined as $G^{+}(A)=(V,E\bigcup F)$. Due to space constraints, we omit the proof and give the conclusion here. Exists edge $(i,j)\in F$ if and only if there is a vertex k $(k<i,j)$ that joins vertex i and vertex j in the original graph $G(A)$. The readers can refer to [8] for detailed proof.

The design of *supernode* [7] consists of a series of columns with the same nonzero patterns, that are stored in dense format. The *supernode* structure transforms many sparse vector operations into dense matrix operations, thus enables the performance speedup by using the computing resources on modern processors. By replacing the column i,j with *supernode* i,j in left-looking or right-looking Cholesky factorization, we can derive the supernodal left-looking or right-looking Cholesky factorization.

To further parallelize sparse Cholesky factorization based on *supernode* structure, existing work such as Level-set [9] and H-levelset [10], derives the dependency between *supernodes* and constructs a directed acyclic graph (DAG) to exploit the parallelism. The Level-set method divides the *supernodes* into several levels, all *supernodes* within one level are independent of each other and can be calculated in parallel. However, synchronizations are performed after the calculation of each set to ensure the correctness. For instance, in Figure 2(b), the 9 *supernodes* are divided into 5 levels. H-levelset is a variant of Level-set, as shown in Figure 2(c), it merges several levels into a l-partition, and then divides l-partiton into several w-partitions using load-balanced level coarsening algorithm (LBC). The w-partitions in one l-partition are independent of each other. Although H-levelset is similar to Level-set, it has better *supernode* partition, which reduces the number of synchronizations and improves the load balance when the sparse matrix A is large enough. Our work is based on the H-levelset method with in-depth optimizations for Sunway architecture.

2.2 Existing Techniques on Optimizing Cholesky Factorization

The computation of sparse Cholesky factorization is composed of large amount of dense kernels. These kernels have diverse scales, different computation characteristics and varying memory access patterns. Therefore, it is necessary to schedule these dense kernels efficiently. Currently, existing work on kernel scheduling of sparse Cholesky factorization is mainly focusing on the CPU+GPU hybrid platform, which can be classified into three categories: *1)* assigning certain kinds of kernels always to CPU or GPU, disregarding the kernel scale. A representative work in this category is [11]. It adopts auto-tuning mechanism to select the most time-consuming kernels to offload to GPU. *2)* identifying large-scale kernels to offload to GPU, and leaving small-scale kernels to CPU. A representative work in this category is [12]. It sorts the kernels based on their input scale, and assigns the kernels to GPU from the largest to the smallest scale and simultaneously to CPU but in the opposite direction. *3)* assigning batched small-scale kernels to GPU. The representative work is [13]. The GPU performs several small dense Cholesky factorizations with fixed scale simultaneously.

However, naively adopting the above scheduling strategies on Sunway architecture could lead to severe performance degradation. Firstly, we can only assign small-scale kernels to MPE, because the peak performance of MPE is much lower than CPU. As the kernel scale increases, the MPE can easily become the performance bottleneck. Secondly, although assigning the kernels with medium scale to CPEs can achieve higher performance than MPE, it cannot fully utilize the computing resources of CPEs. Moreover,

there are different ways to run the kernel on CPEs, either in task parallel manner or in data parallel manner. Therefore, we need to design different queues for kernels with varying scales on Sunway, and dynamically assign the kernels to the corresponding queues during runtime. In addition, we need to determine the appropriate parallelization manner when running kernels on CPEs.

In addition to the kernel scheduling, the computation of sparse Cholesky factorization relies on four dense kernels including *syrk*, *gemm*,*trsm* and *potrf*, which also worths the attention for optimization. However, existing work rarely deals with optimization at kernel level due to the popularity of the dense linear algebra (DLA) libraries, such as *BLAS*, *MKL* and *CUBLAS*. Although there are several DLA libraries on Sunway, such as *BLAS/LAPACK*, *xMath* and *swGEMM*, the limitations of these libraries prevent their adoption in our implementation of sparse Cholesky factorization on Sunway. For instance, *BLAS* and *LAPACK* only support to sequential execution on MPE, which leads to poor performance. The *xMath* supports parallel execution on CPEs, it leads to poor efficiency when computing small-scale kernels. The *swGEMM* is designed for accelerating convolution operation in *swDNN*, it can only process matrices of specific scale and storage format. Therefore, we argue a dense linear algebra library tailored for sparse Cholesky factorization on Sunway is required in order to achieve good performance on Sunway by leveraging the unique architectural features. The proposed dense linear algebra library provides customized optimizations for Cholesky factorization, but can also be used by other applications in general. Moreover, we collaboratively optimize the design of dense linear algebra library and multiple kernel task queues, which enables processing sparse matrices with varying structures on Sunway efficiently.

3 Re-designing and Optimizing Sparse Cholesky Factorization on Sunway

In this section, we focus on describing the design and implementation of *swCholesky*, as well as optimization techniques to accelerate *swCholesky* on Sunway.

3.1 Design Overview

The design of *swCholesky* primarily contains two parts: the symbolic part and the numeric part, as shown in Figure 3.

In the symbolic part, it analyzes the input matrix A to determine the parameters critical to overall performance, and then generates corresponding *supernodes* based on H-levelset. The detailed process is as follows. Firstly, the SPD matrix A is analyzed, and then based on the sparse structure of A as well as the performance model of sparse Cholesky factorization, the auto-tuning mechanism is carried out to determine the settings of several performance-critical parameters. Then the *supernodes* are generated based on the auto-tuned parameters. According to the dependency, the *supernodes* are divided into several l-partitions, and each l-partition is further divided into several independent w-partitions.

In the numeric part, it performs the numeric calculations of sparse Cholesky factorization based on H-levelset and corresponding parameter settings, derived from the symbolic part. We evenly assign the w-partitions within each l-partition to 4 CGs of one Sunway processor, thus the 4 w-partitions are calculated by the 4 CGs in parallel. On each CG, the *supernodes* of each w-partition are calculated in sequential. And the MPE is responsible for scheduling all computation tasks. The entire computation of Cholesky factorization is performed by MPE and CPEs collaboratively.

For the computation kernels applied to *supernodes*, we design multiple kernel task queues (*KTQs*) and a dense linear algebra library *swCHOLBLAS*. The *KTQs* are responsible for scheduling the kernels with different scales, and determine the execution mode of the kernels on CPEs. The *swCHOLBLAS* is responsible for optimizing the dense kernels including *syrk*,*gemm*, *trsm* and *potrf* targeting the Sunway architecture. The kernels in *swCHOLBLAS* are further classified into basic kernels and combined kernels. The basic kernels consist of small-scale *trsm* and full-scale *gemm*, whereas the combined kernels consist of full-scale *syrk*, *trsm* and *potrf*. In addition, the combined kernels can be split into several basic kernels based on the parameter settings derived from auto-tuning.

Through the above design, the proposed *swCholesky* realizes a series of optimizations including multiple kernel task queues (Section 3.2) and dense linear algebra library (Section 3.3) by leveraging the architecture advantage of Sunway processor. Note that, although this paper is targeting the Sunway architecture, the optimizations proposed in this paper such as kernel task queues and dense linear algebra library are also applicable to other many-core cacheless architectures that share similar design to Sunway.

3.2 Kernel Task Queues

As shown in Figure 3, when calculating the *supernode* s, one first needs to pull the corresponding matrices $M1_i$ and $M2_i$ from *supernode* i that s depends on. The pseudocode is also shown in Algorithm 2 (lines 3 - 13). The dense matrix S of *supernode* s contains two dense parts: $S1$ and $S2$, on which the dense matrix operations are performed: $S1 -= syrk(M1_i)$ and $S2 -= gemm(M2_i, M1_i^T)$. Since *syrk* and *gemm* update different parts of dense matrix S, these two operations are independent from each other. At the same time, $S1$ and $S2$ are both updated by *subtraction* operation, therefore the final results of both $S1$ and $S2$ are independent from the order how the *syrk* and *gemm* operations are applied. This means, in the process of updating dense matrix S, all kernel operations are commutative. Based on the above observation, we design three kernel task queues to schedule the kernels with different scales.

