# Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures [*]

Fengguang Song
University of Tennessee
EECS Department
Knoxville, TN, USA
song@eecs.utk.edu

Stanimire Tomov
University of Tennessee
EECS Department
Knoxville, TN, USA
tomov@eecs.utk.edu

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester
dongarra@eecs.utk.edu

## ABSTRACT

We present a new methodology for utilizing all CPU cores and all GPUs on a heterogeneous multicore and multi-GPU system to support matrix computations efficiently. Our approach is able to achieve four objectives: a high degree of parallelism, minimized synchronization, minimized communication, and load balancing. Our main idea is to treat the heterogeneous system as a distributed-memory machine, and to use a heterogeneous 1-D block cyclic distribution to allocate data to the host system and GPUs to minimize communication. We have developed heterogeneous rectangular-tile algorithms with two different tile sizes (one for CPU cores and the other for GPUs) to cope with processor heterogeneity. We also propose an auto-tuning method to determine the best tile sizes to attain both high performance and load balancing. We have implemented a new runtime system and applied it to the rectangular tile Cholesky and QR factorizations. Our experiments on a compute node with two Intel Westmere hexa-core CPUs and three Nvidia Fermi GPUs demonstrate the weak scalability, strong scalability, load balance, and efficiency of our approach.

## 1. INTRODUCTION

As the performance of both multicore CPU and GPU continues to scale at a Moore's law rate, it is becoming appealing and pervasive to use heterogeneous multicore and multi-GPU architectures to attain the highest performance possible from a single compute node. Today it is not uncommon to find a shared-memory machine with dozens of cores and a few GPUs that can achieve a maximum performance of more than 2.5 Teraflops using double precision floating point arithmetic. However, the heterogeneity in the multi-core and multi-GPU architecture has introduced new challenges to algorithm design and software systems.

Over the last few years, our colleagues at the University of Tennessee have developed the PLASMA library [2] to solve linear algebra problems on multicore architectures. In parallel with PLASMA, we have also developed another library called MAGMA [27] to solve linear algebra problems on GPUs. While PLASMA and MAGMA aim to provide the same routines as LAPACK [4], one is used for multicore CPUs, and the other for a single core with an attached GPU, respectively. Our goal is to utilize all cores and all GPUs efficiently on a single multicore and multi-GPU system to support matrix computations.

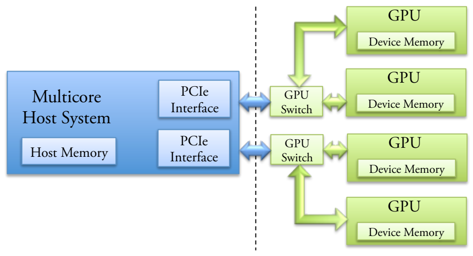


**Figure 1: The architecture of a heterogeneous multicore and multi-GPU system.** The host system is connected to four GPUs via two PCI express connections. The host system and each GPU have separate memory spaces.

Figure 1 shows the architecture of a heterogeneous multicore and multi-GPU system we are considering. The multicore host system is connected to four GPUs via two PCI express connections and each pair of GPUs share a GPU switch. The system is different from both shared-memory and distributed-memory systems due to the following features: (1) The system is not a shared-memory machine since the host and the GPUs have different memory spaces and an explicit memory copy is required to transfer data between the host and a GPU; (2) It is also different from a conventional distributed-memory machine since each GPU is actually controlled by a thread running on the host (more like pthreads on a shared-memory machine); (3) GPUs and CPU cores have distinct computational performance; (4) GPU is optimized for throughput and expects larger input to produce high performance than a core which is optimized for latency [24]; (5) The system requires two different computational libraries: one is developed and optimized for CPUs, the other for GPUs. In this work, we take into account all these factors and strive to meet four objectives in order to obtain high performance: a high degree of parallelism, minimized synchronization, minimized communication, and load balancing. We propose to design new heterogeneous algorithms and to use a simple but practical static data distribution to achieve the objectives simultaneously.

This paper describes rectangular tile algorithms with hybrid tile sizes, heterogeneous 1-D column block cyclic data distribution, a new runtime system, and an auto-tuning method to determine the hybrid tile sizes. The rectangular tile algorithms build upon the previous tile algorithms, which divide a matrix into square tiles and have a high degree of parallelism and minimized synchronizations [13, 14].

[*]This material is based upon work supported by the NSF grants CCF-0811642, OCI-0910735, by the DOE grant DE-FC02-06ER25761, and by Microsoft Research.

(Section 2.1 introduces the tile algorithms briefly). However, a unique tile size does not work well for both CPU cores and GPUs at the same time (either too small or too big). A big tile will clobber a CPU core and a small tile cannot attain high performance on a GPU. Therefore, we have redesigned the tile algorithms so that they comprise between two types of tiles: smaller tiles suitable for CPU cores and bigger tiles suitable for GPUs. For instance, Fig. 2 depicts two matrices consisting of a set of small and large rectangular tiles. The rectangular tile algorithms execute in a fashion similar to the tile algorithms such that whenever a task computing a tile at $[I,\ J]$ is completed, it will trigger new tasks on the right hand side of the $J$-th tile column and below the $I$-th tile row. Here the rectangle tile at $[I,\ J]$ can be either small or big and is different from the tile algorithms.

We regard the multicore and multi-GPU system as a distributed memory machine and place greater emphasis on communication minimization. We statically store small rectangular tiles on the host and large rectangular tiles on the GPUs respectively to cope with processor heterogeneity and reduce data movement. In order to distribute the small and big rectangular tiles to the host and GPUs evenly, we propose a heterogeneous 1-D column block cyclic distribution method. The basic idea is that we first map a matrix to only GPUs using a 1-D column block cyclic distribution, then we cut a slice from each block and assign it to the host system. Our analysis shows that the static distribution method is able to reach a near lower bound communication volume. We also propose an auto-tuning method to determine the best slice size to be cut from each block for load balancing.

We have designed a runtime system to support dynamic scheduling on the heterogeneous CPU+GPU system. The runtime system allows our programs to be executed in a data-availability-driven model where a parent task always tries to triggers its children. In order to address the specialties of the heterogeneous system, we have implemented a number of techniques to extend a centralized runtime system to a new one that also considers the machine a distributed memory system. The new runtime system is "hybrid" in the sense that its scheduling and computing components are centralized and resident in the single host system but data, pools of buffers, the communication components, and task queues are distributed in the host and different GPUs.

