Implementing a Parallel Matrix Factorization Library on the Cell Broadband Engine

Vishwas B. C., Abhishek Gadia, Mainak Chaudhuri*
Department of Computer Science and Engineering
Indian Institute of Technology
Kanpur 208016
INDIA
mainakc@cse.iitk.ac.in

15th May, 2008
Last revision: 21st September, 2008

Abstract

Matrix factorization (or often called decomposition) is a frequently used kernel in a large number of applications ranging from linear solvers to data clustering and machine learning. The central contribution of this paper is a thorough performance study of four popular matrix factorization techniques, namely, LU, Cholesky, QR, and SVD on the STI Cell broadband engine. The paper explores algorithmic as well as implementation challenges related to the Cell chip-multiprocessor and explains how we achieve near-linear speedup on most of the factorization techniques for a range of matrix sizes. For each of the factorization routines, we identify the bottleneck kernels and explain how we have attempted to resolve the bottleneck and to what extent we have been successful. Our implementations, for the largest data sets that we use, running on a two-node 3.2 GHz Cell BladeCenter (exercising a total of sixteen SPEs), on average, deliver 203.9, 284.6, 81.5, 243.9, and 54.0 GFLOPS for dense LU, dense Cholesky, sparse Cholesky, QR, and SVD, respectively. The implementations achieve speedup of 11.2, 12.8, 10.6, 13.0, and 6.2, respectively for dense LU, dense Cholesky, sparse Cholesky, QR, and SVD, when running on sixteen SPEs. We discuss the interesting interactions that result from parallelization of the factorization routines on a two-node non-uniform memory access (NUMA) Cell Blade cluster.

Keywords: Parallel matrix factorization, Cell broadband engine, Scalability, LU, Cholesky, QR, Singular value decomposition, New data structures.

* Corresponding author. Telephone: 91-512-2597890, FAX: 91-512-2590725, e-mail: mainakc@cse.iitk.ac.in

1 Introduction

Matrix factorization plays an important role in a large number of applications. In its most general form, matrix factorization involves expressing a given matrix as a product of two or more matrices with certain properties. A large number of matrix factorization techniques has been proposed and researched in the matrix computation literature [14] to meet the requirements and needs arising from different application domains. Some of the factorization techniques are categorized into two classes depending on whether the original matrix is dense or sparse. In this paper, we focus on four most commonly used matrix factorization techniques[1], namely, LU, Cholesky, QR and singular value decomposition (SVD). Due to the importance of Cholesky factorization of sparse matrices, we explore, in addition to the dense techniques, sparse Cholesky factorization also.

LU factorization or LU decomposition is perhaps the most primitive and the most popular matrix factorization technique finding applications in direct solvers of linear systems such as Gaussian elimination. LU factorization involves expressing a given matrix as a product of a lower triangular matrix and an upper triangular matrix. Once the factorization is accomplished, simple forward and backward substitution methods can be applied to solve a linear system. LU factorization also turns out to be extremely useful when computing the inverse or determinant of a matrix because computing the inverse or the determinant of a lower or an upper triangular matrix is easy. In this paper, we consider a numerically stable version of LU factorization, commonly known as PLU factorization, where the given matrix is represented as a product of a permutation matrix, a lower triangular matrix, and an upper triangular matrix.[2] This factorization arises from Gaussian elimination with partial pivoting which permutes the rows to avoid pivots with zero or small magnitude.

Cholesky factorization or Cholesky decomposition shares some similarities with LU factorization, but applies only to symmetric positive semi-definite matrices[3] and in this case the factorization can be expressed as U^TU or LL^T where U is an upper triangular matrix and L is a lower triangular matrix. Although Cholesky factorization may appear to be a special case of LU factorization, the symmetry and positive semi-definiteness of the given matrix make Cholesky factorization asymptotically twice faster than LU factorization. Cholesky factorization naturally finds applications in solving symmetric positive definite linear systems such as the ones comprising the normal equations in linear least squares problems, in Monte Carlo simulations with multiple correlated variables for generating the shock vector, in unscented Kalman filters for choosing the sigma points, etc. Cholesky factorization of large sparse matrices poses completely different computational challenges. What makes sparse Cholesky factorization an integral part of any matrix factorization library is that sparse symmetric positive semi-definite linear systems are encountered frequently in interior point methods (IPM) arising from linear programs in relation to a number of important finance applications [5, 24]. In this paper, we explore the performance of a sparse Cholesky factorization technique, in addition to the techniques for dense matrices.

QR factorization and singular value decomposition (SVD) constitute a different class of factorization techniques arising from orthogonalization and eigenvalue problems. The QR factorization represents a given matrix as a product of an orthogonal matrix[4] and an upper triangular matrix which can be rectangular. QR factorization finds applications in solving linear least squares problems and computing eigenvalues of a matrix. SVD deals with expressing a given matrix A as $U\Sigma V^T$ where U and

[1] We focus on real matrices with single-precision floating-point entries only.
[2] A permutation matrix has exactly one non-zero entry in every row and every column and the non-zero entry is unity.
[3] A matrix A is said to be symmetric if $A = A^T$. A matrix A is said to be positive semi-definite if for all non-zero vectors x, x^TAx is non-negative.
[4] The columns of an orthogonal matrix A form a set of orthogonal unit vectors i.e., $A^TA = I$ where I is the identity matrix.

V are orthogonal matrices and Σ is a diagonal matrix. The entries of the matrix Σ are known as the singular values of A. Some of the most prominent applications of SVD include solving homogeneous linear equations, total least squares minimization, computing the rank and span of the range and the null space of a matrix, matrix approximation for a given rank with the goal of minimizing the Frobenius norm, analysis of Tikhonov regularization, principal component analysis and spectral clustering applied to machine learning and control systems, and latent semantic indexing applied to text processing. Finally, SVD is closely related to eigenvalue decomposition of square matrices. Interested readers are referred to [14] for an introduction to applications of SVD.

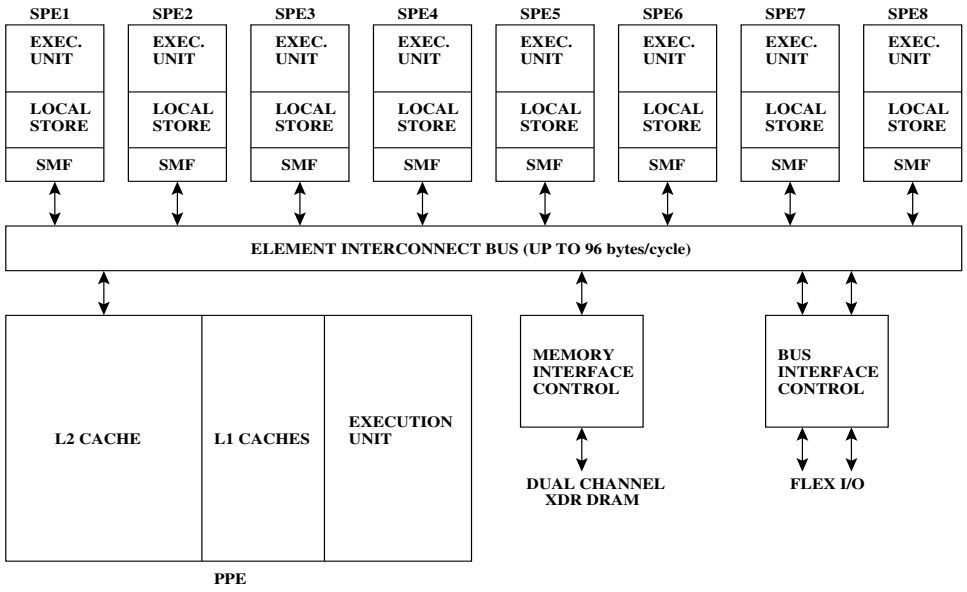


Figure 1: The logical architecture of the Cell processor. The figure neither corresponds to the actual physical floorplan nor is drawn to the scale. SPE stands for synergistic processing element, PPE stands for Power processing element, SMF stands for synergistic memory flow controller, and XDR stands for extreme data rate.

Due to its wide-spread applications, parallel and sequential algorithms for matrix factorization have been extensively explored by the high-performance computing community. The ScaLAPACK [33] distribution offers parallel message-passing implementations of all the four factorization techniques that we consider in this paper. However, the STI Cell broadband engine [15, 16, 23] poses new challenges and often requires substantial re-engineering of the existing algorithms. In return, one gets excellent performance due to the dense computational power available in the Cell broadband engine. While dense Cholesky, QR, and PLU factorizations have been studied on the Cell processor [25, 26, 27], to the best of our knowledge, we are the first to explore sparse Cholesky factorization and singular value decomposition on the Cell processor. Further, we bring this diverse set of factorization techniques under a single design philosophy that we discuss in Section 1.2. Before moving on to a brief discussion about the salient features of the Cell broadband engine in the next section, we would like to note that some of the relevant studies in parallel QR and PLU factorizations include [12, 30, 31]. The SuperMatrix runtime system introduced in [9] keeps track of dependencies between blocks of matrices and schedules the block operations as and when the dependencies are satisfied. The authors demonstrate the applicability of this parallel runtime system on dense Cholesky factorization, triangular inversion, and triangular matrix multiplication. Finally, we note that the Watson Sparse Matrix Package [34] includes a range of sparse linear system solvers.

1.1 Overview of the Cell Architecture

The STI Cell broadband engine is a heterogeneous single-chip multiprocessor (see Figure 1). It has one PowerPC processing element (PPE) and eight simple, yet high-performance, vector units called the synergistic processing elements (SPEs). There is a specialized high-bandwidth ring bus, called the element interconnect bus (EIB) connecting the PPE, the SPEs, the memory controller, and the I/O elements through the bus interface controller. The PPE implements the 64-bit PowerPC 970 instruction set on an in-order execution pipe with support for two-way simultaneous multithreading. It contains a 64-bit general purpose register (GPR) file, a 64-bit floating-point register (FPR) file, and a 128-bit Altivec register file to support a short-vector single-instruction-multiple-data (SIMD) execution unit. The L1 instruction and data caches are 32 KB each and the L2 cache is 512 KB.

Each SPE consists of a synergistic processing unit, a 256 KB local store, and a synergistic memory flow controller. It is equipped with a 128-entry vector register file of width 128 bits each and supports a range of SIMD instructions to concurrently execute two double-precision floating-point operations, four single-precision floating-point operations, eight halfword operations, etc. The vector register file is very flexible in terms of its data interface and in a single access one can read out a 128-bit value, or two 64-bit values, or four 32-bit values, or eight 16-bit values, or just a 128-entry bit vector. The SPEs are particularly efficient in executing SIMD instructions on single-precision floating-point operands, since these instructions are fully pipelined. The execution of double-precision operations on the SPEs is not as efficient. Overall, each SPE can complete four fused multiply-add instructions, each operating on four single-precision values, in a single clock cycle. This translates to a peak throughput of 25.6 GFLOPS at 3.2 GHz frequency leading to an overall 204.8 GFLOPS throughput of all the eight SPEs. To keep the design simple, the SPEs do not offer any special support for scalar data processing. The scalar operations make use of vector instructions with data aligned to the "preferred slot" of the vector registers being acted upon. The preferred slot for most instructions is the leftmost slot in the vector register. The SPEs do not implement any data alignment network in hardware to bring the scalar operands to the appropriate vector register slots. The necessary shift and rotate operations are exposed to the software and must be explicitly programmed before carrying out a scalar operation. If efficiently programmed, the total time taken for data alignment in software turns out to be the same as a hardware implementation would have offered.

The most interesting aspect of the Cell architecture is the explicitly managed 256 KB local load/store storage of each SPE. The communication between the local stores and between each local store and the main memory must be explicitly managed by the software via insertion of direct memory access (DMA) instructions or intrinsics. The DMA instructions come in two major flavors. In one flavor, up to 16 KB of data from a contiguous range of addresses can be transferred between the main memory and the local store. The other DMA flavor supports a gather operation where a list of different addresses can be specified and from each address a 128-bit data element can be fetched. Each address in the list must be aligned to a 16-byte boundary. The scatter counterpart of this gather operation is also available which writes data back to memory at the specified addresses. This list DMA operation is extremely helpful in gathering irregularly accessed data and buffering them in a contiguous region of the local store so that subsequent operations on that buffer can enjoy spatial locality, even though the original access pattern did not have any spatial locality. Managing the local store area appropriately often turns out to be one of the most challenging components of designing parallel algorithms for the Cell broadband engine because the DMA stall cycles play a significant role in determining the end-performance. Although this problem may sound similar to managing the communication in a distributed

memory multiprocessor, the small size of the local store poses significant challenges. Sub-optimal use of this precious local store area may lead to poor scalability. The SPEs, the PPE, and the memory and I/O controllers connect to a high-bandwidth element interconnect bus (EIB). The synergistic memory flow controller on each SPE supports a 16 bytes/cycle interface in each direction with the EIB. The memory controller connects to a dual-channel extreme data rate RAMBUS DRAM module.

1.2 Library Design Philosophy

We follow the algorithms and architecture approach (AAA) introduced in [1] to design our factorization library. In this approach, the algorithms are optimized for the target architecture, which in our case, is the Cell processor. Our algorithms use the square block format to represent and compute on a given matrix. It was pointed out in [13] and [19] that such type of new data structures can lead to high performance compared to more traditional row-major, column-major, and packed formats. Further, it is surprisingly easy to convert an algorithm that works element-wise on the matrices to one that uses square blocks. Essentially, scalar operations become BLAS routines operating on square blocks. We will exploit this observation in formulating the dense Cholesky, PLU, and QR algorithms. Sometimes we will refer to the square blocks as tiles and the corresponding data layout as tile-major or block-major.

We identify three primary philosophies in our design. First, as was shown in [19], dense linear algebra factorization algorithms are mostly comprised of BLAS-3 routines. As a result, our library extensively uses carefully optimized BLAS-2, BLAS-2.5, and BLAS-3 routines. Second, we achieve communication hiding by appropriately scheduling computations. In this case also, the BLAS-3 routines help us because they have a high computation-to-communication ratio [2]. The computation-communication overlap is achieved by exploiting the asynchronous DMA interface of the Cell architecture and using an appropriate number of buffers in the local stores of the SPEs. Finally, we compute on square blocks or tiles of data so that the utilization of the local stores is maximized.