When the size of input matrix is small, it is ineffective to assign the *syrk* and *gemm* kernels to CPEs. This is because MPE can finish the kernel computation during the time it initiates the computation on CPEs. Therefore, the small-scale kernels should be calculated on MPE for better efficiency. Based on the above observation, we design the instant queue (*InstQ*) to schedule small-scale kernels to be instantly calculated on MPE as shown in Algorithm 2 (lines 7- 8).

As the size of input matrix increases, it becomes more beneficial to perform the kernel computation on CPEs. However, before the matrix size is large enough, the kernel

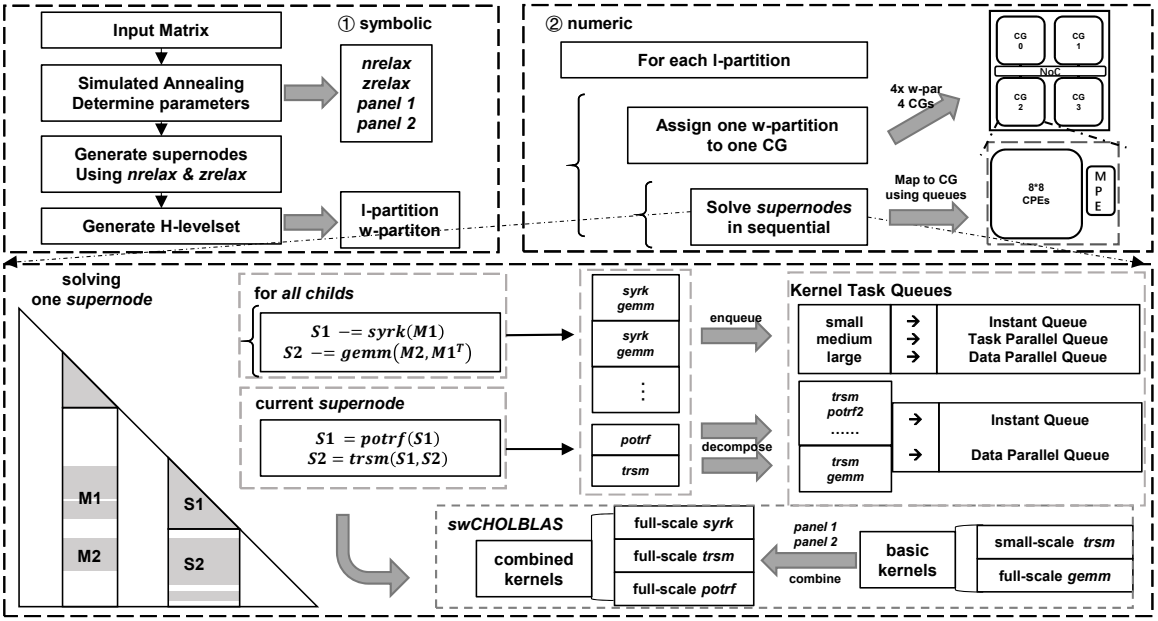


Fig. 3. The overall design of *swCholesky*, including the symbolic and numeric parts, the detailed calculation inside one *supernode*, and the brief display of *swCHOLBLAS* and Kernel Task Queues.

Algorithm 2 Kernel Task Queues.

Input: *supernodes* and their dependency relationship

1: **for** *supernode* s in w-partition **do**
2: Find child list $[C_1, C_2, ..., C_n]$
3: **for** C_i in child list **do**
4: Find dense part $M1_i, M2_i$, which affect S
5: // *syrk* kernel $S1 -= M1_i \cdot M1_i^T$
6: // *gemm* kernel $S2 -= M2_i \cdot M1_i^T$
7: $addTask(Queue, syrk, M1_i)$
8: $addTask(Queue, gemm, M2_i, M1_i)$
9: **if** $isQueueFull \parallel isLastChild$ **then**
10: $fusionQueue(Queue, syrk, gemm)$
11: $execQueue(Queue)$
12: **end if**
13: **end for**
14: *potrf* kernel $S1 = L1 \cdot L1^T, S1 \leftarrow L1$
15: *trsm* kernel $S2 = L2 \cdot L1^T, S2 \leftarrow L2$
16: **end for**

computation cannot fully utilize the resources of the 64 CPEs such as the LDM and DMA bandwidth. Since the 64 CPEs are independent of each other, they are suitable for computation in task parallel. Therefore, for the medium-scale kernels, we design the task parallel queue (*TPQ*). As shown in Algorithm 2 (lines 9 to 12), when the *TPQ* is full or there is no remaining task to process, the tasks in *TPQ* are assigned to 64 CPEs evenly. Moreover, due to the limited size of LDM, we fuse the computation of *syrk* and *gemm*. Both the *gemm* and *syrk* kernel relies on $M1_i$ as its input, as shown in Algorithm 2 (line 5 and line 6). By fusing the *syrk* and *gemm* kernels, the dense matrix $M1_i$ can be re-used, which helps to reduce the memory footprint and save the LDM space.

When the size of input matrix is large enough, the LDM on a single CPE cannot accommodate the entire matrix. For the kernels compute on such large matrices, we use the LDMs on the 64 CPEs as a whole to improve the data locality. The data stored on one CPE can be shared with other CPEs through register communication. For the large-scale kernels, we design the data parallel queue (*DPQ*) to schedule the kernels on the 64 CPEs that are accelerated by the way of data parallelism.

With *KTQs*, the kernels with different scales are computed more efficiently on Sunway with better resource utilization.

3.3 Optimizing Kernels in Cholesky Factorization

In this subsection, we present the design and implementation of *swCHOLBLAS*, which is the dense linear algebra library tailored for sparse Cholesky factorization on Sunway. Firstly, The basic kernels used in Cholesky factorization such as small-scale *trsm* and full-scale *gemm* are optimized significantly. Then, the combined kernels such as full-scale *trsm* and *potrf* can be decomposed into the combination of the above basic kernels. In *swCHOLBLAS*, we implement all the kernels in sequential on MPE as well as in parallel on CPEs. During runtime, the *swCHOLBLAS* automatically selects the appropriate implementation based on the input size of the kernel.

3.3.1 Basic Kernel Optimization

We first introduce the special case of ***trsm*** kernel, which is also known as *trsv* kernel. We illustrate the *trsv* kernel with dense lower triangular matrix $a \in \mathbb{R}^{n \times n}$ and dense vector $b \in \mathbb{R}^{m \times n} (m = 1)$. As shown in Algorithm 3 (lines 2 - 9), it calculates the element $b[0, k]$ and then updates the subsequent elements $b[0, j]$ $(j > k)$ according to $b[0, k]$. The strong data dependency prevents the parallel implementation of *trsv*.

The inputs of *trsm* kernel are dense lower triangular matrix $a \in \mathbb{R}^{n \times n}$ and dense matrix $b \in \mathbb{R}^{m \times n}$ $(m > 1)$.

Algorithm 3 Dense *trsm* kernel (when $m = 1$, *trsv* kernel).

Input: a: dense lower triangular matrix of column major
Input: b: multiple right-hand vectors (when $m = 1$, a dense vector)

1: **for** $k = 0$ **to** $n - 1$ **do**
2: **for** $i = 0$ **to** $m - 1$ **do**
3: $b[k \times ldb + i] \leftarrow b[k \times ldb + i]/a[k \times lda + k]$
4: **end for**
5: **for** $j = k + 1$ **to** $n - 1$ **do**
6: **for** $i = 0$ **to** $m - 1$ **do**
7: $b[j \times ldb + i] \leftarrow b[j \times ldb + i] - a[k \times lda + j] \times b[k \times ldb + i]$
8: **end for**
9: **end for**
10: **end for**

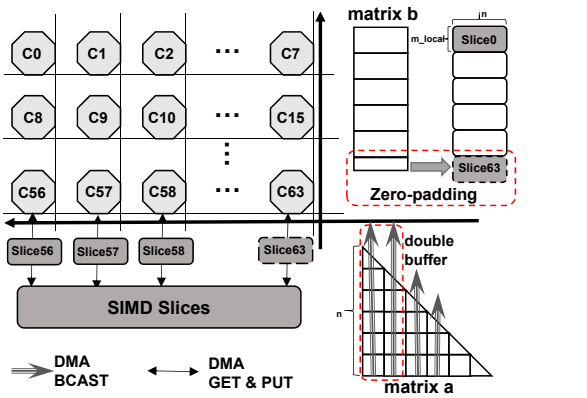


Fig. 4. The design and implementation of basic *trsm* kernel on Sunway.