We conducted experiments on the Keeneland system at the Oak Ridge National Laboratory. On a compute node with two Intel Westmere hexa-core CPUs and three Nvidia Fermi GPUs, both of our Cholesky factorization and QR factorization exhibit scalable performance. In terms of weak scalability, we can attain a nearly constant Gflops/core and Gflops/GPU performance from 1 core to 9 cores+3 GPUs. And in strong scalability, we can reduce the execution time by two orders of magnitude from 1 core to 9 cores+3 GPUs.

To our best knowledge, this is the first work to consider the multicore and multi-GPU system a distributed-memory machine to minimize communication in support of matrix computations. Our work makes the following contributions: (**i**) new heterogeneous rectangular-tile algorithms with hybrid tiles to handle processor heterogeneity, (**ii**) a heterogeneous 1-D block cyclic distribution with a novel two-level partitioning scheme, (**iii**) an auto-tuning method to achieve load balancing, (**iv**) and a new runtime system to accommodate the special features of the heterogeneous system (i.e., a hybrid of a shared- and distributed-memory system).

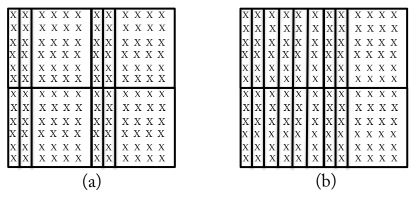


**Figure 2: Matrices consisting of a mix of small and large rectangular tiles.** **(a)** A $12 \times 12$ matrix is divided into 8 small tiles and 4 big tiles. **(b)** A $12 \times 12$ matrix is divided into 16 small tiles and 2 big tiles.

In the rest of this paper, Section 2 provides the background and motivations for our work. Section 3 presents the rectangular tile algorithms for Cholesky and QR factorizations. Section 4 describes the implementation and the auto-tuning method. Section 5 shows the experimental results. Section 6 presents related work and Section 7 summarizes our work.

## 2. BACKGROUND

In this section, we first give a brief introduction to the previous tile algorithms [14]. Then we describe the motivations for our optimizations on GPUs and the reasons for choosing a static distribution method over a dynamic load balancing method.

### 2.1 Tile Algorithms

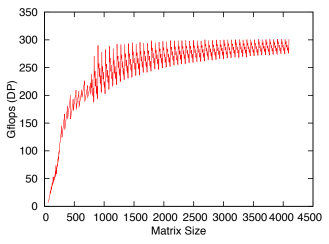
A tile algorithm divides an $n \times n$ matrix $A$ into a number of small $b \times b$ submatrices (aka "tiles") such that $A$ consists of $n_b \times n_b$ tiles, where $n_b = \frac{n}{b}$. $A$ can be expressed as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \dots & A_{n_b,n_b} \end{pmatrix},$$

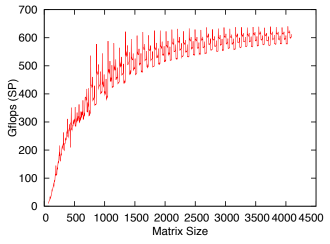
where $A_{i,j}$ is a $b \times b$ tile. In the tile algorithm, every task works on a small tile so that at any time there are a great amount of tasks available to execute. This way we can increase a program's thread level parallelism, which is desirable on multicore architectures. Take the tile QR factorization as an example. At the first iteration, the algorithm computes a QR factorization for $A_{1,1}$. The output of $A_{1,1}$ is then used to update the set of tiles on $A_{1,1}$'s right hand side in an embarrassingly parallel way (i.e., $A_{1,2}, A_{1,3}, \dots, A_{1,n_b}$). As soon as a tile-update in the $i$-th row completes, its below neighbor in the $(i+1)$-th row can start immediately. One could visualize the execution as falling columns of dominos from top to bottom. After updating all tiles in the $n_b$-th row, tile QR will continue to apply the same steps to the trailing submatrix $A_{2:n_b,2:n_b}$ recursively [14]. Our work extends the tile algorithms to rectangular tile algorithms to accommodate the processor heterogeneity between CPUs and GPUs.

### 2.2 Optimization Issues on GPUs

Although computational performance can be increased by adding more cores to a GPU, it is much more difficult to increase the network performance at the same rate. For instance, it has taken three years to introduce the PCIe 3.0 Base specification to double its predecessor's bandwidth. We also expect the ratio of computational performance over



(a) Double precision



(b) Single precision

**Figure 3: Matrix multiplication with CUBLAS 3.2 on an Nvidia Fermi M2070 GPU.** **(a)** The maximum performance in double precision is 302 Gflops and the distance between the peaks is 64. **(b)** The maximum performance in single precision is 622 Gflops and the distance between the peaks is 96.

communication bandwidth on a GPU will continue to increase. Hence, one of our objectives is to minimize communication. A number of existing algorithms on distributed-memory supercomputers have addressed the issue to minimize communication. In ScaLAPACK [10], the parallel Cholesky, QR, and LU factorizations have been proven to reach the communication lower bound to within a logarithmic factor [8, 16, 18]. This has inspired us to adapt these efficient methods to optimize communication on multicore and multi-GPU systems.

Another issue is that a GPU cannot reach its high performance until given a sufficiently large input. Figure 3 shows the performance of matrix multiplication on an Nvidia Fermi GPU using CUBLAS 3.2 in double precision and single precision, respectively. In general, the bigger the matrix size, the better the performance is, but the double precision matrix multiplication does not reach 95% of its maximum performance (max=302 Gflops) until the matrix size $N \geq 1088$. In single precision, it does not reach 95% of its maximum (max=622 Gflops) until $N \geq 1344$. Unlike GPU, it is common for a CPU core to reach 90% of its maximum when $N \geq 200$ for matrix multiplications. However, solving a big matrix of size $N > 1000$ by a single core is much slower than dividing it into smaller blocks and solving them in parallel by multiple cores. One could still use several cores to solve the big matrix in a fork-join way, but it will introduce additional synchronization overhead and more CPU idle time [3, 12, 14]. Therefore, we are motivated to design new algorithms to expose different tile sizes suitable for CPUs and GPUs, respectively.

## 2.3 Using a Static Distribution Strategy

At first sight, it seems that we could look at the multicore and multi-GPU system as a shared memory machine and offload an appropriate amount of compute-intensive work to the GPUs. This results in a dynamic load balancing problem where a runtime system is required to monitor the cores and GPUs and to profile the performance of different tasks on a core or a GPU to keep load balancing. In consideration of additional communication optimization and a related software cache mechanism, the problem becomes more challenging. Also the dynamic approach may prove inefficient due to the runtime and communication overhead (especially for small matrices). More important, to solve an $n \times n$ matrix, the cache size on a GPU must be close to $\frac{n^2}{P}$ to achieve an optimal communication volume as proven by Theorem 1.