1.3 Result Highlights and Organization

On a dual-node 3.2 GHz Cell BladeCenter, our implementations exercising non-uniform memory access (NUMA)-aware data placement to minimize inter-node DMAs, achieve speedup of 11.2, 12.8, 10.6, 13.0, and 6.2 for 16-way threaded parallel dense PLU, dense Cholesky, sparse Cholesky, QR, and SVD, respectively. The respective delivered GFLOPS are 203.9, 284.6, 81.5, 243.9, and 54.0.

Section 2 discusses the parallel algorithms for dense Cholesky, dense PLU, sparse Cholesky, QR, and SVD. For each of the factorization techniques, we also discuss the implementation challenges specific to the Cell processor. Sections 3 and 4 detail our experimental results and we conclude in Section 5.

2 Parallel Algorithms for Matrix Factorization

This section discusses the parallel algorithms and implementation details on the Cell processor for dense Cholesky and PLU factorization, sparse Cholesky factorization, QR factorization, and singular value decomposition.

2.1 Dense Cholesky Factorization

The left-looking and right-looking variants of Cholesky factorization are well-known. We have chosen the right-looking algorithm for implementation. The details of this algorithm can be found in [3]. A detailed discussion on implementation of the left-looking algorithm on the Cell broadband engine can be found in [25].

In blocked Cholesky, input matrix A can be decomposed as

$$A = \begin{bmatrix} A_{11} & * \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix},$$

where A_{11} and L_{11} are of size $b \times b$, A_{21} and L_{21} are $(n-b) \times b$, and A_{22} and L_{22} are $(n-b) \times (n-b)$. The involved steps in blocked factorization are as follows.

1. Compute Cholesky factorization (using the right-looking algorithm) of the diagonal block. This computes L_{11} corresponding to the diagonal block i.e. $A_{11} = L_{11}L_{11}^T$; $L_{11} = \text{CholeskyFac}(A_{11})$; $A_{11} \leftarrow L_{11}$.
2. Update the block A_{21} based on the diagonal block as $L_{21} = A_{21}L_{11}^{-T}$; $A_{21} \leftarrow L_{21}$.
3. Update the sub-matrix A_{22} with $A_{22} - L_{21}L_{21}^T$.
4. Recurrently repeat the steps 1 to 3 on A_{22}.

For better memory performance, the algorithm needs to be modified to work on smaller tiles that can fit in the on-chip storage. Cholesky Factorization can be decomposed easily into tiled operations on smaller sub-matrices. The input matrix is divided into smaller tiles of size $b \times b$ and all the operations are applied to these tiles. The following elementary single-precision kernels are modified to work on the tiles.

- $SPOTRF(X)$: Computes the Cholesky factorization of a $b \times b$ tile X. This is denoted by the function CholeskyFac in the above algorithm.
- $STRSM(X, D)$: Updates a tile X under the diagonal tile D, which corresponds to step 2 in the above algorithm i.e. $X \leftarrow XD^{-T}$.
- $SSYRK(X, L_1)$: Performs a symmetric rank-k update of the tile X in the sub-matrix A_{22}, which corresponds to step 3 in the above algorithm i.e. $X \leftarrow X - L_1L_1^T$.
- $SGEMM(X, L_1, L_2)$: Performs $X \leftarrow X - L_1L_2^T$.

2.1.1 Implementation on the Cell Processor

In the following, we describe the important details of implementing a single precision Cholesky factorization on the Cell processor. The discussion on huge pages, tile size selection, and double/triple buffering is significantly influenced by the work presented in [25]. We would like to mention that our implementation is derived from the right-looking version of Cholesky factorization, while the work presented in [25] considers the left-looking version.

Block Layout and Huge Pages Row-major storage of a matrix in main memory creates several performance bottlenecks for an algorithm that works on tiled data. A tile is composed of parts of several rows of the matrix. Therefore, the DMA operation that brings a tile into the local store of an SPE can be severely affected by pathological distribution of the rows across the main memory banks. Specifically, a large number of bank conflicts can lead to very poor performance of this list DMA operation. Since the difference in the addresses between two consecutive list elements can be very large (equal to the size of one row of the matrix), the chance of memory bank conflicts is very high. Note that the Cell processor's memory subsystem has 16 banks and the banks are interleaved at cache block boundaries of size 128 bytes. As a result, addresses that are apart by a multiple of 2 KB will suffer from bank conflicts. This essentially means that if the input single precision floating-point matrix has number of columns in multiples of 512, different rows of a tile will access the same main memory bank and suffer from bank conflicts. Another problem is that the rows in the DMA list can be on different physical page frames in memory. As a result, while working on multiple tiles, the SPE can suffer from a large number of TLB misses. The SPE contains a 256-entry TLB and the standard page size is 4 KB.

One solution to these problems is to use a block layout storage for the matrix, where each tile is stored in contiguous memory locations in the main memory. This is achieved by transforming a two-dimensional matrix into a four-dimensional matrix. The outer two dimensions of the matrix refer to the block row and block column while the inner two dimensions refer to the data elements belonging to a tile. For example, if a 1024×1024 matrix is divided into 64×64 tiles, the matrix can be covered using 256 tiles and a particular tile can be addressed using two indices (p, q) where $0 \leq p, q \leq 15$. In the four-dimensional transformed matrix, the element (p, q, i, j) would refer to the $(i, j)^{\text{th}}$ element of the $(p, q)^{\text{th}}$ tile. By sweeping i and j from 0 to 63 we can access the entire tile and more importantly, all these data elements within the tile now have contiguous memory addresses. The memory request for a tile from an SPE will have a single DMA request specifying the entire tile size. If the tile size is chosen appropriately, the size of the DMA request will also be a multiple of 128 bytes. The Cell architecture is particularly efficient in serving DMA requests with size multiple of 128 bytes.

The use of large page sizes addresses the problem of TLB thrashing. We choose a page size of 16 MB. The use of large memory pages also reduces the number of page faults while converting the matrix from row-major layout to a block-major layout.

Tile Size A good choice of the tile size is very important to keep the SPE local store usage at an optimal level. The 256 KB local store in the SPE is shared between the code and the data. A tile size of 64×64 is chosen so that we have sufficient amount of buffers required for triple buffering (see below). This leads to a tile size of 16 KB for single-precision floating-point data elements. This is the largest size allowed by a DMA request. Such a tile size also helps overlap the communication with the computation and allows us to achieve close to peak performance for the critical step 3 in the above algorithm.

Double and Triple Buffering To overlap computation and communication, double and triple buffering is used. In double buffering, while processing one tile, another tile is brought into the local store simultaneously. Triple buffering is necessary if the tile which is being processed has to be written back to the main memory. Therefore, while a tile is being worked on, another one is read into the local store and the third is written back to the main memory. For example, the rank-k update performs a matrix-matrix multiplication as $A_{22} = A_{22} - L_{21}L_{21}^T$. Here consecutive tiles of A_{22} are read into the SPE local store, modified,

1	**2**	**3**	**4**	**1**	**2**
	3	**4**	**1**	**2**	**3**
		4	**1**	**2**	**3**
			4	**1**	**2**
				3	**4**
					1

Figure 2: Tile allocation to the SPEs for dense Cholesky factorization. The number on each tile refers to the SPE which factorizes the tile (assuming four active SPEs).

and stored back to main memory. In this case, triple buffering is used for the tiles of A_{22}, where three buffers are allocated in the SPE local store. While a tile is read into one of the buffers, a previously fetched tile is processed, and a processed tile in the third buffer is stored back to main memory, thereby forming a smooth pipeline. However double buffering with two buffers is enough for the tiles of L_{21}. Double and triple buffering almost completely hide the DMA latency, except for fetching and storing the first and the last tiles in the pipeline.

Optimization of the Kernels Special attention is given to optimizing all the three kernels, namely, $SPOTRF$, $STRSM$, $SSYRK$, and $SGEMM$. They are vectorized using the SPE vector intrinsics. The SPEs can issue two instructions every cycle, one of which is a floating-point instruction and the other one is a load/store instruction. Enough memory instructions and floating-point instructions should be provided so that both the load/store unit and the floating-point unit are kept busy. The loops are sufficiently unrolled to provide a long stretch of instructions without branches. Also, the loading of a data and its subsequent use are ensured to be far apart so that there are no pipeline stalls. The important $SGEMM$ kernel is derived from the optimized matrix multiplication example provided in [8].

Parallel Implementation The PPE is responsible for creating a single thread on each SPE, after which it waits for the SPEs to finish their work. Each SPE executes the same code but works on the set of tiles assigned to it (i.e. a typical single program multiple data or SPMD style parallelization). However, it needs to synchronize with other SPEs before and after the rank-k update step so that the threads do not read stale data. The barrier needed for this synchronization is implemented with the help of signals. The SPEs communicate with each other in a tree-like manner and the root of the tree initiates the release signal down the tree when all the threads have arrived. On a server with two Cell processors, the SPEs on the same chip synchronize among each other first and then the root of each node synchronizes among each other so that the number of slow inter-chip signals is minimized. Cholesky factorization only works on half of the symmetric matrix. This makes load balancing slightly tricky. The tiles are assigned to the SPEs in a cyclic manner as shown for four threads in Figure 2. This allocation ensures almost optimal load balancing among the SPEs. We chose to work on the upper triangle, since it makes vectorization of the kernels easier. At the end of the factorization, the L^T matrix is generated.

Multi-node NUMA Implementation The Cell BladeCenter QS20, on which the results are recorded, is a dual-Cell server with a NUMA architecture. Each node has faster access to its local memory than to remote memory on the other node. As access to memory from a remote node is slower, the factorization will fail to scale well beyond a single chip if pages are not mapped to the nodes in a NUMA-aware fashion. Before the factorization begins, the input, which is assumed to be in row-major layout in the first node's memory, is converted to tile-major layout. The tiles assigned to each node are placed in the corresponding local memory with the help of `numa_set_membind` API. This ensures that all the writes in each node are local. However, a node may need to read a remote tile for processing a local tile. Most of these remote reads happen during step 3 mentioned in the algorithm above, as performing $SGEMM$ on a tile requires two other tiles which may not be local. To make these reads local, all the tiles worked on during step 2 are replicated in both remote and local memory. This ensures that all reads during the critical step 3 are local. Note that the algorithm is such that these replicated data do not introduce any coherence issues. Finally, after the factorization is completed, the tiles are converted back to row-major layout and collated in a single contiguous range in the first node's main memory.

2.2 Dense PLU Factorization

A detailed description of LU decomposition with partial pivoting can be found in [3]. We implement a tiled (or blocked) algorithm for computing the PLU factorization of the $n \times n$ input matrix A. In the following, we discuss this algorithm.[5] Consider

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where A_{11} is $b \times b$, A_{12} is $b \times (n-b)$, A_{21} is $(n-b) \times b$, and A_{22} is $(n-b) \times (n-b)$. The steps of the tiled algorithm are enumerated below.

1. Apply the basic LU algorithm on the $n \times b$ panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ to obtain $P \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11}$.
2. Permute the entire matrix A using the permutation matrix P and overwrite the panel $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ with the factored panel.
3. Compute $U_{12} = L_{11}^{-1} A_{12}$ and update A_{12} with U_{12}.
4. Compute the rank-k update $A_{22} \leftarrow A_{22} - L_{21} U_{12}$.
5. Recurrently apply all the steps to A_{22}.

2.2.1 Implementation on the Cell Processor

In the following, we discuss the parallel implementation of blocked (or tiled) PLU on the Cell processor. This design significantly differs from the one presented in [26].

[5] The end-result of $PA = LU$ is the decomposition $A = P^{-1}LU = P^T LU$, since P is a permutation matrix and is orthogonal by definition. Note that P^T is also a permutation matrix.

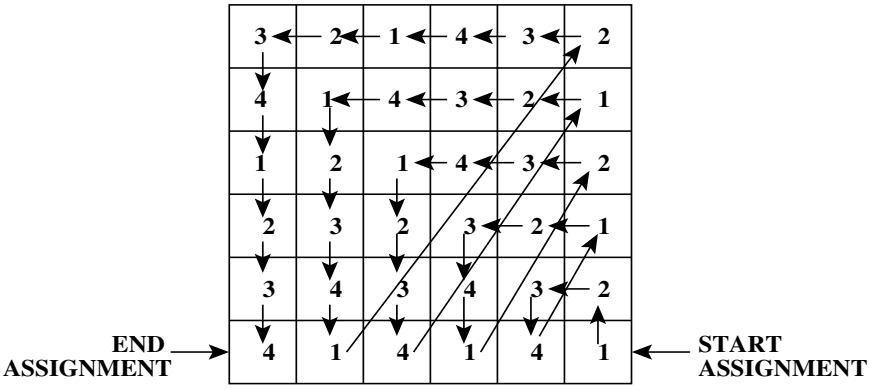


Figure 3: Tile assignment to the SPEs in LU decomposition. The number on each tile refers to the SPE which factorizes the tile (assuming four active SPEs). The tile assignment order is shown with arrows.

The matrix has to be split into small tiles so that the tiles belonging to the working set can fit in the SPE local store. LU decomposition, like Cholesky factorization, can be computed via tiled operations on smaller sub-matrices. The matrix is divided into tiles of size $b \times b$. The tile size is chosen to be 64×64 for similar reasons as dense Cholesky factorization.

The elementary single-precision kernels involved on the tiles are the following.

- $SGETF2(X)$: Computes the k^{th} iteration of the basic in-place LU factorization on tile X.
- $STRSM(X, U)$: Updates the tile X under a diagonal tile, upper triangle of which is U, during the k^{th} iteration of the basic algorithm: $X \leftarrow XU^{-1}$.
- $STRSM(X, L)$: Updates the tile X belonging to A_{12} using the lower triangle L of a diagonal tile as $X \leftarrow L^{-1}X$.
- $SGEMM(X, L, U)$: Updates X using the tiles L and U as $X \leftarrow X - LU$.