Compared to *trsv* with single right-hand side ($m = 1$), *trsm* has multiple right-hand sides ($m > 1$), therefore it has better data-level parallelism. In other words, the *trsm* kernel can be seen as applying *trsv* kernel m times to the same input matrix a. Based on this property, we implement the parallel *trsm* kernel by evenly assinging the m *trsv* kernels into the 64 CPEs, so that each CPE processes $m\_local = m/64$ *trsv* kernels. And each CPE stores $m\_local \times n$ elements of matrix b. As shown in Figure 4, each CPE reads/writes a corresponding slice of b through DMA. In each iteration, such as iteration k, only CPE 0 reads $a[k : n, k]$, a column of matrix a through DMA in BCAST mode, then broadcasts $a[k : n, k]$ to other CPEs. Then all CPEs perform the calculation and update their local slices.

In order to overlap the computation and memory access through DMA, we design the double buffer mechanism for the basic *trsm* kernel. While processing the iteration k, we first issue a DMA request of next column $a[k + 1 : n, k + 1]$, and perform the calculation simultaneously. As the calculation completes, the data of the next column $a[k+1 : n, k+1]$ is available for the next iteration. In this way, memory access is overlapped by the computation.

Moreover, we leverage the 256-bit SIMD vector units on CPEs to implement the vectorization of *trsm*. On Sunway, all vector operations require 256-bit data alignment in LDM, which are 4 double-precision floating points. However, we cannot guarantee that the value of m_local is strictly a multiple of 4. Inspired from Caffe [14] that applies zero-padding to the feature maps, we apply zero-padding to matrix b as shown in Figure 3, so that the m dimension of matrix b is 256-double aligned, and thus each SIMD slice is 4-double aligned. Therefore, each CPE performs the calculation on one SIMD slice, which is 4-double aligned in column-major matrix. As shown in Algorithm 3, the scalar operations in line 3 and line 7 on a SIMD slice are converted into vector operations. Note that, we also parallelize the padding and de-padding procedure to reduce their performance overhead to *trsm* kernel.

As a component of *swDNN*, *swGEMM* can only accelerate matrix multiplication of a particular scale and in specific storage format, which cannot be directly applied to sparse Cholesky factorization. Therefore, we propose a new ***gemm*** kernel tailored for sparse Cholesky factorization on Sunway.

We adopt the design of leading dimension to the traditional memory addressing mechanism such as in *swGEMM*, and implement the support of leading dimension in the *gemm* kernel of *swCHOLBLAS*. For instance, the memory addressing of $A(i, j)$, is replaced by $A[j \times lda + i]$. The addressing of matrix B and C is also modified similarly. The batch size and stride size of the DMA transfer are also adjusted to support the design of the leading dimension. The leading dimensions allow matrix A, B and C to be sub-matrices extracted from larger matrices, which enables the computation applied to the matrices at a more flexible granularity. Regarding the parameters (α, β), the *gemm* kernel in *swCHOLBLAS* requires special settings of $(-1, 1)$ and $(1, 0)$. We apply padding to matrix A, B and C. In order to reduce the memory access latency, we apply double buffer mechanism during the DMA transfer. To support the case of $(\alpha, \beta) = (-1, 1)$, we modify the assemble code of *gemm* kernel to replace the vector multiply-add instruction *vmad* with vector multiply-subtract instruction *vnmad*. What's more, we also improve the availability of the original blocking parameter search mechanism, and apply it to the *gemm* kernel of *swCHOLBLAS*.

3.3.2 Combined Kernel Optimization

The ***syrk*** performs the matrix multiplication of the lower triangular $L \in \mathbb{R}^{n \times k}$ and its transposition $L^T \in \mathbb{R}^{k \times n}$ to derive $C \in \mathbb{R}^{n \times n}$, that is, $C = \alpha \times L \cdot L^T + \beta \times C$. Obviously, the calculation of *syrk* kernel is similar to *gemm* kernel. Therefore, the *syrk* can be converted into *gemm*. It worths mentioning that in the original *syrk*, the upper triangular part of matrix C is not modified, however, it is modified if the *syrk* is implemented using *gemm*. Fortunately, the upper triangular part of matrix C is ignored in sparse Cholesky factorization, thus the correctness is not violated after the conversion.

The ***trsm*** kernel can be decomposed into several basic *trsm* and *gemm* kernels. When the scale of input matrices is large, we divide lower triangular matrix $L \in \mathbb{R}^{n \times n}$ into several Lp and divide matrix $B \in \mathbb{R}^{m \times n}$ into several Bp according to the *adjustable* parameter $p1 = panel1$. As shown in Figure 5(a), each Lp consists of a triangular part Lp_tri and a block part Lp_blk. Figure 5(a) presents an example of *trsm* kernel, L and B are both divided into 3 panels. The size of matrix B after padding is $(m1+m2) \times n$, where $m1 \times n$ and $m2 \times n$ indicates the sub-matrix and padding matrix of B respectively. To compute panel 0, we

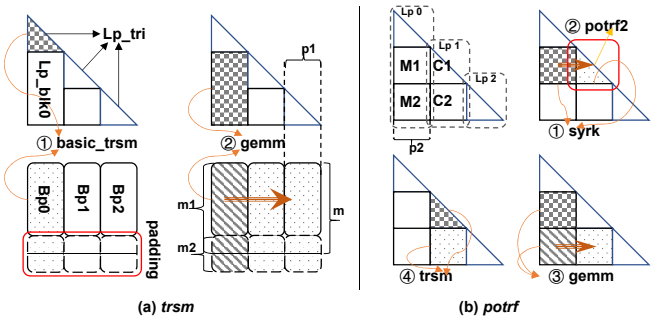


Fig. 5. The computation procedure of the combined *trsm* (a) kernel and combined *potrf* (b) kernels.

TABLE 1
List of important symbols.

Symbol	Description
$v_{inB}(x,y)$	DMA (read) bandwidth (with batch size x, stride size y)
$v_{outB}(x,y)$	DMA (write) bandwidth (with batch size x, stride size y)
v_{bcastB}	DMA bandwidth in BCAST mode
v_{flop}	Number of floating point operations per second on CPE
$p1$, $panel1$	Panel width of combined *trsm* kernels
$p2$, $panel2$	Panel width of combined *potrf* kernels
$nrelax[0:3]$	Maximum width of a merged *supernode*
$zrelax[0:3]$	Maximum proportion of non-zeros of a merged *supernode*
N	Column number of the given sparse SPD matrix L
Nnz	Number of non-zero elements of L
d	Approximate non-zero density of L
λ	Time coefficient for *syrk* and *gemm* kernels
ΔT	Time change before and after the *supernode* merging
F	Target function in simulated annealing

first apply *trsm* kernel to the input matrices Lp_tri0 and $Bp0$, and the result is updated to matrix $Bp0$. Then we apply *gemm* kernel to the input matrices Lp_blk0 and $Bp0$, and the result is updated to matrices $Bp1$ and $Bp2$. The computation of other panels is similar.

The ***potrf*** kernel performs the dense Cholesky factorization. The matrix L is divided into several parts according to the *adjustable* parameter $p2 = panel2$, as shown in Figure 5(b). The computation of each panel involves four kernels including *syrk*, *gemm*, *trsm* and *potrf*2(only MPE version). Take the computation of $Lp1$ in Figure 5(b) for example, $Lp1$ is composed into the triangular matrix $C1$ and matrix $C2$, and the computation of $C1$ and $C2$ relies on the matrix $M1$ and $M2$ on the left. The computation consists of four steps: *1)* a *syrk* kernel, $C1 -= M1 * M1^T$, *2)* a *potrf*2 kernel, $C1 = L1 * L1^T$, and $C1$ is updated by $L1$, *3)* a *gemm* kernel, $C2 -= M2 * M1^T$ and *4)* a *trsm* kernel, $C2 = \hat{C}2 * L1^T$, and $C2$ is updated by $\hat{C}2$.

3.4 Performance Auto-tuning

The settings of the adjustable parameters in *swCholesky* have a significant impact on its performance. In this subsection, we first establish the performance models for the basic kernels, the combined kernels and the entire Cholesky factorization, respectively. Based on these performance models, we propose an auto-tuning mechanism to search for the optimal parameter settings on Sunway using simulated annealing algorithm. Some important symbols and their descriptions are listed in Table 1.