THEOREM 1 ([20]). *The communication volume of the classic matrix multiplication algorithm is equal to* $\Omega(\frac{n^3}{P\sqrt{M}})$, *where* $n$ *is the matrix size, and* $M$ *is the local memory size on each of* $P$ *processes. When* $M = O(\frac{n^2}{P})$, *the communication volume* $\Omega(\frac{n^2}{\sqrt{P}})$ *is optimal.*

Note that the communication volume of matrix multiplication is also the lower bound for many other $\Theta(n^3)$ matrix computations such as Cholesky, QR, and LU factorizations. By setting the software cache size on each GPU as large as $O(\frac{n^2}{P})$, the caching scheme would become the same as a static 1D or 2D block distribution method regarding memory usage. Later Section 3.5 shows that using a static distribution method can guarantee a near lower bound communication volume. To optimize both computation and communication without resorting to complex scheduling policies and software caches, we choose to use the simple static distribution strategy for the domain of matrix computations.

## 3. RECTANGULAR TILE ALGORITHMS

We extend the tile algorithms [14] to rectangular tile algorithms and apply them to the Cholesky and QR factorizations. We also introduce a two-level partitioning method and a heterogeneous 1-D column block cyclic distribution to map tiles and tasks to the host and GPUs to minimize communication.

### 3.1 Hybrid-Size Rectangular Tiles

The rectangular tile algorithm divides a matrix into a mix of small and big rectangular tiles. Figure 2 depicts two examples of matrices that are divided into rectangular tiles. The two matrices have the same dimension but consist of a different number of small and big tiles.

However, the way to divide a matrix into rectangular tiles is not arbitrary. Constrained by the correctness of the algorithm, rectangular tiles must be aligned with each other and located in a collection of rows and columns. Their dimensions, however, could vary row by row or column by column (e.g., a row of tall tiles followed by a row of short tiles). Since we target a heterogenous system with two types of processors (i.e., CPU and GPU), we use two tile sizes: a small one for CPU and a big one for GPU. It should be easy to extend the algorithm to have more tile sizes.

On a heterogeneous multicore and multi-GPU system, we propose to use the following two-level partitioning scheme to create small tiles and big tiles:

1. At the top level, we divide a matrix into large square tiles of size $B \times B$.
2. We subdivide each top-level tile of size $B \times B$ into a number of small rectangular tiles of size $B \times b$ and a remaining tile.

We use this scheme because it not only results in a clean code structure but also allows us to use a simple auto-tuning method to achieve load balancing. For instance, as shown in Fig. 2 (a), we first divide the $12 \times 12$ matrix into four $6 \times 6$ tiles, then we divide each $6 \times 6$ tile into two $6 \times 1$ and one $6 \times 4$ rectangular tiles. How to partition the top-level large tiles is dependent on the performance of the host and the performance of a GPU. Section 4.2 introduces a method to determine an appropriate partitioning.

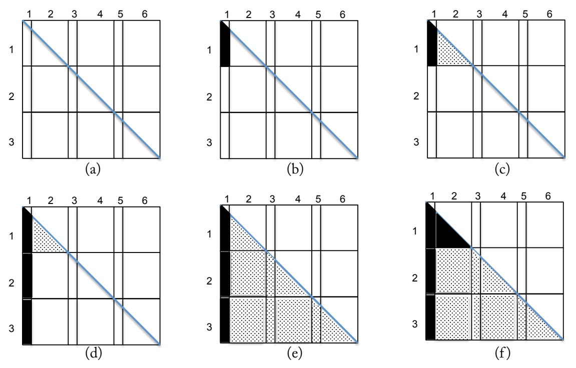


**Figure 4: The operations of rectangular tile Cholesky factorization.** **(a)** The symmetric positive definite matrix A. **(b)** computes POTF2' to solve $L_{11}$. **(c)** applies $L_{11}$ to update its right $A_{12}$ by matrix multiplication. **(d)** computes TRSMs for all tiles below $L_{11}$ to solve $L_{21}$ and $L_{31}$. **(e)** applies GSMMs to update all tiles on the right of TRSMs. **(f)** At the 2nd iteration, we repeat performing (b), (c), (d), (e) on the trailing submatrix that starts from the 2nd column.

## 3.2 Rectangular Tile Cholesky Factorization

Given a matrix A of size $n \times n$ and two tile sizes of B and b, A can be expressed as follows:

$$\left(\begin{array}{c|c|c} \overbrace{\begin{matrix} a_{11} & a_{12} \dots & A_{1s} \\ a_{21} & a_{22} \dots & A_{2s} \\ & \vdots & \\ a_{p1} & a_{p2} \dots & A_{ps} \end{matrix}}^{B} & \overbrace{\begin{matrix} a_{1(s+1)} & a_{1(s+2)} \dots & A_{1(2s)} \\ a_{2(s+1)} & a_{2(s+2)} \dots & A_{2(2s)} \\ & \vdots & \\ a_{p(s+1)} & a_{p(s+2)} \dots & A_{p(2s)} \end{matrix}}^{B} & \begin{matrix} \dots \\ \dots \\ \ddots \\ \dots \end{matrix} \end{array}\right), \text{ where}$$

an $\overbrace{a_{i(ks+1)} a_{i(ks+2)} \dots A_{i(ks+s)}}$ forms a large tile of size $B \times B$. Here $a_{ij}$ represents a small rectangular tile of size $B \times b$, and $A_{ij}$ represents a tile of size $B \times (B - b(s-1))$ that is usually larger. We also assume $n = pB$ and $B > b$.

Algorithm 1 shows the rectangular tile Cholesky factorization. Here we don't differentiate $a_{ij}$ and $A_{ij}$ and always use $A_{ij}$ since i and j imply a unique tile (either $a_{ij}$ or $A_{ij}$). In addition, we denote $A_{ij}$'s submatrix that starts from its local $x$-th row and $y$-th column to its original bottom right corner by $A_{ij}[x, y]$. We denote $A_{ij}[0, 0]$ by $A_{ij}$ for short.