The PPE spawns one thread on each of the SPEs and after that it assists in maintaining the permutation array P. The parallelization has been done in SPMD style where each SPE executes the same code but works on different data. The steps in the algorithm discussed above have to be completed sequentially. So, parallelism is extracted within each step. Barriers are used for synchronizing the SPEs after each step.

The kernels have been optimized with techniques discussed earlier like vectorization, loop unrolling, and pipeline stall reduction with proper instruction placement. The PLU implementation uses tile-major storage and huge pages, as in dense Cholesky factorization.

Tile Assignment Figure 3 shows the tile assignment among the SPEs. The assignment policy is chosen with the following criteria in mind: (1) the work distribution among the active threads should be equal, and (2) a single thread should be made responsible for all the updates to a tile. The second criterion helps in multi-node environments, where tiles can be stored close to the owner node reducing remote memory operations. The tile assignment primarily tries to minimize the work imbalance during the rank-k updates in step 4 of the algorithm. Tile assignment is started from the tile at the bottom-right corner, which constitutes the single tile for the final rank-k update. Each row and column of tiles added for the next bigger sub-matrix gets assigned in a cyclic manner counter-clockwise. How the tiles are traversed by the assignment algorithm starting from the

bottom-right corner is shown using arrows in Figure 3. With this assignment, the work allocation for every rank-k update is evenly distributed. To find the ownership of a tile quickly, the owners of all the diagonal tiles are precomputed and stored. Knowing the owner of the diagonal tile T_{ii} is sufficient to find the owner of any tile T_{pq} for $p > i$ and $q > i$, as apparent from Figure 3.

Hiding DMA Latency Techniques of double and triple buffering are used to hide the DMA latencies. The BLAS-3 operations $STRSM(X, L)$ and $SGEMM(X, L, U)$ can almost completely hide the latency of the DMA operations. But the BLAS-2 routines, $STRSM(X, U)$ and $SGETF2$, which consist of $O(n^2)$ operations on $O(n^2)$ data do not have enough computation time to hide the DMA latency. To partially counter this problem, a total of eight buffers are allocated for panel factorization during step 1 of the tiled algorithm. During this step, the tiles of the panel need to be worked on repeatedly once for each iteration. If the number of tiles requiring $STRSM(X, U)$ and $SGETF2$ operations for an SPE is less than or equal to the number of buffers allocated, all the tiles can be kept in the local store for the subsequent iterations of the entire panel factorization. For example, during the first iteration on a 4096×4096 matrix with a tile size of 64×64, the number of tiles in the first panel is 64. If eight SPEs share the work, each SPE gets allocated eight tiles. Having eight buffers allows us to accommodate the entire panel in the local store. This eliminates the need for DMA operations for the complete duration of the LU factorization of the panel. Finally, when the number of columns being worked on within a tile during the $STRSM(X, U)$ operation is less than or equal to 32 (i.e. half of the number of columns in a tile), only half of the tile is brought by building a DMA list. Each entry in the DMA list brings half of a row within a 64×64 tile and this amounts to 128 bytes of data for single-precision matrices. We note that 128-byte DMA operations are particularly efficient.

Partial Pivoting Finding the pivots during the panel factorization in step 1 requires the SPEs to communicate after each iteration to compute the maximum pivot. The pivot row identified has to be swapped with the first row. The need for synchronization after each iteration and after each swap presents a significant bottleneck to achieving good performance. A permutation matrix is generated after the panel is factorized which has to be applied to the entire matrix. We divide the swap operations outside the panel into two phases, namely, first to the rows of the matrix on the right side of the panel and then to the rows on the left of the panel. The swap operations on the right side are overlapped with the $STRSM(X, L)$ operation. During the $STRSM(X, L)$ operation on the tile in column j, the rows of the tiles in column $j + 1$ are swapped. The swap operations on the left side of the panel are done after the rank-k update and without overlap with any computation. The PPE assists in maintaining the permutation array. It synchronizes with the SPEs using the mailbox.

Multi-node NUMA Implementation The use of NUMA-aware memory binding is similar to Cholesky factorization. While the input matrix is converted from row-major storage to tile-major storage, the tiles owned by a node are placed in the local memory of that node. This ensures that all the writes are local. The number of remote reads during the rank-k updates is minimized by writing the tiles to both local and remote memory during the $STRSM(X, L)$ operations, which precedes the rank-k updates. At the end of the factorization, the tiles from remote and local memory are merged into a single contiguous address range in a row-major format.

2.3 Sparse Cholesky Factorization

Sparse Cholesky factorization represents a sparse symmetric positive semi-definite matrix A as LL^T. The following steps are required to compute L. Let L be initialized with the lower triangular part of the input matrix A.

- Reordering of sparse matrix: Some fill-in will occur in L while decomposing A and therefore, L will have less sparsity than A. The amount of computation depends on the amount of fill-in. Reordering algorithms (e.g., the minimum degree algorithm) reorder the rows and columns of the matrix A to reduce the density of L, thereby reducing the amount of computation.
- Symbolic factorization: After reordering the matrix A, the non-zero structure of the matrix L can be computed. This step determines the structure of L, the elements of which will be calculated in the next step.
- Numerical factorization: The actual non-zero values of L are computed in this step. This is the most time-consuming step.

Blocked (or tiled) algorithms are used to carry out the numerical factorization step efficiently because this step has a lot of BLAS-3 operations. We implement the blocked algorithm proposed in [32]. This algorithm, after symbolic factorization, divides the non-zero columns of the matrix L into $(p_1, p_2, \ldots, p_n)$, where each p_i is a set of non-zero columns in L taken in sequence from left to right. In a similar way, the non-zero rows of the matrix L are divided into $(q_1, q_2, \ldots, q_n)$, where q_i is a set of non-zero rows in L taken in sequence from top to bottom. The block L_{ij} refers to the set of non-zeros that fall within rows contained in q_i and columns contained in p_j. The non-zero rows and the non-zero columns of L are divided such that the generated blocks interact in a simple way. This is ensured by the supernode-guided global decomposition technique [32]. This technique divides the matrix such that the columns of L having similar sparsity are grouped together. This reduces the book-keeping overhead while performing the factorization. For more details on this decomposition technique, the readers are referred to [32]. The generated blocks L_{ij} are stored in compressed column format, since the data files are available in this format only. In the compressed column format, a matrix is represented using the following three arrays.

- NZ: A single precision array of size n_nz (i.e., the total number of non-zeros in the matrix), containing all the non-zero values of the matrix.
- ROW: An integer array of size n_nz, containing the row index of each entry in the NZ array.
- COL: An integer array of dimension $n + 1$ (where n is the total number of columns in the matrix). The integer in the i^{th} entry of this array specifies the NZ index of the first element in column i. The last element of the array is n_nz.

An array, $LCol$, of size n (i.e., the number of columns in matrix L) stores the mapping of non-zero column numbers to actual column numbers. For example, $LCol[r]$ is the actual column number of the r^{th} non-zero column. Thus, depending on the number of completely zero columns, the last few entries of $LCol$ may remain unused. If there is not a single non-zero column in the matrix, $LCol[r]$ holds the value r for all r. This single array is used by all the blocks as a representation of the non-zero column structure. Notice, however, that due to this coarse-grain representation, a block that contains part of the i^{th}

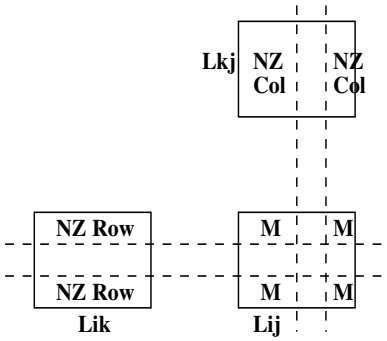


Figure 4: $BMOD$ operation: NZ Row represents the non-zero rows of L_{ik}, NZ Col represents the non-zero columns of L_{jk}, and M represents the updated sections of L_{ij}.

non-zero column may not actually have any of the non-zero entries of that column. Each block L_{ij} uses an array, $Structure_{ij}$, to store the non-zero row structure of that block. $Structure_{ij}[r]$ holds the row index of the r^{th} non-zero entry in the first column in that block. The block formation algorithm is such that all the columns of L_{ij} have the same non-zero structure. As a result, one $Structure_{ij}$ array per block L_{ij} is sufficient for representing the row non-zero structure of the entire block.

We have implemented the reordering and symbolic factorization steps from the SPLASH-2 suite [35]. The algorithms are modified slightly so that they work on large matrices. These two steps execute on the PPE. In the following, we present the details of the third step, which we have parallelized for the SPEs. SPEs are assumed to be arranged in a $p_r \times p_c$ grid. The basic kernels for sparse Cholesky factorization are shown below.

- $BFAC(L_{ik})$: Computes the Cholesky factorization of the block L_{ik}.
- $BDIV(L_{jk}, L_{kk})$: Updates a block L_{jk} as $L_{jk} \leftarrow L_{jk}L_{kk}^{-1}$.
- $BMOD(L_{ij}, L_{ik}, L_{jk})$: Updates the block L_{ij} as $L_{ij} \leftarrow L_{ij} - L_{ik}L_{jk}^{T}$. Since the product $L_{ik}L_{jk}^{T}$ can be of different density than block L_{ij}, only specific parts of L_{ij} are updated as shown in Figure 4. So, this kernel is completed in two steps, namely, computation of update ($Temp \leftarrow L_{ik}L_{jk}^{T}$) and scattering of update ($L_{ij} \leftarrow L_{ij} - NZMap(Temp)$). The function $NZMap$ does the following. The $(i_1, i_2, \ldots, i_s)$ rows of L_{ij} are updated, where $i_1, i_2, \ldots, i_s$ are the non-zero rows of block L_{ik}. Similarly, the $(j_1, j_2, .., j_t)$ columns of L_{ij} are updated, where $j_1, j_2, \ldots, j_t$ are the non-zero rows of block L_{jk}. The first stage of $BMOD$ is simple matrix multiplication with $O(n^3)$ operations on $O(n^2)$ data. The second stage has $O(n^2)$ operations on $O(n^2)$ data. Since the second stage has sparse operations, it cannot be efficiently vectorized, and suffers from a lot of stall cycles.

All the kernels operate on variable-sized sparse blocks. With each block, the number of $BMOD$ updates to be done on that block is also stored.

Algorithm 1 presents the details of this step. Separate queues are maintained for $BDIV$, $BMOD$, and $BFAC$ operations. The queue for $BFAC$ is referred to as the $TaskQueue$ in this discussion. Each non-zero column partition has one $BDIV$ queue and one $BMOD$ queue. Initially, the PPE, when converting the matrix into block layout (see next section), builds an array of size equal to the number of blocks. Each entry of this array corresponds to a block and stores two pieces of information, namely, the block owner and what operations have to be done on the block. After the threads are spawned on the SPEs, each

Algorithm 1 Blocked Algorithm for Sparse Cholesky Factorization

```
1: while all pending BMOD updates on all the blocks are not done do
2:    Receive block L_ik from TaskQueue.
3:    if L_ik is a diagonal block then
4:       BFAC(L_ik)
5:       for each block L_jk in BDIVQueue(k) do
6:          L_jk = L_jk L_kk^{-1}. Send L_jk to TaskQueue of processors in (j mod p_r) row and (j mod p_c) column of grid p_r × p_c.
7:       end for
8:    else
9:       Add L_ik to BMODQueue(k).
10:      for all L_jk in BMODQueue(k) do
11:         Locate L_ij.
12:         Compute L_ij = L_ij − L_ik L_jk^T. Decrement pending BMOD update count.
13:         if all BMOD updates on L_ij are done then
14:            if L_ij is a diagonal block then
15:               BFAC(L_ii).
16:               Send L_ii to TaskQueue of processors in (i mod p_r) row and (i mod p_c) column of grid p_r × p_c.
17:            else if L_ii is done then
18:               L_ij = L_ij L_ii^{-1}.
19:               Send L_ij to TaskQueue of processors in (i mod p_r) row and (i mod p_c) column of grid p_r × p_c.
20:            else
21:               add L_ij to BDIVQueue(j).
22:            end if
23:         end if
24:      end for
25:   end if
26: end while
```

SPE scans through this array and finds out its blocks. If a block L_{ii} is found to be ready for $BFAC$, the owner computes the factorization and sends the block information to the $TaskQueue$ of all other SPEs in row $(i \bmod p_r)$ and column $(i \bmod p_c)$ within the $p_r \times p_c$ grid of SPEs. This is how the $TaskQueue$ of an SPE gets populated. The algorithm clearly describes how the $BDIVQueue$ and $BMODQueue$ are populated and drained. The $BDIVQueue$ contains blocks to be operated on by $BDIV$. $BMODQueue(k)$ contains the blocks L_{*k}. For doing a $BMOD$ operation, an SPE scans this queue and picks the first two blocks L_{ik} and L_{jk} that it encounters provided the SPE is the owner of L_{ij}. The $TaskQueue$ contains the already factorized blocks.

2.3.1 Implementation on the Cell Processor

Block Layout and Huge Pages Matrix L is converted to block layout using the PPE. Extra padding is applied to the blocks to make them a multiple of 4×4. This ensures that the block size is always a multiple of 16 bytes. Also, with the block size being a multiple of 4×4, the kernels can be better optimized. If a block has a sparse nature, it also maintains an associated $Structure$ array (already discussed), which represents its non-zero row structure. Allocating both the block and its $Structure$ array in contiguous memory locations results in better data locality. As already explained, huge pages are used to reduce the volume of page faults.

Tile Size Because of the sparse structure of the matrix it is not possible to use a fixed tile size. Because of the limited local store area a tile is never made bigger than 48×48. Although for dense Cholesky factorization we use 64×64 tiles, in sparse

Cholesky this is too large to fit in the local store because of the additional memory requirements of the various task queues (e.g., $BMODQueue$, $BDIVQueue$, etc.).

Local Store Utilization Each block of data has an associated 128 bytes of metadata. Instead of storing the entire metadata in $BMODQueue$, $BDIVQueue$, or $TaskQueue$, minimal information (e.g., block id and location of the block) is stored and whenever a metadata information is needed it is transferred from main memory via DMA operations. We overlap the DMA latency using the concept of work-block lists described below.