3.4.1 Performance Model for Basic Kernels

The performance model of basic *trsm* kernel is defined in Equation (1a), and that of basic *gemm* kernel is defined in Equation (1b). Their parameters m, n and k indicate the size of input/output matrices, whereas lda,ldb and ldc indicate the leading dimensions. And $blkM$, $blkN$ and $blkK$ are already determined blocking parameters of *gemm* kernel. The default DMA bandwidth (with batch size x and stride size y) is denoted as $v_{inB}(x,y)$ and $v_{outB}(x,y)$ respectively. The DMA bandwidth in BCAST mode is denoted as v_{bcastB}. And the number of real floating point operations per second on CPE is denoted as v_{flop}.

$$\begin{aligned} T_{basic\_trsm}(m,n,ldb) &= T_{inB} + T_{outB} + \max(T_{calc}, T_{inA}) \\ &= \frac{\lceil m/256\rceil \times 256 \times n}{v_{inB}(\lceil m/256\rceil \times 4, ldb)} + \frac{\lceil m/256\rceil \times 256 \times n}{v_{outB}(\lceil m/256\rceil \times 4, ldb)} \\ &+ \max\left(\sum_{i=1}^{n} \frac{i \times m}{v_{bcastB}}, \frac{n^2 \times \lceil m/256\rceil \times 256}{v_{flop}}\right) \end{aligned} \tag{1a}$$

$$\begin{aligned} T_{basic\_gemm}&(m,n,k,lda,ldb,ldc) \\ &= \max(T_{calc}, T_{inC} + T_{outC} + T_{inA} + T_{inB}) \\ &= \max\left(\frac{Me \times Ne \times Ke}{v_{flop}}, \frac{Me \times Ne}{v_{inB}(blkN, ldc)} + \frac{Me \times Ne}{v_{outB}(blkN, ldc)}\right. \\ &\left. + \frac{numN \times Me \times Ke}{v_{inB}(blkM, lda)} + \frac{numM \times Ke \times Ne}{v_{inB}(blkN, ldb)}\right) \end{aligned} \tag{1b}$$

3.4.2 Performance Model for Combined Kernels

Since the combined kernels can be decomposed into the combination of basic kernels, we build the performance models for the combined kernels on top of the basic kernels.

$$T_{comb\_syrk}(n,k,lda,ldc) = T_{basic\_gemm}(n,n,k,lda,lda,ldc) \tag{2a}$$

$$\begin{aligned} T_{comb\_trsm}(m,n,lda,ldb,p1) = &\sum_{i=0}^{numP1-1} (T_{basic\_trsm}(m,p1,ldb) \\ &+ T_{basic\_gemm}(m,p,(n-(i+1)\times p1),ldb,lda,ldb)) \end{aligned} \tag{2b}$$

$$\begin{aligned} T_{comb\_potrf}(n,lda,p2,p1) = &\sum_{i=0}^{numP2-1} (T_{comb\_syrk}(p2,i\times p2,lda,lda) \\ &+ T_{basic\_gemm}((n-(i+1)\times p2),p2,i\times p2,lda,lda,lda) \\ &+ T_{comb\_trsm}((n-(i+1)\times p2),p2,lda,lda,p1) \\ &+ T_{mpe\_potrf2}(p2,lda)\times 64) \end{aligned} \tag{2c}$$

The performance model of *syrk* contains four parameter variables, where n and k refer to the sizes of input matrices, and lda and ldc refer to the leading dimensions of input matrices. The performance model is defined in Equation (2a).

The computation time of the combined *trsm* kernel can be decomposed into the computation time of $numP1$ panels. The width of each panel is $p1 = panel1$. Upon each panel, a basic *trsm* kernel and a basic *gemm* kernel is applied. The performance model is defined in Equation (2b).

The computation time of the combined *potrf* kernel can also be decomposed into the computation time of $numP2$ panels. The width of each panel is $p2 = panel2$. Upon each panel, four different kernels are applied. The size of input matrix is denoted by n, and the leading dimension is denoted by lda. The performance model is defined in Equation (2c). Note that, *potrf*2 kernel is performed on MPE. The float-point performance of MPE and a single CPE is roughly the same, so $T_{mpe\_potrf2}(n,lda) = \frac{n^3/3+n^2/2+n/6}{v_{flop}}$. Therefore, we add a penalty factor with the value of 64 to it, which can reduce T_{mpe_potrf2} during the auto-tuning.

3.4.3 Performance Model by Putting Altogether

Based on the performance models of the basic and combined kernels, we can build the overall performance model of our proposed *swCholesky*.

$$F(nrelax, zrelax, p1, p2) = \frac{1}{4}\sum_{i=0}^{3} f(nrelax[i], zrelax[i], p1, p2) \tag{3a}$$

$$f(ns, z, p1, p2) = \frac{1}{ns-1}\sum_{nscol2=1}^{ns-1} \Delta T \tag{3b}$$

$$\begin{aligned}\Delta T = {} & T_{snf}(ns, ldf, p1, p2) \\ & - T_{sn}(nscol1, ld1, p1, p2) - T_{sn}(nscol2, ld2, p1, p2)\end{aligned} \tag{3c}$$

$$\begin{aligned}T_{sn}(ns, ld, p1, p2) = {} & \lambda(T_{comb\_syrk}(ns, ns, ld, ld) \\ & + T_{basic\_gemm}(ld-ns, ns, ns, ld, ld)) + T_{comb\_potrf}(ns, ld, p2, p1) \\ & + T_{comb\_trsm}(ld-ns, ns, ld, p1)\end{aligned} \tag{3d}$$

In symbolic part, there are 8 parameters including $nrelax[0:3]$ and $zrelax[0:3]$ controlling the merging of adjacent *supernodes*, where $nrelax[0:2]$ denote the maximum width (defined as ns) of a merged *supernode*, and $zrelax[0:2]$ denote the proportion of non-zero elements (defined as z) of a merged *supernode*. The $nrelax[3]$ and $zrelax[3]$ are set to relatively large values to improve the accuracy of our performance model. These 8 parameters form four cases when the *supernodes* can be merged: *1)* $ns \le nrelax0$ and $z \le zrelax3$, *2)* $ns \le nrelax1$ and $z \le zrelax0$, *3)* $ns \le nrelax2$ and $z \le zrelax1$, and *4)* $ns \le zrelax3$ and $z \le zrelax2$. Considering the parameter $panel1$ and $panel2$ in *swCHOLBLAS* that determine the way how the combined kernels are converted to the basic kernels, there are entirely 10 parameters to be included in the performance model.

Assuming that the input lower triangular matrix L of *swCholesky* contains N columns and Nnz nonzero elements. The approximate nonzero density of L is $d = \frac{2\times(Nnz-N)}{N^2}$. Assuming that the width of two adjacent *supernodes* $sn1$ and $sn2$ is $nscol1$ and $nscol2$ respectively, the width of the merged *supernode* snf is $ns = nscol1 + nscol2$. Assuming the nonzero density $d1$ of $S1$ is d, and the same applies to the nonzero density $d2$ of $S2$, the number of rows for $sn1$, $sn2$ and snf can be estimated by $ld1 = N \times d + nscol1$, $ld2 = N \times d + nscol2$ and $ldf = N \times (1-z) + ns$.

The execution time of *supernode* sn is defined in Equation (3d). The $\lambda > 1$ because each *supernode* (except the root) has at least one ancestor, and thus at least one *syrk* kernel and one *gemm* kernel should be applied. But the coefficient of snf is 2λ because the merged *supernode* has at least two ancestors. The execution time difference before and after the merging is defined in Equation (3c). Therefore, the overall performance model of *swCholesky* can be defined in Equations (3).

3.4.4 Auto-tuning using Simulated Annealing Algorithm

The constraints for optimization using SA are as follows: *1)* $p1$ and $p2 \in [64, 1024]$, *2)* $nrelax[0:2] \in (0, 512)$, $nrelax3 \in (512, 1024)$, and *3)* $zrelax[0:2] \in (0, 1)$, $zrelax3 \in (0.8, 1.0)$, and the step is set to 0.01. The objective function of the SA algorithm is to minimize the performance model F of *swCholesky*, with the parameters to be determined including $nrelax[0:3]$, $zrelax[0:3]$, $p1$ and $p2$.