**Algorithm 1** Rectangular Tile Cholesky Factorization

```
for t ← 1 to p do
  for d ← 1 to s do
    k ← (t - 1) * s + d /* the panel index */
    Δ ← (d - 1) * b /* row offset within a tile */
    POTF2'($A_{tk}$[Δ,0], $L_{tk}$[Δ,0])
    for j ← k + 1 to t * s do
      GSMM($L_{tk}$[Δ+b,0], $L_{tk}$[Δ+(j-k)*b,0], $A_{tj}$[Δ+b,0])
    end for
    for i ← t + 1 to p do
      TRSM($L_{tk}$[Δ,0], $A_{ik}$, $L_{ik}$)
    end for
    for i ← t + 1 to p do
      for j ← k + 1 to i * s do
        j' = $\lceil \frac{j}{s} \rceil$
        if (j' = t) GSMM($L_{ik}$, $L_{tk}$[Δ+(j-k)%s*b,0], $A_{ij}$)
        else GSMM($L_{ik}$, $L_{j'k}$[(j-1)%s*b,0], $A_{ij}$)
      end for
    end for
  end for
end for
```

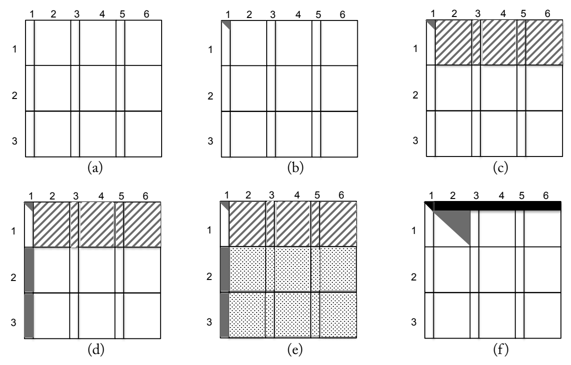


**Figure 5: The operations of rectangular tile QR factorization.** **(a)** The matrix A. **(b)** We compute QR factorization of $A_{11}$ to get $R_{11}$ and $V_{11}$. **(c)** We apply $V_{11}$ to update all tiles on the right of $A_{11}$ by calling LARFB. **(d)** computes TSQRTs for all tiles below $A_{11}$ to solve $V_{21}$ and $V_{31}$. **(e)** applies $V_{21}$ and $V_{31}$ to update all tiles on $V_{21}$ and $V_{31}$'s right hand side. **(f)** After the 1st iteration, we have solved the $R$ factors on the first row with a hight equal to $R_{11}$'s size. At the 2nd iteration, we repeat performing (b), (c), (d), (e) on the trailing submatrix that starts from the 2nd column.

The rectangular tile algorithm for Cholesky factorization invokes the same set of kernels as the tile Cholesky factorization algorithm [14] except for the kernel `POTF2'`:

- POTF2'($A_{tk}$, $L_{tk}$): Given a matrix $A_{tk}$ of size $m \times n$ and $m \geq n$, we let $A_{tk} = \binom{A_{tk1}}{A_{tk2}}$, where $A_{tk1}$ is of size $n \times n$, and $A_{tk2}$ is of $(m-n) \times n$. Similarly we let $L_{tk} = \binom{L_{tk1}}{L_{tk2}}$. POTF2' computes $\binom{L_{tk1}}{L_{tk2}}$ by solving $L_{tk1}$ = Choleksy($A_{tk1}$) and $L_{tk2} = A_{tk2} L_{tk1}^{-T}$.

- TRSM($L_{tk}$, $A_{ik}$, $L_{ik}$) computes $L_{ik} = A_{ik} L_{tk}^{-T}$.

- GSMM($L_{ik}$, $L_{jk}$, $A_{ij}$) computes $A_{ij} = A_{ij} - L_{ik} L_{jk}^{T}$.

Figure 4 illustrates the operations of the rectangular tile Cholesky factorization. It shows a matrix of $3 \times 3$ top-level large tiles (i.e., $p = 3$), each of which is divided into one small and one big rectangular tiles (i.e., $s = 2$). The algorithm goes through 6 ($= p \cdot s$) iterations, where the $k$-th iteration solves a submatrix starting from the $k$-th column. Since all iterations apply the same operations to different trailing submatrices, we only show the operations of the first iteration.

## 3.3 Rectangular Tile QR Factorization

Algorithm 2 shows the rectangular tile QR factorization. The rectangular tile QR uses the following set of kernels that are identical to those used in the tile QR factorization [14]. For completeness, we present them briefly here:

- GEQRT($A_{tk}$, $V_{tk}$, $R_{tk}$, $T_{tk}$) computes $(V_{tk}, R_{tk}, T_{tk})$ = QR($A_{tk}$).

- LARFB($A_{tj}$, $V_{tk}$, $T_{tk}$, $R_{tj}$) computes $R_{tj} = (I - V_{tk} T_{tk} V_{tk}^{T}) A_{tj}$.

- TSQRT($R_{tk}$, $A_{ik}$, $V_{ik}$, $T_{ik}$) computes $(V_{ik}, T_{ik}, R_{tk})$ = QR$\binom{R_{tk}}{A_{ik}}$.

**Algorithm 2** Rectangular Tile QR Factorization

```
for t ← 1 to p do
  for d ← 1 to s do
    k ← (t - 1) * s + d /* the panel index */
    Δ ← (d - 1) * b /* row offset within a tile */
    GEQRT($A_{tk}$[Δ,0], $V_{tk}$[Δ,0], $R_{tk}$[Δ,0], $T_{tk}$[Δ,0])
    for j ← k + 1 to p * s do
      LARFB($A_{tj}$[Δ,0], $V_{tk}$[Δ,0], $T_{tk}$[Δ,0], $R_{tj}$[Δ,0])
    end for
    for i ← t + 1 to p do
      TSQRT($R_{tk}$[Δ,0], $A_{ik}$, $V_{ik}$, $T_{ik}$)
    end for
    for i ← t + 1 to p do
      for j ← k + 1 to p * s do
        SSRFB($R_{tj}$[Δ,0], $A_{ij}$, $V_{ik}$, $T_{ik}$)
      end for
    end for
  end for
end for
```

- SSRFB($R_{tj}$, $A_{ij}$, $V_{ik}$, $T_{ik}$) computes $\binom{R_{tj}}{A_{ij}}$ $= (I - V_{ik}T_{ik}V_{ik}^T)\binom{R_{tj}}{A_{ij}}$.

Figure 5 illustrates the operations of the rectangular tile QR factorization. It shows a matrix of 3 tile rows and 6 tile columns. The algorithm goes through 6 iterations for the 6 tile columns. Since every iteration performs the same operations on a different trailing submatrix, Fig. 5 only shows the operations of the first iteration.

## 3.4 Heterogeneous Block Cyclic Distribution

We divide a matrix $A$ into $p \times (s \cdot p)$ rectangular tiles using the two-level partitioning method which first partitions $A$ into $p \times p$ large tiles at the top level, then partitions each large tile into $s$ rectangular tiles. Given a multicore and multi-GPU machine, we will distribute $A$'s tile columns to the host and a number of $P$ GPUs in a 1-D block cyclic way. That is, we statically allocate the $j$-th tile column to $P_x$, where $P_0$ represents the host system and $P_{x\geq 1}$ represents the $x$-th GPU. We compute $x$ as follows:

$$x = \begin{cases} ((\frac{j}{s} - 1) \mod P) + 1 & : \; j \mod s = 0 \\ 0 & : \; j \mod s \neq 0 \end{cases}$$

In other words, the columns whose indices are multiples of $s$ are mapped to the $P$ GPUs in a cyclic way and all the other columns go to the single host system.