Parallel Implementation Because of the sparse nature of the matrix, it is very difficult to achieve good load balance. The SPEs are assumed to be arranged in a $p_r \times p_c$ grid and the block L_{ij} is mapped on $< i \bmod p_r, j \bmod p_c >$. However, the work distribution gets skewed as the number of processors is increased. Also, the work distribution becomes uneven when odd number or prime number of processors are used.

Double and Triple Buffering Most of the work is done in step 12 of Algorithm 1. To do double and triple buffering for the blocks in this step, lists are used. Two work-block lists are created as described below.

- Updatee block list: contains the blocks L_{ik} and L_{jk}, which satisfy $map(L_{ij}) = \text{mySPEid}$. The function map takes i and j and returns the block's owner SPE by computing $< i \bmod p_r, j \bmod p_c >$, as already discussed.
- Destination block list: contains all the blocks L_{ij}, which will get updated by $L_{ik}L_{jk}^T$.

These lists are generated by scanning the $BMODQueue(k)$ and they contain only metadata. Algorithm 2 is used to do double and triple buffering.

Algorithm 2 Double and Triple Buffering Algorithm for Sparse Cholesky

1: **if** $ListSize > 1$ **then**
2: Get block data and $Structure$ of L_{ik}. Set $aid = id = 0$. // aid takes values 0, 1, 2 and id takes values 0 and 1.
3: $L_{jk}[id] = UpdateeBlockList[1]$. Get L_{jk}.
4: $L_{ij}[aid] = DestinationBlockList[1]$. Get L_{ij}.
5: $i = 1$
6: **while** $i < ListSize - 1$ **do**
7: $i = i + 1$
8: $L_{jk}[id \text{ xor } 1] = UpdateeBlockList[i + 1]$.
9: Get $L_{jk}[i + 1]$.
10: $L_{ij}[(aid + 1)\%3] = DestinationBlockList[i + 1]$.
11: Get $L_{ij}[i + 1]$.
12: Perform steps 12–23 of Algorithm 1 on $L_{ij}[aid]$ using $L_{jk}[id]$ and L_{ik}.
13: DMA diagonal block L_{ii} (if required) before performing $BMOD$.
14: $i = i + 1$
15: $id = id \text{ xor } 1$
16: $aid = (aid + 1)\%3$
17: **end while**
18: Perform steps 12–23 of Algorithm 1 on $L_{ij}[aid]$ using $L_{jk}[id]$ and L_{ik}.
19: DMA diagonal block L_{ii} (if required) before performing $BMOD$.
20: **end if**

Kernel Optimization The $BMOD$, $BDIV$, and $BFAC$ kernels are optimized for variable-sized blocks. As the kernels are generic and not specialized for every different block size, they suffer from performance loss because of branch stall cycles. Our $BMOD$ kernel delivers around 15.5 GFLOPS when it is executed with a 48×48 block size.

Multi-node NUMA Implementation Matrix L is replicated on each node, with each processor updating it using local reads and local writes. Blocks are written remotely only after $BDIV$ and $BFAC$ operations are performed on them. One task queue per SPE is maintained. Whenever an SPE completes a $BDIV$ or $BFAC$ operation on block L_{ij}, it writes the block id and its location to the task queues of the SPEs which own the blocks in the i^{th} block-row or the i^{th} block-column of matrix L. Note that these writes have to be atomic. We use the `mutex_lock` construct from the synchronization library provided with the Cell SDK for this purpose. Reads are synchronized with writes with the help of read/write pointer comparisons, which need not use any atomic instructions. Further, to reduce the number of remote reads, every SPE reads from a task queue in chunks (as opposed to reading one entry at a time).

2.4 QR Factorization

The QR decomposition of a real-valued $m \times n$ matrix A, with $m \geq n$ decomposes A as $A = QR$, where Q is an orthogonal matrix ($Q^T Q = I$) of size $m \times m$ and R is an upper triangular matrix of size $m \times n$. QR decomposition is more stable than LU decomposition, but takes more floating-point operations. There are many methods for computing the QR decomposition, such as Givens rotations, Gram-Schmidt orthogonalization, or the Householder reflections [14]. An algorithm using Householder transformation has been chosen for implementation in this paper.[6]

LAPACK [28] uses an algorithm which brings in a panel of the matrix (i.e. a set of columns) and computes its QR factorization accumulating the Householder reflectors of these columns. The accumulated Householder reflectors are then applied all at once using a BLAS-3 operation. However, this panel factorization algorithm is not very scalable because computing the QR factorization on the panel involves a lot of very memory intensive BLAS-2 operations. Also, this algorithm will not be very efficient to implement on the Cell processor because of the very limited size of the local store which would force the panels to be very small. A tiled version of sequential QR factorization is chosen to be parallelized in this paper. The sequential algorithm is explained in [7] and discussed in detail in an out-of-core context in [20]. In the tiled algorithm, the input matrix A is divided into small tiles of size $b \times b$, which fit into the local store. The following are the main operations in the tiled algorithm for single-precision matrices.

- $SGEQT2(A_{kk}, T_{kk})$: Computes the QR factorization of the diagonal tile A_{kk}. This operation overwrites the upper triangular part of the tile with R_{kk} and the lower triangular part with the b Householder reflectors Y_{kk}. It also generates a temporary upper triangular matrix T_{kk} containing the accumulated Householder reflectors.

- $SLARFB(A_{kj}, Y_{kk}, T_{kk})$: Applies the accumulated Householder reflectors in T_{kk} to the tile A_{kj} with $j > k$, reading Y_{kk} from the diagonal tile generated in $SGEQT2$. This operation updates the tiles to the right of the diagonal tile A_{kk}. The operation can be described as

[6] Concurrently with our effort, a LAPACK working note [27], discussing QR factorization on the Cell processor, was published.

$$A_{kj} \leftarrow (I - Y_{kk}T_{kk}Y_{kk}^T)A_{kj}\ \forall j > k.$$

- $STSQT2(A_{ik}, R_{kk}, T_{ik})$: Computes the QR factorization of the tiles A_{ik} below the diagonal tile (i.e. $i > k$). This routine updates R_{kk} generated in the diagonal tile by $SGEQT2$ with $\tilde{R}_{kk}$, updates the tile A_{ik} with the Householder reflectors Y_{ik}, and computes a temporary T which contains the accumulated Householder reflectors. The operation is summarized below.

$$\begin{pmatrix} R_{kk} \\ A_{ik} \end{pmatrix} \leftarrow \begin{pmatrix} \tilde{R}_{kk} \\ Y_{ik} \end{pmatrix}, T_{ik} \leftarrow T\ \forall i > k$$

- $SSSRFB(A_{ij}, A_{kj}, Y_{ik}, T_{ik})$: Applies the accumulated Householder reflections T_{ik} computed by $STSQT2$ to the tile A_{ij}. It also updates the tile A_{kj} computed by $SLARFB$, which is on the row of the diagonal tile A_{kk} and on the same column as tile A_{ij}. This operation is summarized below.

$$\begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix} \leftarrow \left(I - \begin{pmatrix} I \\ Y_{ik} \end{pmatrix} \cdot (T_{ik}) \cdot \left(IY_{ik}^T\right) \right) \begin{pmatrix} A_{kj} \\ A_{ij} \end{pmatrix}\ \forall i, j > k$$

The complete algorithm is shown in Algorithm 3. The input matrix A is assumed to be covered by $M \times N$ tiles, each of size $b \times b$.

Algorithm 3 Algorithm for Tiled QR Factorization

```
1: for k in 1 to N do
2:    SGEQT2(A_kk, T_kk)
3:    for j in k + 1 to N do
4:       SLARFB(A_kj, Y_kk, T_kk)
5:    end for
6:    for i in k + 1 to M do
7:       STSQT2(A_ik, R_kk, T_ik)
8:       for j in k + 1 to N do
9:          SSSRFB(A_ij, A_kj, Y_ik, T_ik)
10:      end for
11:   end for
12: end for
```

2.4.1 Implementation on the Cell Processor

The following discussion brings out the dependencies between various operations. Understanding these flow dependencies helps determine the task schedule and data decomposition (i.e. tile assignment to the SPEs) in the parallel implementation.

- All the operations in an iteration of k depend on the diagonal tile (A_{kk}) to be updated with $SGEQT2$. Other operations in the same iteration depend on the diagonal tile, either directly or indirectly. The $SLARFB$ operation on the tiles A_{k*} also depends on the diagonal tile A_{kk}.

- $STSQT2$ operation on A_{*k} has a column dependency. This dependency is not apparent in Algorithm 3. Recall that $STSQT2$ updates the upper triangular part R_{kk} of the diagonal block. These updates have to be done in sequence along

1	**2**	**3**	**4**	**1**	**2**
1	**2**	**3**	**4**	**1**	**2**
1	**2**	**3**	**4**	**1**	**2**
1	**2**	**3**	**4**	**1**	**2**
1	**2**	**3**	**4**	**1**	**2**
1	**2**	**3**	**4**	**1**	**2**

Figure 5: Tile assignment to the SPEs in QR factorization. The number on each tile refers to the SPE which factorizes the tile (assuming four active SPEs).

the column starting from the tile below the diagonal up to the last tile in the column. As a result, $STSQT2$ on A_{ik} cannot be started until $STSQT2$ on $A_{i-1,k}$ completes.

- $SSSRFB$ operation on A_{ij} depends on the $STSQT2$ operation on A_{ik} and the $SLARFB$ operation on A_{kj}. Further, each column A_{*j} has to be completed sequentially, i.e $SSSRFB$ on A_{ij} cannot start until $SSSRFB$ on $A_{i-1,j}$ completes. This dependency arises due to the fact that the $SSSRFB$ operation on A_{ij} for all $j > k$ updates the tile A_{kj} and unless all these updates are completed, $SSSRFB$ on $A_{i+1,j}$ cannot start updating A_{kj}.

These data dependencies prompted us to assign the tiles in a column cyclic manner as shown in Figure 5. The column dependencies mentioned above are satisfied without any synchronization as the tiles in a column belong to a single thread, and will be executed in the correct sequence within the thread.

Many of the optimization ideas discussed in the previous factorization techniques are used here as well. Block layout storage is used instead of row-major storage, huge pages of size 16 MB are used to reduce the volume of TLB misses, and double and triple buffering ideas are implemented to hide the DMA latencies. The core operations discussed above are implemented with optimized BLAS-1, BLAS-2, and BLAS-3 routines, which use loop unrolling, vectorization, and proper placement of loads and stores to reduce the amount of pipeline stalls. The following single-precision BLAS routines are used. BLAS-1: NORM2 (computes 2-norm); BLAS-2: GEMV (computes matrix-vector multiplication) and GER (computes rank-1 update); BLAS-3: GEMM (computes matrix-matrix multiplication). The matrix-matrix multiplication code is adopted from the Cell SDK distribution. It is modified to work on triangular matrices, as lot of BLAS-3 operations in QR factorization involve the triangular T and Y matrices.

The PPE spawns the threads on the SPEs and waits till the SPEs complete the factorization. A global task queue of pending tasks is maintained. The SPEs pick and execute tasks from it. A task is composed of a set of operations on a row. The operations are either $SLARFB$ on all the tiles owned by an SPE in a row or $SSSRFB$ on all the tiles owned by an SPE in a row. Along with computing the $SSSRFB$ operations, $SGEQT2$ or $STSQT2$ of the next iteration is computed if the first tile in the row is owned by the SPE which is computing the $SSSRFB$ operation. An entry is written into the task queue when either the $SGEQT2$ or the $STSQT2$ operation on a tile A_{ij} is completed. Each entry in the task queue is a pair (k, i), where k is the current outer iteration value (please refer to Algorithm 3) and i is the column which is just completed. The task queue

is read separately by the SPEs and each maintains a local pointer to the front of the queue. Busy waiting on the empty queue is avoided by using signals. If the queue is empty, each SPE waits on a signal register. When a new entry is added to the queue, a signal is sent to all the SPEs.

Multi-node NUMA Implementation The use of NUMA-aware memory binding is similar to the other dense matrix factorization routines discussed. However, we did not observe any advantage of using NUMA-specific memory binding for QR decomposition. The speedup numbers achieved with and without using NUMA binding are similar. This behavior can be explained by carefully examining the most commonly used two operations, namely, $SSSRFB$ and $SLARFB$. The $SSSRFB$ operation computes three matrix-matrix multiplications and the $SLARFB$ operation computes two matrix-matrix multiplications. This large amount of computation along with triple buffering provides enough slack to hide the DMA operations, even when communicating with remote memory.

2.5 Singular Value Decomposition

Given a matrix A of size $m \times n$ with $m \geq n$, singular value decomposition (SVD) of A is defined as $A = U\Sigma V^T$, where $U \in R^{m \times m}$ and $V \in R^{n \times n}$ are orthogonal matrices and Σ is a non-negative diagonal matrix. The columns of U and V are called the left and right singular vectors of A, respectively. The diagonal entries $\sigma_1, \sigma_2 \ldots, \sigma_n$ of the matrix Σ are called the singular values of A. Computation of SVD can be done in two ways, namely, iterative Jacobi method or via bidiagonalization. The fastest available iterative Jacobi algorithms [11] are still slower than the fastest algorithms based on bidiagonalization. In this paper, we have chosen an algorithm based on bidiagonalization to compute SVD. We focus on computing the singular values only and do not compute the orthogonal U and V matrices.

Bidiagonalization is a process of decomposing a matrix A of size $m \times n$ into $U_1 \tilde{B} V_1^T$, where U_1 and V_1 are orthogonal matrices. $\tilde{B}$ is an $m \times n$ matrix having non-zeroes only on the diagonal and one off-diagonal. The SVD of a bidiagonal matrix is defined as $\tilde{B} = U_2 \Sigma V_2^T$, where U_2 and V_2 are orthogonal matrices and Σ holds the singular values of matrix A. The singular vectors of matrix A are calculated as $U = U_1 U_2$ and $V = V_1 V_2$. Algorithms based on iterative QR or bidiagonal divide-and-conquer are used to compute the SVD of a bidiagonal matrix. We have used a bidiagonal divide-and-conquer algorithm [17] to compute the SVD of the bidiagonal matrix. The details of this algorithm are discussed after the bidiagonalization process is explained below. In the following discussion, we will concentrate on the $n \times n$ upper submatrix of $\tilde{B}$ only and refer to this submatrix as B. Note that the remaining $(m - n)$ rows of $\tilde{B}$ are all zero.