The SA algorithm searches for the optimal parameter settings in the following procedure. Firstly, the parameters

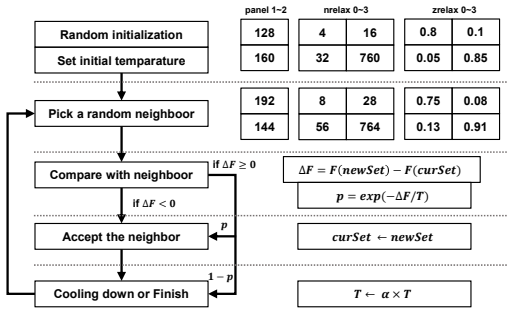


Fig. 6. The Auto-tuning mechanism using simulated annealing.

are initialized to a set of random values within their ranges, and this initial set is denoted by $initSet$. The auto-tuning process is shown in Figure 6, where T refers to the temperature, and α refers to the cooling coefficient. A new set is generated at the beginning of each iteration, which is denoted by $newSet$, and the set generated in the previous iteration is denoted by $curSet$. Then, the algorithm calculates the increment $\Delta F = F(newSet) - F(curSet)$. If $\Delta F < 0$, it accepts $newSet$ and set $curSet \leftarrow newSet$, otherwise, it accepts $newSet$ with the probability $p = \exp(-\Delta F/T)$. Unless reaching the maximum number of iterations, the algorithm sets $T \leftarrow \alpha \times T$ and continues with the next iteration. When the algorithm terminates, the parameters are guaranteed to reach the optimal.

4 Evaluation

4.1 Experimental Setup

colorred All experiments are conducted in double precision. Our experiments are conducted on a Sunway SW26010 processor. We evaluate the performance of *swCholesky* on a CG, and then evaluate its scalability from 1 CG to 4 CGs within a Sunway node. We select 24 representative sparse positive definite matrices from SuiteSparse Matrix Collection [15], the details of these matrices are listed in Table 2. These datasets have various sparse structures as well as diverse *fill-in* ratios. We select the Level-set [9] implementation of sparse Cholesky factorization on MPE as the baseline. To compare with the-state-of-the-art implementations, we parallelize the Level-set [9] and H-levelset [10] methods on Sunway processor by using the CPEs. In addition, we compare to *CHOLMOD* (implementation of the level-set method [12]) that uses Nvidia K40 GPU for acceleration.

4.2 Performance of *swCHOLBLAS*

Figure 7 presents the performance of *swCHOLBLAS* compared to two other implementations using *xMath* and *xMath* + combined kernels. The implementation of *xMath*+combined kernels replaces the basic kernels of *swCHOLBLAS* with *xMath* kernels. Note that, all implementations are based on the design of kernel task queues described in Section 3.2. The average speedup of *xMath*, *xMath* + combined kernels and *swCHOLBLAS* is $38.57\times$, $34.69\times$ and $38.99\times$ respectively. Across all datasets, *swCHOLBLAS* achieves comparable speedup to *xMath*. This is because both *xMath* and *swCHOLBLAS* are highly optimized for

TABLE 2
The datasets used for evaluation.

Matrix shape	Matrix	row×col	nnz	Lnnz
	2cubes_sphere	101K × 101K	874K	49M
	apache2	715K × 715K	2.8M	165M
	bmw7st_1	141K × 141K	3.7M	29.3M
	bmwcra_1	149K × 149K	5.4M	79.5M
	bundle_adj	513K × 513K	10.3M	11.3M
	cfd2	123K × 123K	1.6M	44.6M
	crankseg_2	64K × 64K	7.1M	54.2M
	Dubcova3	147K × 147K	1.9M	9.5M
	ecology2	1M × 1M	3M	54.3M
	hood	221K × 221K	5.5M	30.8M
	ldoor	952K × 952K	23.7M	163M
	m_t1	96K × 96K	4.9M	37.4M
	msdoor	416K × 416K	10.3M	60.2M
	nd12k	36K × 36K	7.1M	161.9M
	nd24k	72K × 72K	14.4M	435.9M
	parabolic_fem	526K × 2.1M	35M	44.6M
	PFlow_742	743K × 743K	19M	598M
	pwtk	218K × 218K	5.9M	55.1M
	raefsky4	20K × 20K	674K	6.8M
	thermal2	1.2M × 1.2M	4.9M	71.8M
	thermocech_dM	204K × 204K	814K	9.6M
	tmt_sym	727K × 727K	2.9M	41.9M
	vanbody	47K × 47K	1.2M	7M
	x104	108K × 108K	5.1M	33.1M

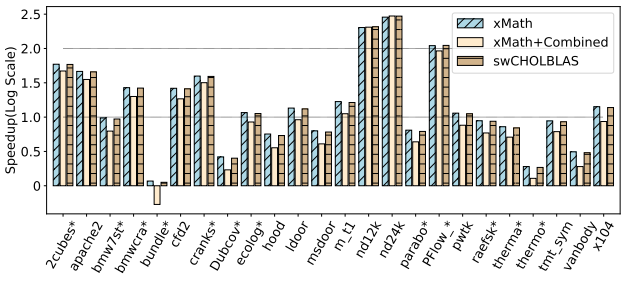


Fig. 7. Performance comparison of *xMath*, *xMath* with combined kernels and *swCHOLBLAS*. The x-axis indicates the different datasets. The y-axis indicates the speedup normalized to baseline (log scaled).

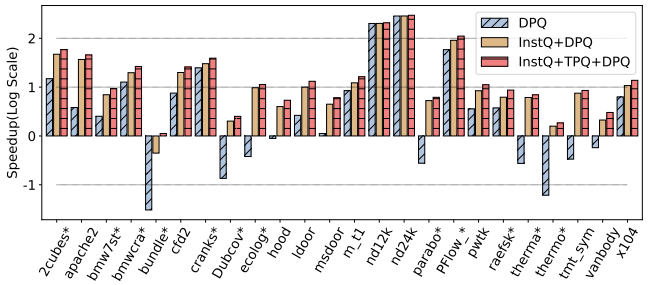


Fig. 8. Performance comparison of using multiple kernel task queues. The x-axis indicates the different datasets. The y-axis indicates the speedup normalized to baseline (log scaled).

dense kernels to take full advantage of Sunway architecture such as many-core parallelization, LDM and vectorization. Although the performance of *swCHOLBLAS* is similar to *xMath*, it supports parallelism by using the 4 CGs within a Sunway node, which is currently not supported by *xMath*. This justifies our design and implementation of *swCHOLBLAS* for better utilizing the computing resources beyond a single CG on Sunway.

We also notice that the performance of *swCHOLBLAS* and xMath is roughly the same, however, the performance of xMath+combined kernels is slightly worse. This is because *xMath* is primarily optimized for large-scale kernels. When the implementation of *xMath* + combined kernels, *xMath* is mainly used to handle the small-scale kernels, which hardly gains any performance benefit. However, the basic kernels in *swCHOLBLAS* are specifically optimized for small-scale kernels, whereas the combined kernels are optimized targeting the large-scale kernels. Therefore, the combination of basic kernels and combined kernels in *swCHOLBLAS* is capable of handling kernels with varying scales efficiently.

On dataset `bundle_adj`, *swCHOLBLAS* and xMath achieves only 1.12× and 1.17× speedup respectively compared to the baseline. This is due to the unusual non-zero structure of `bundle_adj`. It is a "arrowhead" matrix in which the first column, the first row as well as the diagonal are dense. Therefore, nearly all kernels applied to it during the sparse Cholesky factorization deal with the same inputs. The MPE can leverage the data locality to maximize the data re-use with the help of the hardware cache. However, the *xMath* and *swCHOLBLAS* are optimized for the computation of the kernels without considering the data locality across kernels. Whereas with dataset `nd12k` and `nd24k`, the non-zero elements are almost evenly distributed within the matrix, which leads to large *fill-in*. During the Cholesky factorization, a plenty of large-scale kernels are performed, where both *swCHOLBLAS* and *xMath* achieve better performance with super-linear speedup.