Figure 6 (a) illustrates a matrix that is divided into rectangular tiles with the two-level partitioning method. Since we always map an entire tile column to either the host or a GPU, the figure omits the boundaries between rows to better illustrate the 1-D method. Figure 6 (b) displays how a matrix with 12 tile columns is allocated to one host and 3 GPUs using the heterogeneous 1-D column block cyclic distribution. The ratio of the sum of $s$-1 rectangular tiles over their remainder controls the load on the host and on each GPU. Section 4.2 describes a method to determine the ratio for load balancing.

## 3.5 Communication Cost

We consider the heterogeneous system a distributed memory machine such that the host system and the $P$ GPUs represent $P + 1$ processes. We also assume the broadcast between processes is implemented by a tree topology in order to make a fair comparison between our algorithms and the ScaLAPACK algorithms [10].

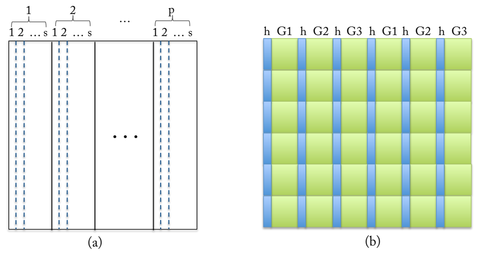


**Figure 6: Heterogeneous 1-D column block cyclic data distribution. (a)** The matrix A divided by a two-level partitioning method. $(p, s)$ determines a matrix partition. **(b)** Allocation of a matrix of $6 \times 12$ rectangular tiles (i.e., $p$=6, $s$=2) to a host and three GPUs: `h`, `G`$_1$, `G`$_2$, and `G`$_3$.

Given a system with one host, $P$ GPUs, and a matrix of size $n \times n$, we partition the matrix into $p \times (p \cdot s)$ rectangular tiles. The small rectangular tile is of size $B \times b$ and $n = p \cdot B$. The number of words communicated by at least one of the processes in the rectangular tile QR (or Cholesky) factorization is bounded by:

$$\text{Word} \;=\; \sum_{k=0}^{p-1}(n - kB)B\log(P+1) \simeq \frac{n^2}{2}\log(P)$$

The communication volume of the rectangular tile algorithm reaches the lower bound of $\Omega(\frac{n^2}{\sqrt{P}})$ (Theorem 1) to within a factor of $\sqrt{P}\log(P)$. If we use a 2-D block cyclic distribution instead of the 1-D distribution, we could attain the same communication volume as ScaLAPACK (i.e., $O(\frac{n^2}{\sqrt{P}}\log P)$ [8, 16]). However, it will result in more messages and produce lower performance in practice for the tile algorithms.

The number of messages sent or received by at least one process in the rectangular QR (or Cholesky) factorization is bounded by:

$$\text{Message} \;=\; \sum_{k=0}^{p-1}(p - k)s\log(P+1) \simeq \frac{p^2 s}{2}\log(P)$$

Although the number of messages is larger than that of ScaLAPACK [8, 16] by a factor of $O(p)$, the rectangular tile algorithms have much smaller messages and exhibit a higher degree of parallelism. Note that we also want to keep a high degree of parallelism in order to obtain high performance particularly on many-core systems [3, 6, 12].

## 4. IMPLEMENTATION

We have implemented a runtime system to support data-availability-driven execution where a parent task tries to trigger its children whenever possible. Before the execution, we use the heterogeneous 1-D block cyclic method to distribute a matrix across a host and different GPUs statically. Since we have preallocated the $j$-th tile column to the host or a certain GPU, we require a task modifying the $j$-th column be executed by the column's owner (the host or the GPU) to save data movement.

We extend the centralized-version runtime system of our previous work ([26] Section 3) to a new one that is suitable for heterogeneous multicore and multi-GPU systems. The centralized runtime system works on multicore architectures and consists of four components (look at Fig. 7 as if it had no GPUs):

- Task window: a fixed size task queue that stores all the generated but not finished tasks. It is an ordered list that keeps the serial semantic order between tasks.
- Ready task queue: a list of tasks whose inputs are all available. Each node in the list is just a pointer pointing to its corresponding task in the task window.
- Master thread: a single thread that executes a serial program and inserts new tasks to the task window.
- Computational threads: every core runs a computational thread. A computational thread picks up a task from the ready task queue whenever it becomes idle. After finishing the task, the thread scans the task window to determine which tasks are the children of the finished task and moves them to the ready task queue.

## 4.1 The Extended Runtime System

We have extended the centralized-version runtime system to make it accommodate to multicore and multi-GPU systems. Figure 7 shows the architecture of the extended runtime system. Note that the master thread and task window have not been changed.

However, since the host and the GPUs own different subsets of a matrix, we want to avoid the situation where a task accessing one GPU's data is dispatched to another GPU or the host. Also the task size intended for GPUs is much larger than that intended for CPUs. Therefore, we make the host and every GPU have their own ready task queues. If a ready task modifies a tile that belongs to the host or a GPU, it is sent to the host or GPU correspondingly.

We have also modified the computational threads. The new runtime system has two types of computational threads: one for CPU cores and the other for GPUs. If a host system has a number of $n$ cores and is connected with $P$ GPUs, the runtime system will launch $P$ computational threads to represent the $P$ GPUs and $(n - P)$ computational threads to represent the remaining CPU cores. Although a GPU computational thread is running on the host, it is able to invoke a GPU kernel automatically as long as the kernel's input is available in the GPU memory.

In the Nvidia CUDA 3.2 programming environment, data movement between different GPUs needs to be relayed by the host. The device memory accessible by a host thread is also restricted. If a host thread is attached to a GPU and allocates a chunk of memory on the GPU, usually only that thread can access the memory. Also there can be only one host thread attached to a GPU at a time. Hence, in our implementation, any data movement to or from a GPU is handled by the GPU's computational thread. With the newly released CUDA 4.0 RC, it is possible to merge several GPU computational threads into one thread.