2.5.1 Bidiagonalization

Bidiagonalization consumes a significant fraction of the total time to compute SVD. It has a drawback of having a lot of matrix-vector multiplication operations. Matrix-vector multiplication requires $O(n^2)$ operations on $O(n^2)$ data, which results in poor computation-to-communication ratio. Recently, cache-efficient bidiagonalization algorithms are derived in [21], which reduce the communication requirement by half. This is achieved by reorganizing the sequence of operations such that two matrix-vector multiplications can be done at the same time. These are termed BLAS-2.5 operations. The two operations involved are $x \leftarrow \beta A^T u + z$ and $w \leftarrow \alpha A x$. We have chosen the tiled version of the Barlow's one-sided bidiagonalization algorithm [6] for

implementation. This algorithm also uses the BLAS-2.5 operations to reduce the data transfer requirement. Here we present a brief overview of the algorithm. Interested readers are referred to [21] for further information. The algorithm is sketched in Algorithm 4 in a MATLAB-like notation. We have omitted the step that computes the product of the Householder reflections. It can be implemented efficiently on the Cell processor using the tiled algorithm, as suggested in [7].

Algorithm 4 Barlow's Tiled One-sided Bidiagonalization Algorithm

1: b: Tile size, $s_1 = A(:,1)$
2: **for** $j = 1$ to $(n-2)/b$ **do**
3: $X = 0_{m \times b}$
4: $W = 0_{n \times b}$
5: **for** $k = 1$ to b **do**
6: $l = (j-1) \times b + k$
{Computation of u_l}
7: **if** $k > 1$ **then**
8: $A(:,l) = A(:,l) - X(:,1:k-1)W(l,1:k-1)^T$
9: $s_l = A(:,l) - \phi_l u_{l-1}$
10: **end if**
11: $\psi_l = \| s_l \|, u_l = s_l / \psi_l$
{Computation of $z^{(1)}$}
12: **if** $k > 1$ **then**
13: $z^{(1)} = -W(l+1:n, 1:k-1)X(:,1:k-1)^T u_l$
14: **else**
15: $z^{(1)} = 0$
16: **end if**
17: $x^{(1)} = 0$
{Computation of $z, x^{(1)}$}
18: **for** $i = l+1$ to n **do**
19: $z(1:n-l-1) = z^{(1)} + A(:,l+1:n)^T u_l$
20: $x^{(1)} = x^{(1)} + A(:,l+1:n) z(1:n-l-1)$
21: **end for**
22: $[\phi_{l+1}, v, x^{(3)}] = householder2(z, x^{(1)}, A[:,l+1])$
{Computation of Householder reflectors $x^{(4)}, v$}
23: $x^{(4)} = x^{(3)} - X(:,1:k-1) * W(l+1:n, 1:k-1)^T v$
24: $W(l+1:n, k) = v$
25: $X(:,k) = x^{(4)}$
26: **end for**
{Matrix update}
27: $A(:, j*b+1:n) = A(:, j*b+1:n) - XW(j*b+1:n,:)^T$
28: $s_{j*b+1} = A(:, j*b+1) - \phi_{j*b+1} u_{j*b}$
29: **end for**

Algorithm 4 can be divided into five basic steps:

- Computation of u_l (BLAS-2: GEMV)
- Computation of $z^{(1)}$ (BLAS-2: two GEMVs)
- Computation of $z, x^{(1)}$ (BLAS-2.5)
- Computation of $v_i, x^{(4)}$ (BLAS-2: two GEMVs)
- Matrix update (BLAS-3: GEMM)

2.5.2 SVD of Bidiagonal Matrices

The bidiagonal divide-and-conquer algorithm [17] recursively divides B into two subproblems

$$B = \begin{pmatrix} B_1 & 0 \\ \alpha_k e_k & \beta_k e_1 \\ 0 & B_2 \end{pmatrix},$$

where B_1 and B_2 are respectively $(k-1) \times k$ and $(n-k) \times (n-k)$ sized upper bidiagonal matrices. The row vector e_k is the k^{th} unit vector in R^k and the row vector e_1 is the first unit vector in R^{n-k}. This divide step is applied recursively to the bidiagonal matrices until the subproblem size is too small (16×16 in our implementation). These subproblems are then solved by using the iterative QR algorithm [10].

Let $Q_1 D_1 W_1^T$ and $Q_2 D_2 W_2^T$ be the SVDs of B_1 and B_2, where Q_i and W_i are orthogonal matrices and D_i is a non-negative diagonal matrix. Let f_i and l_i be the first and the last columns of W_i for $i = 1, 2$. Then we can write

$$B = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} D_1 & 0 & 0 \\ \alpha_k l_1^T & \alpha_k \lambda_1 & \beta_k f_2^T \\ 0 & 0 & D_2 \end{pmatrix} \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix},$$

where λ_1 is the last element of l_1, and α_k, β_k are the diagonal and off-diagonal entries of the k^{th} row of the bidiagonal matrix B. Let $S \Sigma G$ be the SVD of the middle matrix. Then the SVD of B is $Q \Sigma W$, where $Q = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} S$ and $W = \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix} G$. Note that only the first (f) and the last (l) columns of W are required to compute the singular values of B. These can be computed by the following equations.

$$f^T = \begin{pmatrix} f_1^T & 0 \end{pmatrix} G, \; l = \begin{pmatrix} 0 & l_2^T \end{pmatrix} G$$

The SVD of the middle matrix can be computed by permuting the matrix in the following format. The deflation procedure described in [18] is applied to the middle matrix to generate M so that the singular values and the singular vectors are calculated with high accuracy. Let

$$M = \begin{pmatrix} z_1 & z_2 & \dots & z_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

As discussed in [22], let the SVD of M be $S \Sigma G$ with $0 < \sigma_1 < \sigma_2 < \ldots < \sigma_n$, where$\{\sigma_i\}_{i=1}^n$ are the singular values of M satisfying the interlacing property, i.e. $0 = d_1 < \sigma_1 < d_2 < \ldots < d_n < \sigma_n < d_n + \parallel z \parallel_2$, and the secular equations

$f(\sigma_i) = 1 + \Sigma_k \frac{z_k^2}{d_k^2 - \sigma_i^2} = 0$. Also, the singular vectors of M satisfy $s_i = (-1, \frac{d_2 z_2}{d_2^2 - \sigma_i^2}, \ldots, \frac{d_n z_n}{d_n^2 - \sigma_i^2})^T / \sqrt{1 + \Sigma_k \frac{(d_k z_k)^2}{(d_k^2 - \sigma_i^2)^2}}$, and $g_i = (\frac{z_1}{d_1^2 - \sigma_i^2}, \ldots, \frac{z_n}{d_n^2 - \sigma_i^2})^T / \sqrt{\Sigma_k \frac{z_k^2}{(d_k^2 - \sigma_i^2)^2}}$, where s_i and g_i are the columns of S and G, respectively. The root finder algorithm of [29] is used to solve the secular equations and approximate $\{\sigma_k\}_{k=1}^n$ as $\{\hat{\sigma}_k\}_{k=1}^n$. Once the singular values are determined, $\hat{z}_i$ is computed by the following equation [18].

$$|\hat{z}_i| = \sqrt{(\hat{\sigma_n}^2 - d_i^2) \prod_{k=1}^{i-1} \frac{(\hat{\sigma_k}^2 - d_i^2)}{(d_k^2 - d_i^2)} \prod_{k=i}^{n-1} \frac{(\hat{\sigma_k}^2 - d_i^2)}{(d_{k+1}^2 - d_i^2)}}$$

The sign of $\hat{z}_i$ is chosen from z_i. Finally, $\hat{z}_i$ is used to compute the singular vectors g_i and s_i, which, in turn, complete the computation of the much needed vectors f and l. Algorithm 5 summarizes the major steps of the divide-and-conquer procedure.

Algorithm 5 Divide-and-conquer Algorithm for Bidiagonal SVD

1: Procedure divide_and_conquer_SVD (input:B; output:f, l, Σ)
2: **if** size$(B) \leq 16 \times 16$ **then**
3: Perform iterative QR.
4: return f, l, Σ
5: **else**
6: Divide $B = \begin{pmatrix} B_1 & 0 \\ \alpha_k & \beta_k \\ 0 & B_2 \end{pmatrix}$, where α_k and β_k are the k^{th} diagonal and off-diagonal entries of B.
7: Call divide_and_conquer_SVD (input:B_1; output:f_1, l_1, Σ_1)
8: Call divide_and_conquer_SVD (input:B_2, output:f_2, l_2, Σ_2)
9: Form matrix $M = \begin{pmatrix} z_1 & z_2 & \cdots & z_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}$ by using the deflation technique mentioned in [18].
10: Calculate Σ, G, f, l.
11: return f, l, Σ
12: **end if**

2.5.3 Implementation on the Cell Processor

Block Layout, Huge Pages, and Kernel Optimization Matrix A is stored in column-major format. Since a significant amount of time in the bidiagonalization procedure is spent in calculating the vectors z and $x^{(1)}$, the memory layout of A is chosen such that the memory latency is reduced for this phase. However, having A in column-major format requires the matrix update phase of bidiagonalization to be done using DMA list operations, which increase the execution time of the matrix update step. Matrices X and W in the bidiagonalization routine are stored in block-major layout. Huge pages of size 16 MB are used to reduce the volume of TLB misses. A tile size of 64×64 is chosen to perform the update of the trail part of matrix A during bidiagonalization.

The BLAS-2.5 kernel, the matrix-vector multiplication kernel, and the matrix-matrix multiplication kernel are optimized using the techniques discussed in the previous sections. The major kernels of bidiagonal SVD include iterative QR, and the kernels for solving the secular equations. All these kernels are iterative in nature and are very difficult to vectorize. However, in the secular equation solver, the maximum amount of time is spent in computing the values of function $f(\sigma_i)$. Special attention

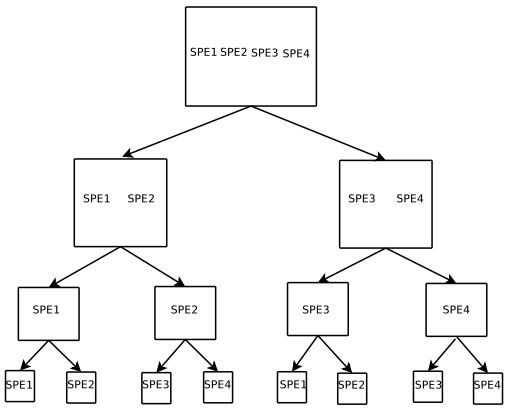


Figure 6: Work-block assignment strategy in bidiagonal SVD implementation (assuming four active SPEs).

is given to optimize the calculation of this function. Also, the kernels for calculating G, f, l, and $\hat{z}$ are carefully optimized.

Parallel Implementation First we discuss the parallelization of the bidiagonalization procedure. Matrix X is divided row-wise among the SPEs. The SPE i is allocated rows $[p_i, p_{i+1})$ where $p_i = i * (m/\text{total number of SPEs})$. Matrix W is also divided in the similar fashion. But after each iteration, the number of rows of W to be used in subsequent computation is reduced by one. To handle this, the work division for matrix W is redone row-wise after each iteration with total number of rows as $(n - l)$, where l is the iteration number. Special care is taken to ensure that each SPE has number of rows to work on always in multiples of four to have efficient DMA operations (four elements of a single-precision matrix take 16 bytes). Work division for panel $m \times b$ of matrix A is done row-wise. The trail part of matrix A is divided column-wise to compute the vectors $x^{(4)}$ and z. As this is the most compute intensive step, work division has to be almost equal among the SPEs for good speedup. Work division for the trail part of matrix A is redone after each iteration. The columns of the trail part of matrix A are divided equally among the SPEs to compute z and $x^{(1)}$. For the matrix update step, work division is done row-wise. However, for multi-node implementation, we have slightly changed the work division strategy to reduce the volume of remote writes. This will be explained later.

The divide-and-conquer bidiagonal SVD algorithm leads to a hierarchy of subproblems. The problems are organized in a tree structure as depicted in Figure 6. The leaves of the tree represent the smallest subproblems, which are solved using the iterative QR algorithm. All the internal nodes represent the merged problems obtained after the conquer step. To have dynamic work assignment, we build a list of work-blocks, where each node in the tree is represented by at least one work-block. The creation of the work-block list is done by the PPE. The PPE initializes a variable to the total number of work-blocks and each SPE atomically reads and decrements this variable when consuming a work-block. Each node of the tree has a $status$ variable, which represents the state of the node. For example, if an internal node has $status = 0$, it implies that none of the node's children have finished their computation. On the other hand, $status = 1$ implies that the children have completed computation and the node is ready for computation.

Having only one work-block per node increases the load imbalance because, as we work up the tree, problem size increases (due to the conquer step). Also, the number of nodes at the top levels decreases, resulting in load imbalance. To avoid this problem, toward the top of the tree we create more than one work-block per node using the formula: number of

work-blocks per node at level $k = \max$(total number of SPEs/total number of tree nodes in level $k, 1$). Figure 6 shows how this formula is applied. The set of work-blocks representing the same node are called the peer work-blocks and are numbered sequentially. The SPE that holds the first work-block among the peer work-blocks is termed the supervisor SPE of that group. Global synchronization between the SPEs computing on the peer work-blocks is done using the signals and $status$ variable. Each SPE in a peer group sends its sequence number to the supervisor SPE and goes to sleep. The supervisor SPE, on receiving signals from all the peer SPEs, increments the $status$ variable of that node in main memory and sends a signal to each of the peer SPEs. The SPEs operating on the same node compute f, l, and σ_i in parallel, but the matrix M is generated by the supervisor SPE sequentially. The time taken to generate matrix M is very small compared to the other operations.