4.3 Performance of Kernel Task Queues

We evaluate the design of multiple kernel task queues and compare the performance of using *DPQ*, *InstQ+DPQ* and *InstQ+TPQ+DPQ*. We use *swCHOLBLAS* for the dense kernels used in the computation. The performance of using kernel task queues is shown in Figure 8. With more queues

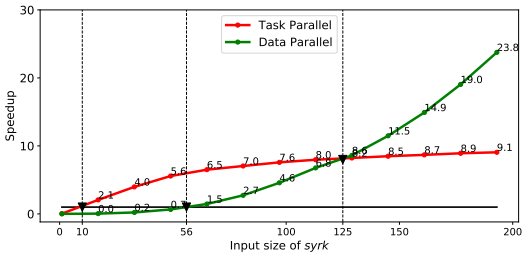


Fig. 9. Performance comparison of small-scale *syrk* kernel running on 64 CPEs in task parallel manner and in data parallel manner. The x-axis indicates the size of the input matrix. The y-axis indicates the speedup normalized to MPE.

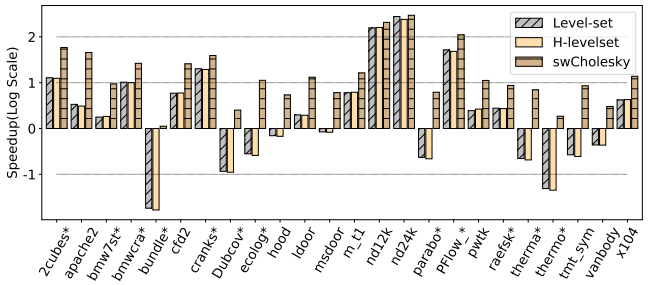


Fig. 10. Performance comparison of Level-set, H-levelset and *swCholesky* running on 1 CG. The x-axis indicates the different datasets. The y-axis is the speedup normalized to the baseline (log scaled).

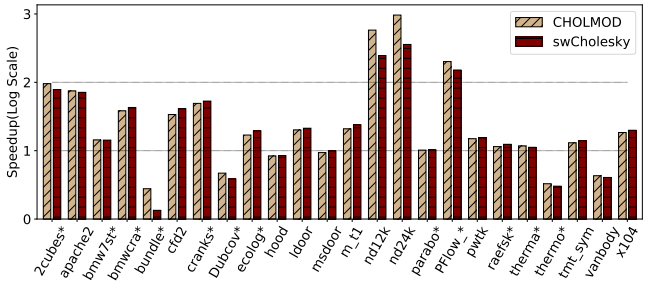


Fig. 11. Performance comparison of *CHOLMOD* running on a Nvidia K40 GPU (1.43TFlops) and *swCholesky* running on 2 Sunway CGs (1.53TFlops). The x-axis indicates the different datasets. The y-axis is the speedup normalized to the baseline (log scaled).

applied, the performance of *swCholesky* across all matrices increases constantly. The average speedup of using *DPQ*, *InstQ+DPQ* and *InstQ+TPQ+DPQ* is $23.56\times$, $34.53\times$ and $38.99\times$ respectively. This is because the kernel task queues are designed for kernels with different scales, therefore with more queues applied, the kernel scheduling becomes more efficient. As described in Section 3.2, the varying sparse structures of matrices in sparse Cholesky factorization lead to kernels with diverse scales. The experimental results verify our observations.

To further investigate the performance improvement with the design of multiple task queues in *swCholesky*, we evaluate the performance of small-scale *syrk* kernel on 64 CPEs in task parallel manner and in data parallel manner, respectively. The input and output matrices are square matrices of the same size. As shown in Figure 9, it is evident that when the size of the input matrix is extremely small (from size 0 to 8), the MPE achieves the best performance. As the size of the input matrix scales (from size 8 to 128), executing the kernels on 64 CPEs in task parallel manner achieves the optimal performance. When the input matrix is large enough (> 128), executing the kernels on 64 CPEs in data parallel manner achieves the optimal performance. The above evaluation results justify our design of multiple kernel task queues.

Besides, we notice that on dataset `bundle_adj`, `Dubcova3`, `ecology2`, `parabolic_fem`, `thermal2`, `thermomech_dM` and `tmt_sym`, the performance of using *DPQ* is worse than the baseline. This is because the matrices in these datasets are either banded or diagonal matrices. Therefore, many small-scale kernels are assigned to *DPQ* and then scheduled on CPEs, which leads to large amount useless computations of padding and thus deteriorates the performance using *DPQ*. As for dataset `nd12k` and `nd24k`, the vast majority of the kernels are large-scale kernels, thus the kernels are assigned to *DPQ*. Therefore, the performance becomes almost identical with different number of queues applied.

4.4 Overall Performance

4.4.1 Overall Performance on 1 CG

Figure 10 shows the performance comparison of Level-set, H-levelset and *swCholesky* running on 1 CG. The Level-set and H-levelset methods are implemented using *xMath* so that they can use the CPEs for parallelization. We choose the performance of Level-set running on MPE as the baseline. The average speedup of Level-set, H-levelset and *swCholesky* across all datasets is $18.87\times$, $16.94\times$ and $38.99\times$ compared to the baseline. It is clear that *swCholesky* achieves better performance than both Level-set and H-levelset on Sunway. This is because *swCholesky* can better adapt to the varying sparse structures of different datasets with the design of multiple kernel task queues.

4.4.2 Performance Comparison to CHOLMOD

Figure 11 shows the performance comparison between *swCholesky* and *CHOLMOD* [16] (v3.0.11). We run *swCholesky* and *CHOLMOD* on 2 Sunway CGs (1.53TFlops) and and a K40 GPU (1.43TFlops) respectively, since such a setup delivers similar peak performance in double precision. We use the performance of Level-set method running on one MPE as the baseline. The datasets used in both implementations are re-ordered by *METIS* [17] (v5.1.0). It is clear that *swCholesky* achieves competitive performance compared to *CHOLMOD* across most of the datasets. This is because the tailored dense kernel library in *swCholesky* is highly optimized targeting the Sunway architecture with better utilization of computing resources beyond a single CG. However, on dataset `nd12k` and `n24k`, *CHOLMOD* performs better. This is because the vast majority of the kernels on these two datasets are large-scale kernels, which can benefit from higher memory bandwidth of the GDDR5 memory on K40 (228GB/s) compared to the limited bandwidth on 2 Sunway CGs (58GB/s).

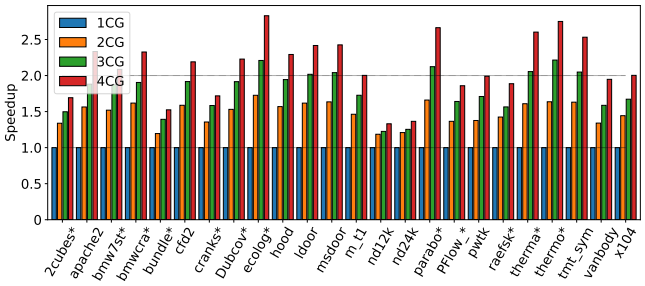


Fig. 12. The scalability of *swCholesky* across multiple CGs on one Sunway processor.

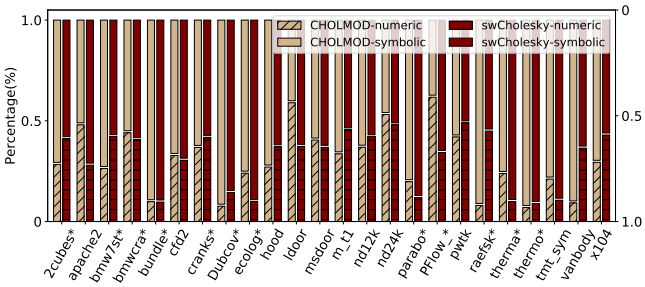


Fig. 13. The percentage of the execution time for the symbolic and numeric parts of *swCholesky* (on 2 Sunway CGs), as well as *CHOLMOD* (on K40 GPU) in a single sparse Cholesky factorization.

4.5 Beyond a single CG

Figure 12 shows the strong scalability results of *swCholesky* from 1 CG to 4 CGs on one Sunway processor. We use the performance of *swCholesky* on 1 CG as the baseline. Across all datasets, *swCholesky* achieves $1.49\times$, $1.80\times$ and $2.13\times$ speedup on average when scaling from 2 CGs to 4 CGs. As the number of CGs increases, the performance of *swCholesky* improves significantly. This is because *swCholesky* adopts the idea from H-levelset to partition *supernodes* into independent w-partitions, therefore different w-partitions are calculated on different CGs in task parallel manner. The reason why *swCholesky* cannot achieve linear scalability is that as the number of CGs increases, a large number of DMA transfers also increases significantly and thus becomes a bottleneck due to the limited NoC bandwidth connecting the 4 CGs. Particularly, the scalability on dataset `nd12k` and `nd24k` is worse than the others. This is because the performance on `nd12k` and `nd24k` is already limited by the memory bandwidth even on 1 CG. When scaling to more CGs, the limited NoC bandwidth among CGs further restricts the performance.