We also assign a message box to each GPU computational thread. A core computational thread does not have a message box since it cannot access the GPU memory allocated by a GPU computational thread. When moving data either from the host to a GPU or from a GPU to the host, it is the GPU computational thread's responsibility to move the data. We consider three cases to handle the data movement among the host and GPUs:

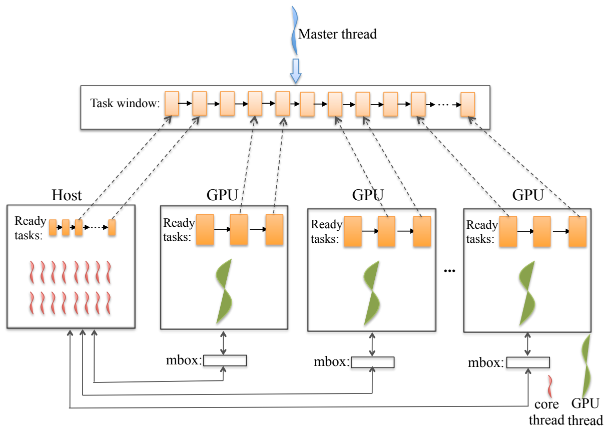


**Figure 7: The extended runtime system for heterogeneous multicore and multi-GPU architectures.**

- Host $\rightarrow$ GPU: after a core computational thread finishes a task, it wants to send the newly modified data to a GPU. The core thread then adds a message to the GPU's message box telling the GPU to get the data.
- GPU $\rightarrow$ Host: after a GPU computational thread finishes a task, the GPU thread adds a message to its own message box and sends the modified data later.
- GPU$_s$ $\rightarrow$ GPU$_t$: the runtime system generates two messages for this case. First, GPU$_s$ adds a "GPU$_s$ $\rightarrow$ Host" message to its own message box with a replay flag to GPU$_t$. When GPU$_s$ processes the message, it copies the data to the host and then adds a "Host $\rightarrow$ GPU$_t$" message to GPU$_t$'s message box telling GPU$_t$ to get the data from the host.

Nearly all communications in the Cholesky and QR factorizations are broadcast. During a GPU broadcast, our runtime system copies data only once from the GPU to the host. Note that it is necessary to copy the data to the host for broadcast operations. The other GPUs will simply copy the data from the host to themselves to minimize communication overhead. A GPU computational thread currently takes charge of both computation and communication and is implemented as follows. With the new CUDA 4.0 RC, it is also possible to move `process_msg` to a dedicated thread to decouple communication from computation.

```
/* lv stores the current thread's info */
while ( !done ) {
  /* calls cudaMemcpy to handle a message */
  process_msg(lv->mbox);
  if( ready = get_task(lv->readyQ) ) {
    compute_task(ready);
    /* adds new ready tasks and new messages */
    fire_children(ready, lv);
  }
}
```

## 4.2 Tile Size Tuning

Load imbalance could happen either between GPUs or between the host and GPUs. We use the 1-D block cyclic distribution method to achieve load balancing between GPUs. Also we adjust the ratio of the CPU tile size to GPU tile size to achieve load balancing between the host and GPUs.

We go through three steps to determine the best tile sizes:

1. We apply the two-level partitioning method to a matrix and suppose the top-level large tile size $B$ is already given (later we mention how to find $B$).

2. We use the following formula to estimate the best partition of size $B \times B_h$ to be cut off from each top-level tile of size $B \times B$:
$$B_h = \frac{\text{Perf}_{core} \cdot \#Cores}{\text{Perf}_{core} \cdot \#Cores + \text{Perf}_{gpu} \cdot \#GPUs} \cdot B$$
`Perf` denotes the maximum performance (in Gflops) of a dominant computational kernel in an algorithm.

3. We start from the estimated size $B_h$ and search for an optimal $B_h^*$ near $B_h$. We wrote a script to execute Cholesky or QR factorization with a random matrix of size $N = c_0 \cdot B \cdot \#GPUs$. In the implementation, we let $c_0 = 3$ to reduce the searching time. The script adapts the parameter of $B_h$ to search for the minimal difference between the host and the GPU computation time. If the host takes more time than a GPU, the script will decrease $B_h$ accordingly. This step is inexpensive since the granularity of our fine tuning is 64 for double precision and 96 for single precision due to the significant performance drop when a tile size is not a multiple of 64 or 96 (Fig. 3). In our experiments, it takes at most three attempts to find $B_h^*$.

The top-level tile size $B$ in Step 1 is critical for the GPU performance. To find the best $B$, we search for the minimal matrix size that provides the maximum performance for the dominant GPU kernel (i.e., `GEMM` for Cholesky and `SSRFB` for QR). Our search ranges from 128 to 2048 and is performed only once for every new kernel implementation and new GPU architecture. For Fermi GPUs, $B$ has to be at least 960 for good performance.

Unlike Step 1, Steps 2 and 3 depend on the number of cores and GPUs used in a computation. Note that there are at most ($\#Cores \cdot \#GPUs$) configurations on a given machine, and not every configuration is useful in practice (e.g., we often use all cores and all GPUs in scientific computing). Later our experimental results show that the auto-tuning method can attain a load imbalance ratio of less than 5% in most cases.

Lemma 1 also justifies our observation that the number of top-level tiles is not related to the load balancing between the host and GPUs for the rectangular tile QR factorization.

LEMMA 1. *Assume a matrix is divided into rectangular tiles by the two-level partitioning method. Given the performance of the host and GPUs, the partitioning of a large tile into two parts (one for the host and one for GPUs) to attain load balancing is not related to the number of large tiles for rectangular tile QR factorization.*

PROOF. Suppose there are $P$ GPUs. We let $t_{panel}^{(host)}$ and $t_{up}^{(host)}$ denote the time for the host to compute a panel factorization and a trailing matrix update for a single tile. Similarly, $t_{panel}^{(gpu)}$ and $t_{up}^{(gpu)}$ denote the time on a GPU. We assume the kernel computation time does not change much during an execution. So given a matrix partitioned into $p \times p$ large tiles at the top level and assuming $p$ is a multiple of $P$, the execution time of the host:
$$T_{host} = \sum_{i=1}^{p} (i \times t_{panel}^{(host)} + 2i^2 \times t_{up}^{(host)}).$$

**Table 1: Experiment Environment**

| | Host | Attached GPUs |
|---|---|---|
| Processor type | Intel Xeon X5660 | Nvidia Fermi M2070 |
| Clock rate | 2.8 GHz | 1.15 GHz |
| Processors per node | 2 | 3 |
| Cores per processor | 6 | 14 SMs |
| Memory | 24 GB | 6 GB per GPU |
| Theo. peak (double) | 11.2 Gflops/core | 515 Gflops/GPU |
| Theo. peak (single) | 22.4 Gflops/core | 1.03 Tflops/GPU |
| Max gemm (double) | 10.7 Gflops/core | 302 Gflops/GPU |
| Max gemm (single) | 21.4 Gflops/core | 635 Gflops/GPU |
| Max ssrfb (double) | 10.0 Gflops/core | 223 Gflops/GPU |
| Max ssrfb (single) | 19.8 Gflops/cores | 466 Gflops/GPU |
| BLAS/LAPACK lib | Intel MKL 10.3 | CUBLAS 3.2, MAGMA |
| Compilers | Intel compilers 11.1 | CUDA toolkit 3.2 |
| OS | CentOS 5.5 | Kernel module 260.19.14 |
| System interface | – | PCIe x 16 Gen2 |