Multi-node NUMA Implementation The salient features of the multi-node implementation pertain to the bidiagonalization step only. So, in the following discussion, we will focus on a few interesting aspects of multi-node bidiagonalization. Initially, matrix A is replicated on both the nodes. Matrices X and W are replicated as they are produced. This ensures that all the read operations are local. Matrix A is updated in two places in Algorithm 4:

- After the l^{th} iteration, the l^{th} column of matrix A is updated. This update of the column is performed in parallel. Each SPE writes its computed part of the column locally as well as remotely, which ensures that both the nodes have a coherent copy of A.
- We have changed the work division strategy in the update of the trail of matrix A to reduce the number of remote writes. The strategy that has been discussed till now requires 50% of the trail matrix to be written remotely for each matrix update call. To avoid this, the trail of matrix A is divided into two panels. Matrix updation to the first panel is divided among the SPEs of the first node and matrix updation to the second panel is divided among the SPEs of the second node. With this strategy, only $b + b/2 + 1$ columns of matrix A need to be written remotely per matrix update call as shown in Figure 7. This is explained in the following. The columns $l : l + b$ have to be replicated on both the nodes, since these columns will be divided row-wise among the SPEs while performing bidiagonalization on these columns. Also, the $(l + b + 1)^{\text{th}}$ column of matrix A needs to be replicated on both the nodes, as this column is needed to calculate $x^{(3)}$ for the $(l + b)^{\text{th}}$ column. In addition to this, $b/2$ columns computed on the second node also need to be written remotely, since these columns will be used by the SPEs of the first node for computation of the vectors z and $x^{(1)}$. Note that this work allocation strategy will work only for an even number of active SPEs. Finally, this strategy may not scale well beyond a small number of nodes, since it requires replicating data, which is done by broadcasting.

3 Evaluation Methodology

All the experiments are performed on a Cell BladeCenter QS20, which has two Cell B.E. processors operating at 3.2 GHz and a total of 1 GB XDR DRAM (512MB per processor). The system is running Fedora Core 6 with Cell SDK 2.1. The simulator provided with the SDK is used for preliminary debugging and performance analysis using instruction profiling before carrying out the production runs on the BladeCenter. Performance monitoring on the BladeCenter is done using calls to `spu_read_decrementer`, which reads the value of the decrementer (timer) register.

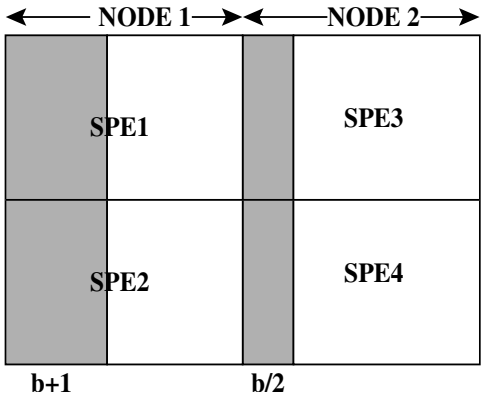


Figure 7: Panel distribution for two-node trail matrix update. The shaded areas represent the blocks that have to be written remotely. In this figure, for the purpose of illustration, we assume two active SPEs per node.

The `XLC` and the `GCC` compilers included in the SDK are used to compile the factorization library. Different compiler flags are used depending on whether the code is hand-optimized or not, and whether the final code size is to be optimized. Most of the hand-optimized kernels are compiled using the `-Os` (optimizes for code size) optimization flag with `XLC` or the `-O3` flag with `GCC`. More aggressive optimization flags are not used as they are found to increase the code size without significant performance improvements. For the other parts of the code that are not hand-optimized, the `-O5` optimization flag of `XLC` is used. The `qstrict` flag of `XLC` is used whenever better floating-point accuracy is desired.

3.1 Data Sets

The performance of dense Cholesky factorization, QR factorization, and LU decomposition is measured using synthetic sample matrices of different sizes. The LAPACK SLAMCH routine is used to generate the inputs for bidiagonalization and SVD. The BCS structural engineering matrices from the Harwell-Boeing collection [4] are used as the input data sets for sparse Cholesky factorization. The actual data sizes will be presented when discussing the results in the next section.

4 Experimental Results

4.1 Dense Cholesky Factorization

We present the speedup and GFLOPS delivered for dense Cholesky factorization in Figure 8. In the GFLOPS results, the number of floating-point operations are calculated as $\frac{1}{3}N^3$ for an $N \times N$ matrix. The results correspond to three different matrix sizes, namely, 4096×4096, 2048×2048, and 1024×1024. We present the results for both NUMA-optimized and NUMA-oblivious executions. The first eight threads are assigned to the first node and the second node is used only for measurements involving more than eight threads. We make the following important observations from these results. First, as expected, the scalability and delivered GFLOPS decrease with decreasing data size. Second, NUMA-aware data placement is very important for scaling beyond eight threads. Even for the largest matrix size of 4096×4096, the speedup levels out quickly around eight beyond eight SPEs. But the NUMA-optimized library gracefully scales beyond eight threads and delivers a speedup of 12.8 and 284.6 GFLOPS when sixteen threads are used. Finally, up to eight threads, the dense Cholesky

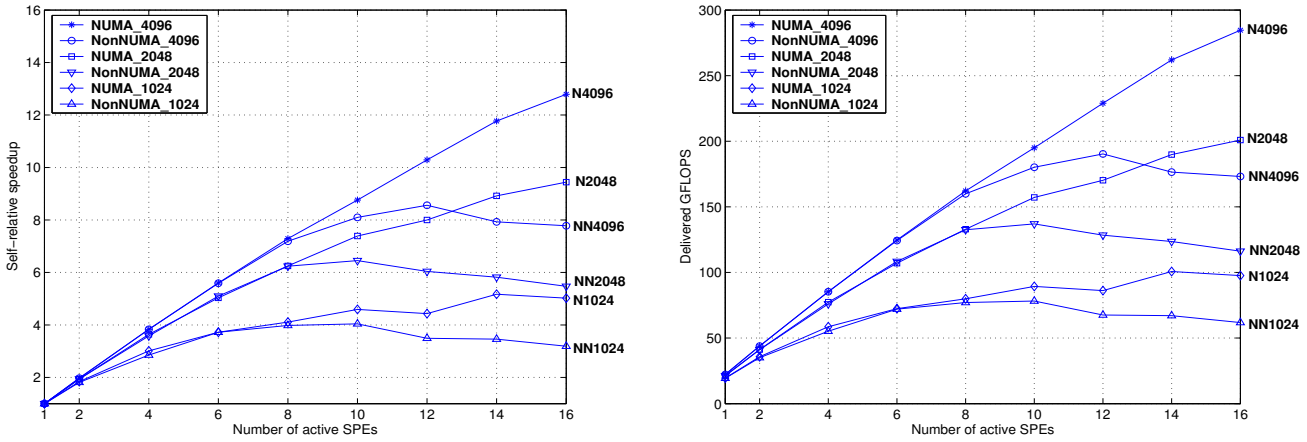


Figure 8: (a) Self-relative speedup and (b) delivered GFLOPS for dense Cholesky factorization. N4096 denotes the NUMA-optimized execution on 4096×4096 matrix. NN4096 denotes the Non-NUMA counterpart.

factorization library delivers almost linear speedup (7.3 on eight SPEs) and excellent GFLOPS (162.1 on eight SPEs) for the largest data size considered. Further, we would like to mention that if the time to convert the input matrix to tile-major layout is excluded from the measurement, the speedups achieved with the largest data set on eight and sixteen SPEs are 7.6 and 13.8, respectively. The corresponding GFLOPS numbers are 172.2 and 312.2. This result on eight SPEs closely matches with the one reported in [25].

Table 1: Average time spent in various activities as a percentage of the total average execution time for dense Cholesky factorization against different SPE counts running on a 4096×4096 matrix with NUMA-aware optimizations

Activities	1 SPE	2 SPEs	4 SPEs	8 SPEs	16 SPEs
$SSYRK$ & $SGEMM$	94.37%	93.16%	90.67%	86.22%	77.81%
$STRSM$	3.15%	3.11%	3.02%	2.86%	2.57%
DMA	0.42%	0.50%	0.85%	2.71%	6.59%
$SPOTRF$	0.10%	0.10%	0.10%	0.10%	0.10%
Barrier	–	0.21%	0.59%	1.86%	4.25%
Misc.	1.97%	2.93%	4.76%	6.25%	8.69%

To further understand the results, Table 1 presents a breakdown of execution time into various activities for 1, 2, 4, 8, and 16 threads running on the 4096×4096 matrix. The last row (Misc.) mostly includes the time to convert the row-major matrix into tile-major and vice versa, the time to find the next tile owned by a particular SPE, and the time to compute the owner of a tile. Therefore, some part of it is a constant overhead, independent of the number of SPEs (the percentage increases because the execution time decreases with the increasing number of SPEs). One can combine the figures in Table 1 and the speedup numbers from Figure 8(a) to get an idea about which components of the library are scaling well and which are not. It is clear that the major scalability bottlenecks come from the DMA and the barrier time, and, to some extent, the constant miscellaneous overhead. In fact, on sixteen threads, this constant overhead is the second most time consuming component, right after $SSYRK$ and $SGEMM$. One can apply Amdahl's Law to compute an upper bound on the achievable speedup

assuming that a fraction x of the total cycles is inherently sequential. Since on two SPEs the speedup is 1.97, we get the following equation involving x: $x + (1 - x)/2 = 1/1.97$. The solution to this equation gives us the number of cycles in dense Cholesky factorization that are inherently sequential. The value of x turns out to be 0.015228. These sequential cycles are contributed by part of the Misc. overhead and the DMA overhead. So, now we can apply Amdahl's Law to compute an upper bound on speedup for sixteen threads. This bound turns out to be 13.0. It is very encouraging to note that our achieved speedup of 12.8 is surprisingly close to this bound. We fail to achieve the bound exactly because the barrier time starts building up, as the number of SPEs increases.

4.2 Dense PLU Factorization

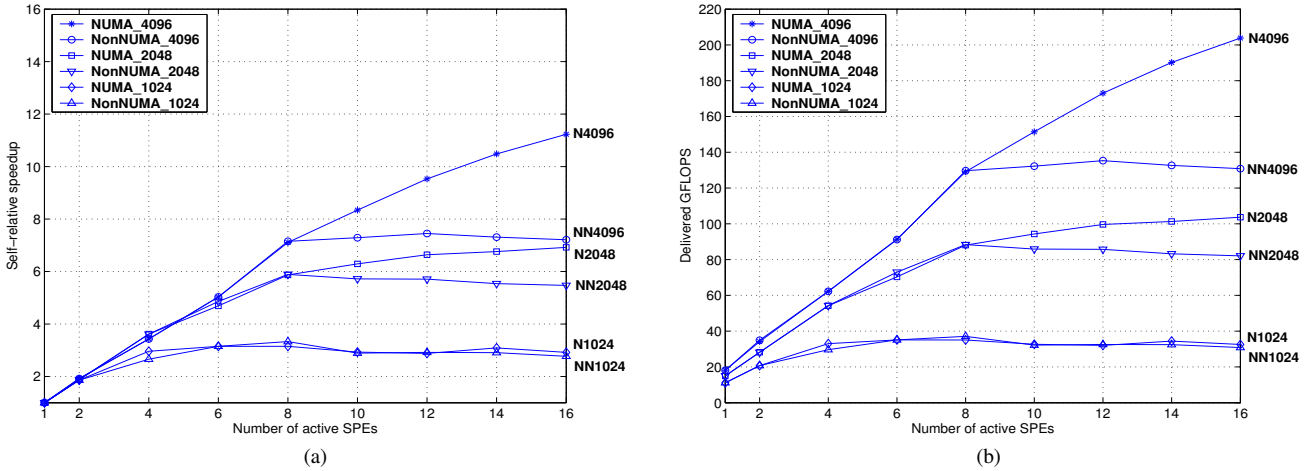


Figure 9: (a) Self-relative speedup and (b) delivered GFLOPS for PLU factorization. N4096 denotes the NUMA-optimized execution on 4096×4096 matrix. NN4096 denotes the Non-NUMA counterpart.

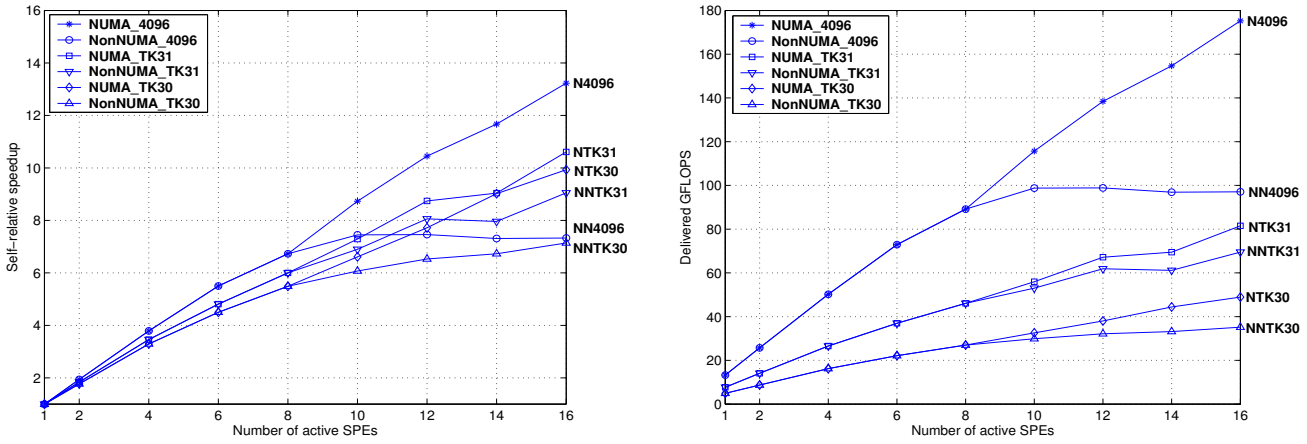
We present the speedup and GFLOPS results for PLU factorization in Figure 9. In the GFLOPS results, the number of floating-point operations are calculated as $\frac{2}{3}N^3$ for an $N \times N$ matrix. As in dense Cholesky, the results correspond to three different matrix sizes, namely, 4096×4096, 2048×2048, and 1024×1024. First, we note that PLU factorization is not as scalable as dense Cholesky factorization. For 4096×4096 matrix size, the achieved speedup on sixteen SPEs is 11.2 with NUMA-aware data placement. It delivers 203.9 GFLOPS. However, up to eight threads, the speedup numbers are similar for dense Cholesky and PLU factorization when using the largest matrix size. The limited performance of PLU factorization can be attributed to four primary reasons. First, the $STRSM(X, U)$ and $SGETF2$ operations of PLU factorization are mainly composed of memory-bound BLAS-2 routines. A BLAS-2 routine has $O(n^2)$ operations on $O(n^2)$ data. It requires three loads, one store, and one floating-point instruction per floating-point operation. This translates to at least four cycles per floating-point operation (the floating-point instruction itself can overlap with a memory instruction). With eight buffers in the local stores, we deliver GFLOPS much better than this, as already discussed. Second, finding the pivot row requires global synchronization after each iteration. The percentage cycle breakdown for the 4096×4096 matrix size presented in Table 2 clearly shows how the barrier overhead increases quickly with increasing number of SPEs. Third, due to a complicated tile-to-SPE assignment

algorithm (please refer to Figure 3), the cycles required to compute the owner of a tile or the next tile owned by a particular SPE are significant. These cycles are included in the miscellaneous cycles (last row of Table 2). The cycles required to do the row swaps are also included in Table 2. The swap operations on the rows to the right of a panel are overlapped with $STRSM(X, L)$, as already discussed. Although the swap operations on the rows to the left of a panel are exposed, these operations take a small percentage (2.6% to 3.2%) of the total execution time. Finally, we would like to mention that if the time to convert the input matrix to tile-major layout is excluded from the measurement, the speedups achieved with the largest data set on eight and sixteen SPEs are 7.3 and 11.7, respectively. The corresponding GFLOPS numbers are 134.68 and 215.72.