4.6 The Overhead of Symbolic Analysis

Figure 13 shows the percentage of execution time for both the symbolic and numeric parts of *swCholesky* in a single factorization running on 2 CGs compared to *CHOLMOD* running on K40 GPU. As mentioned in Section 4.4.2, both platforms have similar peak performance. It is obvious that the time spent at symbolic part of *swCholesky* is similar with that of *CHOLMOD*, although the symbolic part is mainly performed on MPE (*swCholesky*) and CPU (*CHOLMOD*) respectively, where the MPE has much lower frequency and smaller cache than CPU. In real-world applications, the symbolic analysis is not always performed in each factorization. Especially, when the matrices with the same sparse structure but different non-zero values are factorized in a "for" loop, the symbolic analysis only needs to be performed once. In general, the overhead of symbolic analysis in *swCholesky* can be amortized by 3.59 iterations on average across all datasets.

5 Related Work

5.1 Performance Optimization of Sparse Cholesky Factorization

The *supernode* structure of sparse matrices is first proposed by George and Liu [7]. *Supernode* structure is essential to accelerate sparse Cholesky factorization on modern high-performance architectures. Various highly optimized dense linear algebra libraries such as *MKL* [18] for Intel CPU, *CUBLAS* [19] for NVIDIA GPU and *OpenBLAS* [20] for general architectures, are used for acceleration based on *supernode* structure. Many optimizations of sparse Cholesky factorization [21]–[24] are leveraging dense linear algebra libraries on either CPU or GPU. The closest approach to our *swCHOLBLAS* is [13], which proposes tunable Cholesky factorization kernels on GPU, however, it focuses on solving Cholesky factorization with the same problem size. Different from existing works, our *swCHOLBLAS* library is optimized targeting Sunway architecture, that fully utilizes the unique architectural features such as many-core and LDM.

Many highly optimized direct sparse solvers,such as *MKL Pardiso* [25], *PaStiX* [26] and *CHOLMOD* [16] contain sparse Cholesky factorization. These libraries usually use symbolic analysis to collect the characteristics of the matrices and then schedule the numeric calculation based on the collected information. *Sympiler* [27] and *Parsy* [10] adopt the same methodology, and further divide the supernode into H-levelset by using the load-balanced level coarsening algorithm, which improves load balance and reduces synchronization overhead. Our *swCholesky* adapts the H-levelset methodology to Sunway by exploiting the architecture advantage with optimized dense kernel library and specially designed kernel task queues.

There are also research works that are optimizing sparse Cholesky factorization for specific situations, such as Cholesky factorization with dynamic *supernodes* [28] for matrices with frequent changes, approximate Cholesky factorization [29] as the preconditioner.

5.2 Performance Optimization on Sunway

In recent years, many optimization techniques are explored on Sunway to expose the massive computation power to the applications [30]–[34]. Molecular Dynamics [30] simulates the physical movements of atoms and molecules at Peta-scale, and it employs specific optimizations such us a software cache strategy and a full pipeline acceleration for Sunway; Earthquake Simulation [31] re-designs the simulation algorithms to reduce memory costs and adapts a genetic algorithm to tune various parameters. Deep learning applications are also optimized on Sunway such as *swCaffe* [35].

It re-designs various DNN layers targeting Sunway architecture and customizes the communication strategy according to the network topology of Sunway TaihuLight supercomputer. Many important computation kernels, such as *SpMV* [36], *SpTRSV* [37], [38] and *swDNN* [39] are also highly optimized on Sunway. *SpMV* [36] proposes a novel three level partitioning scheme on Sunway. *swDNN* [39] maps the convolution kernel in CNN into 260 cores within one Sunway node leveraging the unique register level communication. The existing optimization works on Sunway provide us valuable experience on how to leverage the architecture advantage for better performance during our design of *swCholesky*.

5.3 Batched BLAS

Many applications such as multifrontal solvers [40] and direct-iterative preconditioned solvers [41] perform quite a number of small matrix operations, and thus rely on batched operations to improve the performance. The batched BLAS has been proposed for solving a large number of small matrix operations simultaneously. Dongarra et al. propose a set of APIs for batched BLAS [42], and discuss the advantages and drawbacks of the batched BLAS proposals [43]. The data layout of batched BLAS including pointer-to-pointer layout, stridden layout, interleaved layout and their variants are also attracting research attentions [43], [44]. As for libraries, both Intel *MKL* and NVIDIA *CuBLAS* have implemented several subroutines such as *gemm* and *trsm*. MAGMA [45] has implemented more subroutines including batched (level-0, level-1 and level-2) BLAS, linear system solvers and matrix factorizations [46], [47]. Moreover, variable batched operations [48], [49] enable solving many independent problems of different sizes. Although the task parallel queue of *swCholesky* has similarity with batched BLAS, it can dynamically map kernels between MPE and CPEs according to the batch size. The unique design of *swCholesky* provides useful insights to implement batched subroutines (e.g. BLAS, LAPACK) on heterogeneous many-core processors.

5.4 Task-based Runtime System

Task-based runtime systems such as StarPU [50], PaRSEC (previously DAGuE [51]), OmpSs [52] and XKAAPI [53], allow developers to express applications using a series of tasks, and then map the tasks to the configured hardware such as multiple CPUs and accelerators. The task-based runtime systems are usually responsible for data transfer and task scheduling, which eases migrating applications to various hardware. Among them, both OmpSs and XKAAPI use pragma-based interface similar to OpenMP, which is friendly to the developers due to the simplicity of programming, but difficult to achieve better performance than other runtime systems. StarPU and PaRSEC build a DAG to represent dependent tasks for each application, then schedule tasks according to the DAG. Additionally, StarPU provides APIs that allows developers to implement unique task scheduling strategies. Several research works have reported improved performance by adopting task-based runtime systems, for instance sparse linear solvers PaStiX [26], qr_mumps [54], linear algebra libraries MAGMA [45], DPLASMA [55] and N-body simulation ScalFMM [56]. However, the task-based runtime system is not adopted in *swCholesky*. This is because the kernel execution sequence of *swCholesky* is determined once the H-levelset is generated. Then *swCholesky* maps the kernels to MPE and CPEs dynamically with the help of kernel task queues. It is unnecessary for *swCholesky* to adopt the complex task-based runtime system.

6 Conclusion

In this paper, we propose *swCholesky*, an effective design and implementation of sparse Cholesky factorization on Sunway many-core processor. In *swCholesky*, we propose multiple kernel task queues to address the diverse computation scale of the dense kernels. According to the varying computation scales, the kernels are dynamically assigned to the MPE for sequential execution and the 64 CPEs for parallel execution in either task parallel or data parallel manner, which improves the computation efficiency of the dense kernels. In addition, we propose a dense kernel library, *swCHOLBLAS*, that is tailored for sparse Cholesky factorization on Sunway by better leveraging the computing resources of multiple CGs. Moreover, a performance auto-tuning mechanism is proposed to automatically search for the optimal parameter settings in *swCholesky* using kernel performance models and simulated annealing algorithm. Our experimental results on 24 representative sparse matrix datasets demonstrate that *swCholesky* achieve better performance on Sunway, compared to the-state-of-the-art methods such as Level-set, H-levelset and *CHOLMOD*.

Acknowledgments

The authors would like to thank all anonymous reviewers for their insightful comments and suggestions. This work is supported by National Key R&D Program of China (Grant No. 2016YFB1000503) and National Natural Science Foundation of China (Grant No. 61502019). Hailong Yang is the corresponding author.

References

[1] J. Dongarra, G. Peterson, S. Tomov, J. Allred, V. Natoli, and D. Richie, "Exploring new architectures in accelerating cfd for air force applications," in *2008 DoD HPCMP Users Group Conference*. IEEE, 2008, pp. 472–478.