And the execution time of each GPU:
$$T_{gpu} = \sum_{i=1}^{p} (i \times t_{panel}^{(gpu)} + 2\frac{i^2}{P} \times t_{up}^{(host)}).$$

It is easy to see that $T_{host}$=$T_{gpu}$ is not related to the number of large tiles $p$. Similarly we can reach the same conclusion for the rectangular tile Cholesky factorization. □

## 5. PERFORMANCE EVALUATION

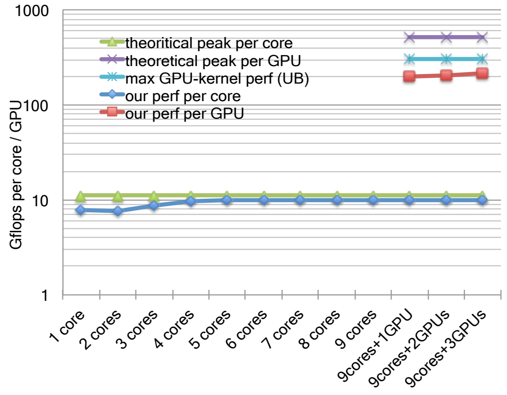
We have implemented the rectangular tile Cholesky and QR factorizations in double and single precisions. In this section, we present their performance data in weak scalability and strong scalability, respectively. We then measure their load imbalance for three different configurations. We also analyze the efficiency of the runtime system. For every experiment, we have verified that its numerical result is correct.

We conducted experiments on a single node of the heterogeneous Keeneland system at the Oak Ridge National Laboratory. The system has 120 nodes and each node has two Intel Xeon X5660 (Westmere) hexa-core processors and three Nvidia Fermi M2070 GPUs. Table 1 lists the hardware and software resources used in our experiments. The table also lists the maximum performance of `gemm` and `ssrfb` used by Cholesky factorization and QR factorization, respectively. The kernel performance serves as an upper bound for the whole program's performance.

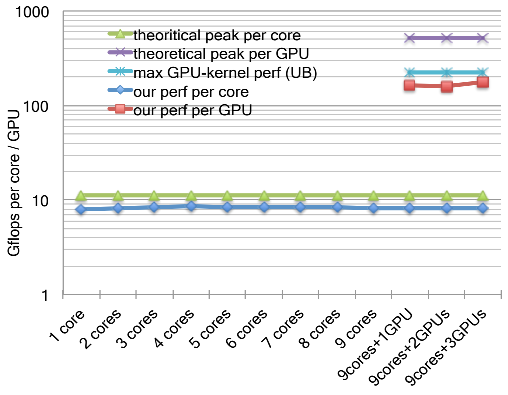
### 5.1 Weak Scalability

We use weak scalability to evaluate the capability of a program to solve potentially larger problems when more computing resources are added. In the weak scalability experiment, we increase the input size accordingly when we increase the number of cores and GPUs.

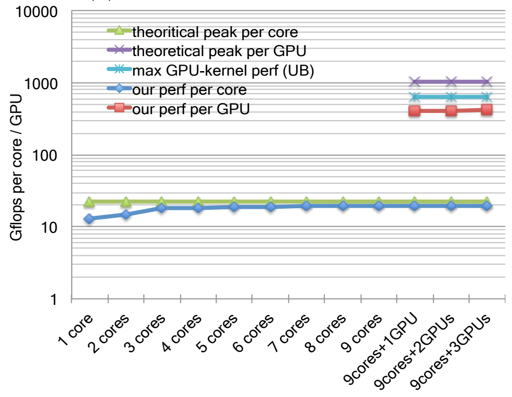
Figure 8 shows the performance of Cholesky and QR factorizations in double precision and single precision, respectively. The x-axis shows the number of cores and GPUs used in the experiment. The y-axis shows Gflops-per-core or Glfops-per-GPU on a logarithmic scale. In each subfigure there are five curves: two "`theoretical peak`"s to denote the theoretical peak performance from a single core or from a single GPU, one "`max GPU-kernel`" to denote the maximum GPU kernel performance used by Cholesky or QR factorization which is the upper bound of the whole program, "`our perf per core`" to denote the performance of our program



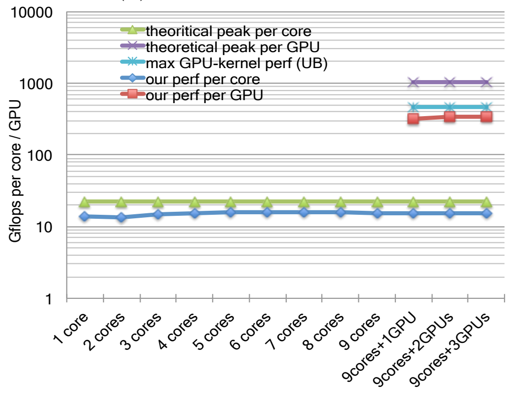
(a) Cholesky in double precision



(b) QR in double precision

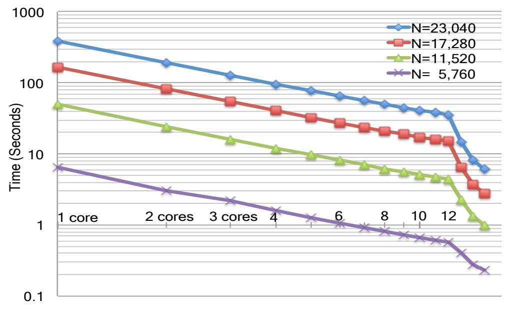


(c) Choleksy in single precision

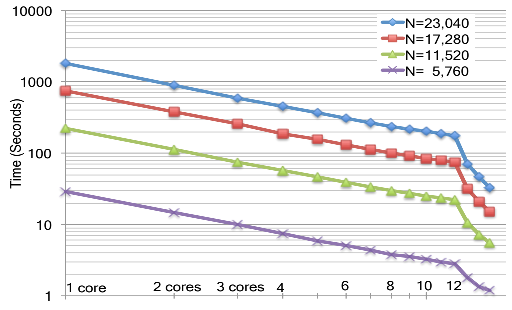


(d) QR in single precision

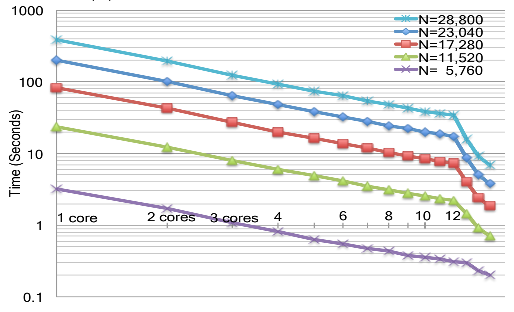
**Figure 8: Weak scalability.** The input size increases too while adding more core and GPUs. The y-axis is presented on a logarithmic scale. OverallPerformance = (Perf$_{per\_core}$ * #cores) + (Perf$_{per\_gpu}$ * #gpus). Note that ideally the performance per core or per GPU should be a flat line.