Table 2: Average time spent in various activities as a percentage of the total average execution time for PLU factorization against different SPE counts running on a 4096×4096 matrix with NUMA-aware optimizations

Activities	1 SPE	2 SPEs	4 SPEs	8 SPEs	16 SPEs
$SGEMM$	76.43%	73.52%	65.60%	68.36%	54.11%
$STRSM(X, U)$	9.75%	9.50%	8.70%	9.27%	7.75%
$STRSM(X, L)$ & Swap rows (right)	4.87%	4.66%	4.16%	4.24%	3.06%
Swap rows (left)	2.59%	2.55%	2.43%	2.99%	3.20%
DMA	1.11%	2.97%	10.84%	2.05%	8.63%
$SGETF2$	0.19%	0.36%	0.65%	1.34%	2.12%
Find max. pivot	0.04%	0.10%	0.28%	0.31%	0.65%
Barrier	–	0.74%	2.17%	4.26%	11.19%
Misc.	5.08%	5.67%	5.24%	7.18%	9.4%

4.3 Sparse Cholesky Factorization



(a) (b)

Figure 10: (a) Self-relative speedup and (b) delivered GFLOPS for sparse Cholesky factorization. N4096 denotes the NUMA-optimized execution on 4096×4096 matrix. NN4096 denotes the Non-NUMA counterpart.

We present the speedup and GFLOPS results for sparse Cholesky factorization in Figure 10. The results correspond to three different matrices. The TK30 data set refers to the BCSSTK30 sparse matrix of size 28924×28924 with 1036208 non-zero entries. This sparse matrix corresponds to the statics module of an off-shore generator platform. The TK31 data set

refers to the BCSSTK31 sparse matrix of size 35588×35588 with 608502 non-zero entries. This sparse matrix corresponds to the statics module of an automobile component. We also execute our sparse Cholesky factorization library on a dense matrix of size 4096×4096. This dense matrix experiment is important for comparing the load-balancing efficiency of the sparse Cholesky factorization algorithm with the dense Cholesky factorization algorithm in the limiting case. We observe that the speedup achieved by sparse Cholesky factorization on the dense matrix matches closely with that achieved by the dense Cholesky factorization. This is good news because from the viewpoint of the efficiency of parallelization, we can conclude that our implementation of parallel sparse Cholesky factorization is quite efficient. As expected, the scalability on the sparse matrices is worse. With sixteen threads, the speedup achieved on the TK31 data set is 10.6 when NUMA-aware data placement is enabled. Interestingly, the scalability gap between the NUMA-optimized and NUMA-oblivious execution for the dense matrix is much bigger than that for the sparse matrices. For example, with sixteen threads, the NUMA-optimized and NUMA-oblivious libraries respectively achieve speedup of 13.2 and 7.3 for the dense matrix. On the other hand, the speedups achieved by these two implementations for the TK31 data set are 10.6 and 9.1, respectively. The primary reason why even the non-NUMA version of sparse Cholesky factorization scales moderately well beyond eight threads is that the slow scatter-update operation used to update the tiles can overlap with the remote memory access latency.

Next, we turn to the GFLOPS figures. We find that sparse Cholesky factorization fails to deliver GFLOPS as high as the dense Cholesky or PLU factorization techniques. With sixteen threads, the NUMA-optimized library delivers only 81.5 GFLOPS for the TK31 data set. The primary reason for this miserably low utilization of resources is that a significant amount of time is spent in carrying out the scatter-update operation and scanning the $BDIVQueue$ and $BMODQueue$ to locate the next tiles to be updated and operated on at an SPE. In addition to these, depending on the sparse structure of the data set, the sparse Cholesky factorization library can suffer from severe load imbalance.

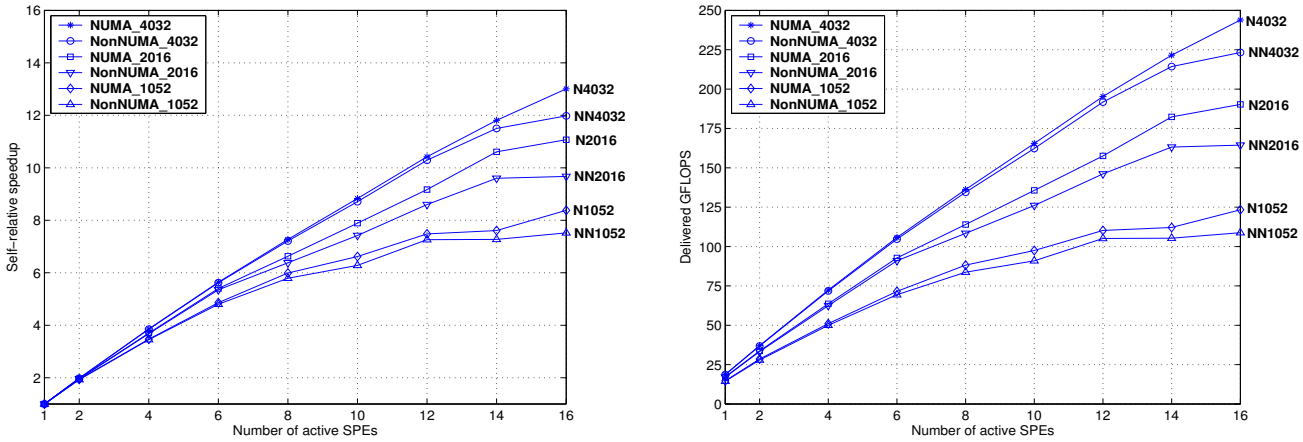
Table 3: Minimum and maximum time (in milliseconds) spent by a thread in various activities of sparse Cholesky factorization for different SPE counts running on TK31 with NUMA-aware optimizations

	1 SPE		2 SPEs		4 SPEs		8 SPEs		16 SPEs	
Activities	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.
$BMOD$	1555.58	1555.58	768.80	786.66	371.80	411.23	185.31	207.75	85.09	112.46
Scatter-update	1040.42	1040.42	513.85	526.58	249.99	275.32	116.06	147.33	53.78	79.12
Book-keeping	267.84	267.84	140.03	141.03	71.26	79.41	40.22	46.39	20.91	26.89
$BDIV$	236.61	236.61	117.51	119.11	57.09	60.40	28.50	30.34	13.63	15.84
DMA	86.61	86.61	53.41	53.65	32.76	33.23	23.44	26.13	14.92	24.65
$TaskQueue$	23.89	23.89	24.76	25.13	19.59	23.02	19.64	23.34	16.17	26.77
$BFAC$	12.47	12.47	6.20	6.27	0	6.27	0	3.14	0	3.14
Waiting time	0	0	55.39	80.47	29.56	72.62	39.72	69.09	27.46	61.67

To further get a feel about the severity of load imbalance, Table 3 presents the maximum and minimum execution time (in milliseconds) spent by a thread in various components of the sparse Cholesky factorization library for 1, 2, 4, 8, and 16 SPEs running on the TK31 data set. Note that for two different components, the min. threads or the max. threads may be different. As a result, adding up the min. (max.) thread's times in a column will not give the total execution time of the fastest (slowest) thread. But that information is not important for this discussion. The book-keeping cycles account for the time to scan the $BDIVQueue$ and $BMODQueue$ to locate the next tiles to be updated and operated on at an SPE. The $BMOD$ and the

scatter-update phases suffer from significant load imbalance. This essentially points to an inherently skewed work distribution arising from the irregular nature of the problem. The uneven distribution of the DMA cycles across the SPEs further supports this fact. Finally, the last row perhaps presents the most important information. It shows the minimum and the maximum waiting times across the SPEs arising from empty task queues. On sixteen SPEs, there is a large gap between the maximum waiting time and the minimum waiting time. Uneven load distribution is the primary reason for this.

4.4 QR Factorization



(a) (b)

Figure 11: (a) Self-relative speedup and (b) delivered GFLOPS for QR factorization. N4032 denotes the NUMA-optimized execution on 4032×4032 matrix. NN4032 denotes the Non-NUMA counterpart. The GFLOPS numbers correspond to the actual volume of floating-point operations of our algorithm i.e. $\frac{5}{3}N^3$. The LAPACK equivalent algorithm has a lower floating-point operation count i.e. $\frac{4}{3}N^3$.

We present the speedup and GFLOPS delivered for QR factorization in Figure 11. In the GFLOPS results, the number of floating-point operations are calculated as $\frac{5}{3}N^3$ for an $N \times N$ matrix. This is 25% more than the LAPACK equivalent algorithm having $\frac{4}{3}N^3$ floating-point operations. The results correspond to three different matrix sizes, namely, 4032×4032, 2016×2016, and 1052×1052. The matrix sizes are chosen such that they are close to a power of two and that the matrices can be perfectly covered by tiles of size 48×48. We observe that unlike the previous three libraries, QR factorization is not very sensitive to NUMA-specific optimizations. The speedup achieved by the largest data size on sixteen threads is 13.0 with NUMA-specific optimizations and 12.0 without these optimizations. The corresponding delivered GFLOPS numbers are 243.9 and 223.2, respectively. As already explained, the major portion of QR factorization involves five BLAS-3 operations and these computations are enough to hide remote memory latency on a two-node system. The primary performance bottlenecks in QR factorization are the $SGEQT2$ and $STSQT2$ routines, which are heavy with BLAS-2 operations having low computation-to-communication ratio.

Table 4 presents the percentage of execution time spent in various parts of QR factorization for the 4032×4032 matrix size running on 1, 2, 4, 8, and 16 SPEs. This table clearly shows that the management of the task queue presents a minor bottleneck. In general, we find that QR factorization is one of the most scalable factorization libraries among the ones considered in this

Table 4: Average time spent in various activities as a percentage of the total average execution time for QR factorization against different SPE counts running on a 4032×4032 matrix with NUMA-aware optimizations

Activities	1 SPE	2 SPEs	4 SPEs	8 SPEs	16 SPEs
$SSSRFB$	85.68%	84.02%	80.47%	78.50%	74.87%
$STSQT2$	8.52%	8.53%	8.52%	8.42%	8.23%
DMA	1.32%	1.41%	1.64%	2.48%	3.57%
$SLARFB$	1.01%	0.99%	0.94%	0.93%	0.91%
Task queue	0.15%	1.61%	4.38%	4.22%	4.28%
$SGEQT2$	0.14%	0.14%	0.14%	0.14%	0.13%
Misc.	3.18%	3.29%	3.91%	5.32%	8.01%

paper. We would like to note that if the time to convert the input matrix to tile-major layout is excluded from the measurement, the speedups achieved with the largest data set on eight and sixteen SPEs are 7.4 and 13.5, respectively. The corresponding GFLOPS numbers are 139.2 and 253.6.

4.5 Singular Value Decomposition

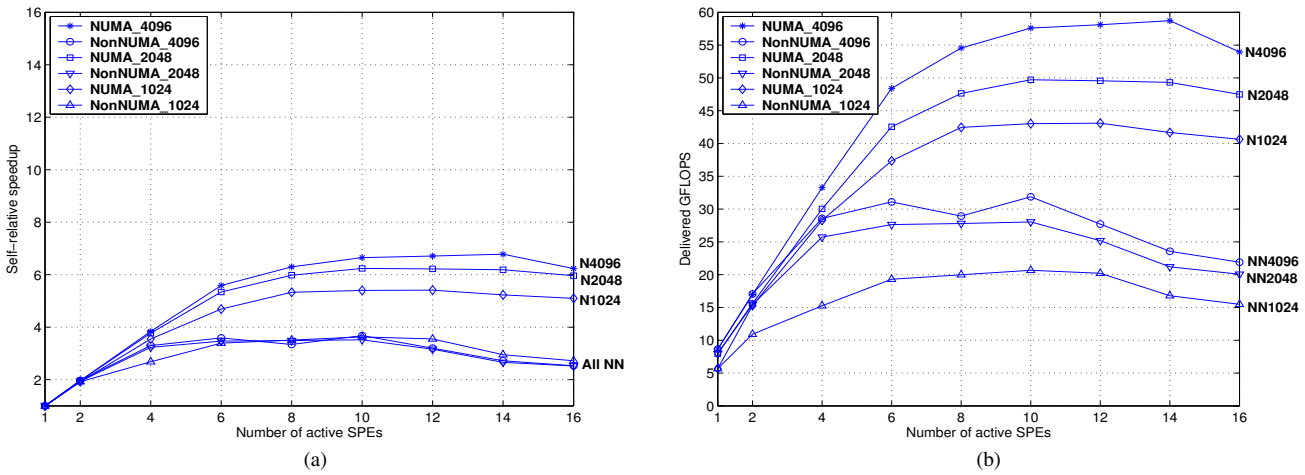


Figure 12: (a) Self-relative speedup and (b) delivered GFLOPS for SVD. N4096 denotes the NUMA-optimized execution on 4096×4096 matrix. All NN denotes the NUMA-oblivious executions.