[2] É. Béchet, H. Minnebo, N. Moës, and B. Burgardt, "Improved implementation and robustness study of the x-fem for stress analysis around cracks," *International Journal for Numerical Methods in Engineering*, vol. 64, no. 8, pp. 1033–1056, 2005.

[3] L. Lines and S. Treitel, "Tutorial: A review of least-squares inversion and its application to geophysical problems," *Geophysical prospecting*, vol. 32, no. 2, pp. 159–186, 1984.

[4] J. T. Smith, "Conservative modeling of 3-d electromagnetic fields, part ii: Biconjugate gradient solution and an accelerator," *Geophysics*, vol. 61, no. 5, pp. 1319–1324, 1996.

[5] T. A. Davis, *Direct methods for sparse linear systems*. Siam, 2006, vol. 2.

[6] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.

[7] A. George and J. W. Liu, "Computer solution of large sparse positive definite," 1981.

[8] A. Pothen and S. Toledo, "Elimination structures in scientific computing." 2004.

[9] D. P. O'leary and G. Stewart, "Data-flow algorithms for parallel matrix computation," *Communications of the ACM*, vol. 28, no. 8, pp. 840–853, 1985.

[10] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Parsy: inspection and transformation of sparse matrix computations for parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 62.

[11] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal factorization of sparse spd matrices on gpus," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 372–383.

[12] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse cholesky factorization on gpus," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2014, pp. 9–16.

[13] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched cholesky factorization and solve for nvidia gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2036–2048, 2016.

[14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

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

[16] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, p. 22, 2008.

[17] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[18] 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.

[19] C. Nvidia, "Cublas library," *NVIDIA Corporation, Santa Clara, California*, vol. 15, no. 27, p. 31, 2008.

[20] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: automatically generate high performance dense linear algebra kernels on x86 cpus," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[21] V. Volkov and J. Demmel, "Lu, qr and cholesky factorizations using vector capabilities of gpus," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49*, vol. 49, 2008.

[22] M. Tang, M. Gadou, S. C. Rennich, T. A. Davis, and S. Ranka, "A multilevel subtree method for single and batched sparse cholesky factorization," in *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 2018, p. 50.

[23] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A scalable high performant cholesky factorization for multicore with gpu accelerators," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 93–101.

[24] D. Irony, G. Shklarski, and S. Toledo, "Parallel and fully recursive multifrontal sparse cholesky," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 425–440, 2004.

[25] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, "Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 69–78, 2001.

[26] P. Hénon, P. Ramet, and J. Roman, "Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, 2002.

[27] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi, "Sympiler: transforming sparse matrix codes by decoupling symbolic analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 13.

[28] T. A. Davis and W. W. Hager, "Dynamic supernodes in sparse cholesky update/downdate and triangular solves," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 4, p. 27, 2009.

[29] D. A. Sushnikova and I. V. Oseledets, ""compress and eliminate" solver for symmetric positive definite sparse matrices," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. A1742–A1762, 2018.

[30] B. Chen, H. Fu, Y. Wei, C. He, W. Zhang, Y. Li, W. Wan, W. Zhang, L. Gan, W. Zhang *et al.*, "Simulating the wenchuan earthquake with accurate surface topography on sunway taihulight," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 40.

[31] X. Duan, P. Gao, T. Zhang, M. Zhang, W. Liu, W. Zhang, W. Xue, H. Fu, L. Gan, D. Chen *et al.*, "Redesigning lammps for peta-scale and hundred-billion-atom simulation on sunway taihulight," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 148–159.

[32] X. Zhong, M. Li, H. Yang, Y. Liu, and D. Qian, "swmr: A framework for accelerating mapreduce applications on sunway taihulight," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[33] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu *et al.*, "Shentu: processing multi-trillion edge graphs on millions of cores in seconds," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 56.

[34] H. Fu, J. Liao, N. Ding, X. Duan, L. Gan, Y. Liang, X. Wang, J. Yang, Y. Zheng, W. Liu *et al.*, "Redesigning cam-se for peta-scale climate modeling performance and ultra-high resolution on sunway taihulight," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 1.

[35] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, and G. Yang, "swcaffe: A parallel framework for accelerating deep learning applications on sunway taihulight," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 413–422.

[36] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, and X. Liu, "Towards efficient spmv on sunway manycore architectures," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 363–373.

[37] M. Li, Y. Liu, H. Yang, Z. Luan, and D. Qian, "Multi-role sptrsv on sunway many-core architecture," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2018, pp. 594–601.

[38] X. Wang, W. Liu, W. Xue, and L. Wu, "swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures," in *ACM SIGPLAN Notices*, vol. 53, no. 1. ACM, 2018, pp. 338–353.

[39] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, and G. Yang, "swdnn: A library for accelerating deep learning applications on sunway taihulight," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 615–624.

[40] 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.

[41] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[42] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov *et al.*, "A proposed api for batched basic linear algebra subprograms," 2016.

[43] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched blas on modern high-performance computing systems," *Procedia Computer Science*, vol. 108, pp. 495–504, 2017.

[44] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact blas and lapack kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 55.

[45] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[46] A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra, "A framework for batched and gpu-resident factorization algorithms applied to block householder transformations," in *International Conference on High Performance Computing*. Springer, 2015, pp. 31–47.

[47] A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, and J. Dongarra, "A guide for achieving high performance with very small matrices on gpu: A case study of batched lu and cholesky factorizations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 973–984, 2018.

[48] P. Valero-Lara, I. Martinez-Perez, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta, "Variable batched dgemm," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 2018, pp. 363–367.

[49] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ortí, "Variable-size batched gauss–jordan elimination for block-jacobi preconditioning on graphics processors," *Parallel Computing*, vol. 81, pp. 131–146, 2019.

[50] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[51] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[52] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[53] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 1299–1308.

[54] A. Buttari, "Fine-grained multithreading for the multifrontal qr factorization of sparse matrices," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C323–C345, 2013.

[55] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief *et al.*, "Distibuted dense numerical linear algebra algorithms on massively parallel architectures: Dplasma," 2010.

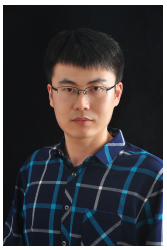
[56] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2608–2629, 2016.

Mingzhen Li is a PhD student in School of Computer Science and Engineering, Beihang University. He is currently working on identifying performance opportunities for scientific applications and sparse matrix algorithms. His research interests include HPC, performance optimization, and deep learning.

Yi Liu is a professor in School of Computer Science and Engineering, and Director of the Sino-German Joint Software Institute (JSI) at Beihang University, China. In 2000, he completed Ph.D in Department of Computer Science of Xi'an Jiaotong University. His research interests include computer architecture, HPC and new generation of network technology.

Hailong Yang is an assistant professor in School of Computer Science and Engineering, Beihang University. He received the Ph.D degree in the School of Computer Science and Engineering, Beihang University in 2014. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency. He is a member of China Computer Federation (CCF).

Zhongzhi Luan received the Ph.D. in the School of Computer Science of Xi'an Jiaotong University. He is an Associate Professor of Computer Science and Engineering, and Assistant Director of the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China. Since 2003, His research interests including distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.



Lin Gan is an assistant researcher in the Department of Computer Science and Technology at Tsinghua University, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include high performance computing solutions based on hybrid platforms such as GPUs, FPGAs, and Sunway CPUs. Gan received a PhD in computer science from Tsinghua University. He is the recipient of the 2016 ACM Gordon Bell Prize, the 2017 ACM Gordon Bell Prize Finalist, the 2018 IEEE-CS TCHPC Early Career Researchers Award for Excellence in HPC, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc. He is a member of IEEE.

Guangwen Yang is a professor in the Department of Computer Science and Technology at Tsinghua University, and the director of the National Supercomputing Center in Wuxi. His research interests include parallel algorithms, cloud computing, and the earth system model. Yang received a PhD in computer science from Tsinghua University. He has received the ACM Gordon Bell Prize in the year of 2016 and 2017, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc. He is a member of IEEE.

Depei Qian is a professor at the Department of Computer Science and Engineering, Beihang University, China. He received his master degree from University of North Texas in 1984. He is currently serving as the chief scientist of China National High Technology Program (863 Program) on high productivity computer and service environment. He is also a fellow of China Computer Federation (CCF). His research interests include innovative technologies in distributed computing, high performance computing and computer architecture.