Figure 12 presents the speedup and GFLOPS delivered for SVD. The results correspond to three different matrix sizes, namely, 4096×4096, 2048×2048, and 1024×1024. Note that in SVD, we start using both the nodes right from two threads. While running $2t$ threads, we spawn t threads on each of the nodes. This is important for obtaining good performance because, as explained below, SVD is limited by the DRAM bandwidth. So distributing the DRAM load on two nodes helps improve performance significantly. From the results, it is readily observed that SVD is the least scalable library among the ones considered in this paper. Even with the largest data set, the speedup achieved on sixteen threads is 6.2 with NUMA-aware work division. The corresponding GFLOPS delivered is only 54.0. In fact, the scalability is excellent up to six threads, beyond which the speedup quickly saturates. Note that the six-threaded SVD runs three threads on one node and three threads on the other. The data presented in Table 5 for the largest data set shows that the execution time of SVD is dominated by BLAS-2.5

operations. The BLAS-2.5 kernels operate at peak 7.35 GFLOPS and perform $4k$ floating-point operations on k floating-point data values.[7] Assuming single-precision floating-point data (our entire factorization library is single-precision), delivering peak BLAS-2.5 GFLOPS would require a DRAM bandwidth of 7.35 GB/s. Since the total DRAM bandwidth per node is 25.6 GB/s, this gets nearly saturated by three SPEs operating on BLAS-2.5 kernels concurrently. This is the reason why SVD scales almost linearly to three threads per node and beyond that the speedup levels out quickly as the threads start competing for the limited DRAM bandwidth. This effect is clearly visible in the data presented in Table 5. On eight and sixteen SPEs, the DMA time increases dramatically. The major portion of this DMA time beyond six threads gets spent in resolving bank conflicts and waiting in the memory request queue.

Table 5: Average time spent in various activities as a percentage of the total average execution time for SVD against different SPE counts running on a 4096×4096 matrix with NUMA-aware optimizations

Activities	1 SPE	2 SPEs	4 SPEs	8 SPEs	16 SPEs
BLAS-2.5 GEMV	72.83%	71.65%	70.18%	57.30%	28.17%
Trail matrix update	12.67%	12.43%	12.17%	9.97%	4.92%
DMA	4.45%	2.20%	2.53%	19.62%	57.05%
SVD conquer	3.47%	3.81%	4.56%	2.86%	1.56%
Other GEMV	2.58%	2.57%	2.58%	2.20%	1.17%
Iterative QR	0.22%	0.21%	0.21%	0.17%	0.09%
Barrier	–	2.82%	4.35%	2.04%	3.00%
Misc.	3.78%	4.32%	3.44%	5.85%	4.04%

5 Summary

We have implemented a parallel matrix factorization library on the Cell broadband engine. The library includes five popular factorization techniques, namely, dense Cholesky, LU with partial pivoting, sparse Cholesky, QR, and SVD. Our experimental results show that all the techniques except SVD show excellent scalability on a single Cell processor when using up to eight threads. For scaling beyond eight threads, we make use of two Cell processors housed in a BladeCenter. However, we find that without NUMA-aware data placement, all the factorization techniques except QR show poor scalability beyond eight threads. We present several NUMA-specific optimizations for each of the factorization techniques and show that all the techniques except SVD continue to exhibit excellent scalability even beyond eight threads. Parallelizing SVD was perhaps the most interesting and challenging experience. It is very different from the other four members of the library. This factorization technique suffers from limited DRAM bandwidth and fails to scale beyond three threads for a single Cell processor. Using multiple Cell processors helps distribute the DRAM load and offers improved speedup. For example, on a two-node BladeCenter, SVD scales up to six threads. Nonetheless, the NUMA-specific optimizations help SVD reach a speedup of around six before it levels out. At the time of this writing, we have limited understanding of the functioning and the protocols of the XDR RAMBUS DRAM module used by the Cell nodes. Some preliminary understanding of the 128-byte grain bank interleaving helped us identify the problem of using row-major layout for the tiled algorithms, as discussed in this paper. As a result, we

[7] Although a BLAS-2.5 operation involves two BLAS-2 operations, the peak GFLOPS is same as the aggregate of two BLAS-2 operations. This is because the two operations involved in a BLAS-2.5 operation are done separately. Although this does not deliver better GFLOPS, it does save communication time and EIB bandwidth because the matrix involved in these two operations is communicated only once over the EIB.

use tile-major layout for the entire library. Perhaps, an even deeper understanding of the DRAM module will help us optimize the SVD kernel further.

While implementing this factorization library we have learned a few important lessons and have prepared a wish-list for the future Cell platforms. First, we find that the size of the local store presents the biggest bottleneck to scalability. Although we achieve excellent speedup on four factorization kernels, many of them do not come even close to the peak deliverable GFLOPS of the machine. The factorization techniques that have less amount of BLAS-3 operations are the ones to deliver very poor GFLOPS. The simple reason for this is low computation-to-communication ratio of BLAS-2 or BLAS-2.5 operations. We believe that the only way to improve this performance is to offer bigger local store to each SPE so that much deeper buffering can be done leading to better hiding of the DMA latency. The end-result would be excellent prefetch performance. However, this may put more pressure on memory controller and demand more DRAM bandwidth. We discuss this aspect below.

Second, we feel that there is a lack of balance between the DRAM bandwidth and the peak aggregate GFLOPS promised by the Cell processor. While this is a standard problem for all high-end computing platforms, offering more DRAM bandwidth would be helpful. The problem of DRAM scheduling has gained importance in the architecture community over the last few years and perhaps better solutions will be in place for taking the full advantage of the offered DRAM bandwidth. However, following the philosophy of the Cell processor, it would be helpful to the programmer if some intrinsics could allow specifying a list of virtual page addresses (in the form of pointers to data structures) that may be accessed together. This hint can help the virtual memory management layer of the operating system in assigning appropriate physical addresses to these pages so that they do not suffer from DRAM bank conflicts. However, such a support has to come with a DRAM module that interleaves data across the banks at a much coarser grain than 128 bytes (e.g., DRAM pages in synchronous DRAMs). In summary, we believe that choosing the appropriate DRAM technology and complementing it with proper source-level intrinsics would be one of the key avenues to achieving high performance on the future Cell platforms.

Finally, as is the case for development on any real multiprocessor platform, debugging on the Cell BladeCenter often turns out to be tedious. At the time of this writing, we are not aware of much debugging facilities on the BladeCenter. Specifically, we feel that user-level debugging support for tracing DMA events would improve the productivity enormously. In the simulator distributed with the SDK, users can generate a dump of the memory flow controller queue and examine the DMA events. A similar support on the real machine would be helpful.

Acknowledgments

We thank Shakti Kapoor and Manish Gupta for helping us get access to a Cell Blade server through the Virtual Loaner Program (VLP). We thank all the VLP staff, especially Jennifer Turner and Rick Eisenmann, for promptly responding to our queries. Special thanks go to Manish Gupta and Sriram Vajapeyam for interesting discussions. We thank Yogish Sabharwal for helping us understand certain performance anomalies. We thank Rahul Garg who was instrumental in initiating the early efforts and connecting us with IBM research. We thank Riyaz Shiraguppi and Dipen Rughwani for early efforts on the Cell implementation of sparse BLAS-2 at IIT Kanpur. The experience and insight gathered during that work helped us greatly. Mainak wishes to thank the entire IBM community for supporting this work through an IBM faculty award. Finally, we thank the anonymous reviewers for constructive suggestions and Fred Gustavson for helping us put together Section 1.2.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Exploiting Functional Parallelism of POWER2 to Design High-performance Numerical Algorithms. In *IBM Journal of Research and Development*, **38**(5): 563–576, 1994.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-performance Matrix Multiply Algorithm on a Distributed Memory Parallel Memory Computer using Overlapped Communication. In *IBM Journal of Research and Development*, **38**(6): 673–681, 1994.

[3] G. S. Baker et al. PLAPACK: High Performance through High-level Abstraction. In *Proceedings of the 27th International Conference on Parallel Processing*, pages 414–423, August 1998.

[4] BCSSTRUC5: BCS Structural Engineering Matrices. *http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc5/bcsstruc5.html*.

[5] R. E. Bixby. Solving Real-world Linear Programs: A Decade and More of Progress. In *Operations Research*, **50**(1): 3–15, January/February 2002.

[6] N. Bosner and J. L. Barlow. Block and Parallel Versions of One-Sided Bidiagonalization. In *SIAM Journal of Matrix Analysis and Applications*, **29**(3): 927–953, October 2007.

[7] A. Buttari et al. Parallel Tiled QR Factorization for Multicore Architectures. *LAPACK Working Note 190*, July 2007. Available at *http://www.netlib.org/lapack/lawnspdf/lawn190.pdf*.

[8] Cell SDK Code Sample. Available at *http://www.power.org/resources/devcorner/cellcorner/codesample1*.

[9] E. Chan et al. SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks. *FLAME Working Note 25*, Department of Computer Sciences, University of Texas at Austin, Technical Report TR-07-41, August 2007.

[10] J. Demmel et al. Computing the Singular Value Decomposition with High Relative Accuracy. *Technical Report UCB/CSD-97-934*, University of California, Berkeley, February 1997. Available at *http://www.eecs.berkeley.edu/Pubs/TechRpts/1997/5373.html*.

[11] Z. Drmač and K. Veselić. New Fast and Accurate Jacobi SVD Algorithm I. In *SIAM Journal on Matrix Analysis and Applications*, **29**(4): 1322–1342, November 2007.

[12] E. Elmroth and F. G. Gustavson. Applying Recursion to Serial and Parallel QR Factorization Leads to Better Performance. In *IBM Journal of Research and Development*, **44**(4): 605–624, July 2000.

[13] E. Elmroth et al. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. In *SIAM Review*, **46**(1): 3–45, 2004.

[14] G. H. Golub and C. F. Van Loan. Matrix Computations. *The Johns Hopkins University Press*, 3rd edition, October 1996.

[15] M. Gschwind et al. Synergistic Processing in Cell's Multicore Architecture. In *IEEE Micro*, **26**(2): 10–24, March/April 2006.

[16] M. Gschwind. Chip Multiprocessing and the Cell Broadband Engine. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 1–8, May 2006.

[17] M. Gu, J. W. Demmel, and I. Dhillon. Efficient Computation of the Singular Value Decomposition with Applications to Least Squares Problems. *Technical Report LBL-36201*, Lawrence Berkeley National Laboratory, October 1994.

[18] M. Gu and S. C. Eisenstat. A Divide-and-Conquer Algorithm for the Bidiagonal SVD. In *SIAM Journal on Matrix Analysis and Applications*, **16**(1): 79–92, January 1995.

[19] F. G. Gustavson. New Generalized Data Structures for Matrices Lead to a Variety of High-performance Algorithms. In *Architecture of Scientific Software*, pages 211–234, October 2000.

[20] B. C. Gunter and R. A. Van de Geijn. Parallel Out-of-core Computation and Updating of the QR Factorization. In *ACM Transactions on Mathematical Software*, **31**(1): 60–78, March 2005.

[21] G. W. Howell et al. Cache Efficient Bidiagonalization using BLAS 2.5 Operators. *MIMS eprint 2006.56*, University of Manchester, April 2006. Available at *http://eprints.ma.man.ac.uk/210/*.

[22] E. R. Jessup and D. C. Sorensen. A Parallel Algorithm for Computing the Singular Value Decomposition of a Matrix. In *SIAM J. Matrix Anal. Appl.*, pages 530–548, 1994.

[23] J. A. Kahle et al. Introduction to the Cell Multiprocessor. In *IBM Journal of Research and Development*, **49**(4): 589–604, July 2005.

[24] S. Kumar. Parallel Computing in the Mainstream. In *Where Will All the Threads Come From?*, Panel in the 13th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, February 2008. Slides available at *http://research.ihost.com/ppopp08/presentations/PPoPP08-Panel-WhereThreads.pdf*.

[25] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving Systems of Linear Equations on the Cell Processor using Cholesky Factorization. *LAPACK Working Note 184*, May 2007. Available at *http://www.netlib.org/lapack/lawnspdf/lawn184.pdf*.

[26] J. Kurzak and J. J. Dongarra. Implementation of Mixed Precision in Solving Systems of Linear Equations on the CELL Processor. In *Concurrency and Computation: Practice & Experience*, **19**(10): 1371–1385, July 2007.

[27] J. Kurzak and J. J. Dongarra. QR Factorization for the Cell Processor. *LAPACK Working Note 201*, May 2008. Available at *http://www.netlib.org/lapack/lawnspdf/lawn201.pdf*.

[28] Linear Algebra Package (LAPACK). *http://www.netlib.org/lapack/*.

[29] R. Li. Solving Secular Equations Stably and Efficiently. *Technical Report: CSD-94-851*, University of California at Berkeley, 1992.

[30] G. Quintana-Orti et al. Scheduling of QR Factorization Algorithms on SMP and Multi-core Architectures. *FLAME Working Note 24*, Department of Computer Sciences, University of Texas at Austin, Technical Report TR-07-37, July 2007.

[31] G. Quintana-Orti et al. Design and Scheduling of an Algorithm-by-Blocks for LU Factorization on Multithreaded Architectures. *FLAME Working Note 26*, Department of Computer Sciences, University of Texas at Austin, Technical Report TR-07-50, September 2007.

[32] E. Rothberg and A. Gupta. An Efficient Block-oriented Approach to Parallel Sparse Cholesky Factorization. In *SIAM Journal on Scientific Computing*, **15**(6): 1413–1439, November 1994.

[33] Scalable Linear Algebra Package (ScaLAPACK). *http://www.netlib.org/scalapack/scalapack_home.html*.

[34] Watson Sparse Matrix Package. Available at *http://www-users.cs.umn.edu/∼agupta/wsmp.html*.

[35] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.