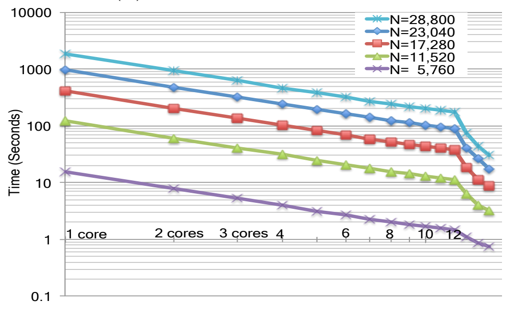
(a) Cholesky in double precision



(b) QR in double precision



(c) Cholesky in single precision



(d) QR in single precision

**Figure 9: Strong scalability. The last three ticks on the x-axis (after `12 cores`) are: `11 cores + 1 GPU`, `10 cores + 2 GPUs`, and `9 cores + 3 GPUs`.** The input size is fixed while adding more cores and GPUs. Both x-axis and y-axis are presented on a logarithmic scale. Note that ideally a strong scalability curve should be a straight line in a log-log graph.

on each CPU core, and "`our perf per GPU`" to denote our program performance on each GPU.

In the experiments, we first increase the number of cores from 1 to 9. Then we add 1, 2, and 3 GPUs to the 9 cores. The input sizes for the double precision experiments (i.e., (a), (b)) are: 1000, 2000, . . . , 9000, followed by 20000, 25000, and 34000. The input sizes for single precision (i.e., (c), (d)) are the same except for the last three sizes that are 30000, 38000, and 46000. From Fig. 8, we can see that Cholesky and QR factorizations are scalable on both CPU cores and GPUs. Note that ideally the performance per core or per GPU is a flat line.

The overall performance of Cholesky factorization or QR factorization can be derived by summing up (perf-per-core $\times$ NumberCores) and (perf-per-gpu $\times$ NumberGPUs). For instance, the double precision Cholesky factorization using 9 cores and 3 GPUs attains an overall performance of 742 Gflops, which is 74% of the upper bound and 45% of the theoretical peak. Similarly, the single precision Cholesky factorization has an overall performance of 1.44 Tflops, which is 69% of the upper bound and 44% of the theoretical peak. Moreover, the overall performance of QR factorization is 79% of the upper bound in double precision, and 73% of the upper bound in single precision.

## 5.2 Strong Scalability

We use strong scalability to evaluate how much faster a program can solve a given problem if a user is provided with more computing resources. In the experiment, we measure the execution time to solve a number of matrices with different sizes. For each fixed-size matrix, we keep adding more computing resources to solve it.

Figure 9 shows the wall clock execution time of Cholesky and QR factorizations in double precision and single precision, respectively. Each graph has several curves, each of which represents a matrix of size $N$. The x-axis shows the number of cores and GPUs on a logarithmic scale. That is, we solve a matrix of size $N$ using 1, 2, . . . , 12 cores, followed by 11 cores + 1 GPU, 10 cores + 2 GPUs, and 9 cores + 3 GPUs. The y-axis shows execution time in seconds also on a logarithmic scale. Note that an ideal strong scalability curve should be a straight line in a log-log graph.

In Fig. 9 (a), we reduce the execution time of Cholesky factorization in double precision from 393 seconds to 6 seconds for $N$=23,040, and from 6.4 to 0.2 seconds for $N$=5,760. In (b), we reduce the execution time of QR factorization in double precision from 1790 to 33 seconds for $N$=23,040, and from 29 seconds to 1 second for $N$=5,760. Similarly, (c) and (d) display the performance in single precision. In (c), we reduce the execution time of Cholesky factorization from 387 to 7 seconds for $N$=28,800, and from 3.2 to 0.2 seconds for $N$=5,760. In (d), we reduce the execution time of QR factorization from 1857 to 30 seconds for $N$=28,800, and from 16 to 0.7 seconds for $N$=5,760.

## 5.3 Load Imbalance

Section 4.2 described how we determined the best tile sizes to achieve load balancing. In this section, we use the metric `imbalance_ratio` to evaluate the quality of our load balancing, where $imbalance\_ratio = \frac{MaxLoad}{AvgLoad}$ as proposed in [22]. Here we use computational time to represent the load on a host or GPU. In our implementation, we put timers above and below every computational kernel and sum them up to measure the computational time.

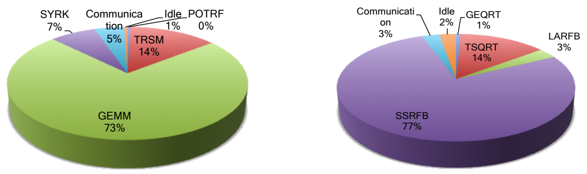
Our experiment uses three different configurations: 3 cores + 1 GPU, 6 cores + 2 GPUs, and 9 cores + 3 GPUs. Given an algorithm (either Cholesky or QR factorization), we first determine the top-level tile size, $B$, for the algorithm; then we determine the partitioning size, $B_h$, for each configuration using the auto-tuning method. We apply the tuned tile sizes to various matrices. For simplicity, we let the matrix size be a multiple of $B$ and suppose the number of tile columns is divisible by the number of GPUs. If it is not divisible by the number of GPUs, we divide its remainder ($\in$[1, NumberGPUs-1]) among all GPUs using a smaller chunk.

Figure 11 shows the imbalance ratio for three configurations each with double and single precisions. An imbalance ratio of 1.0 indicates a perfect load balancing. We can see that most of the imbalance ratios are less than 5%. A few of the first columns have an imbalance ratio of up to 17%. This is because their corresponding matrices have too few top-level tiles. For instance, the first column of the `9Cores+3GPUs` configuration has a matrix of three top-level tiles (Fig. 11 (c), (f)). We could increase the number of tiles to alleviate this problem by reducing the top-level tile size.

## 5.4 Runtime System Efficiency

This section investigates whether our runtime system can schedule tasks efficiently and how much better we can improve our program performance. Here we show the execution time breakdown for the double precision Cholesky and QR factorizations to see where their time goes. The single precision results are the same and not shown here.

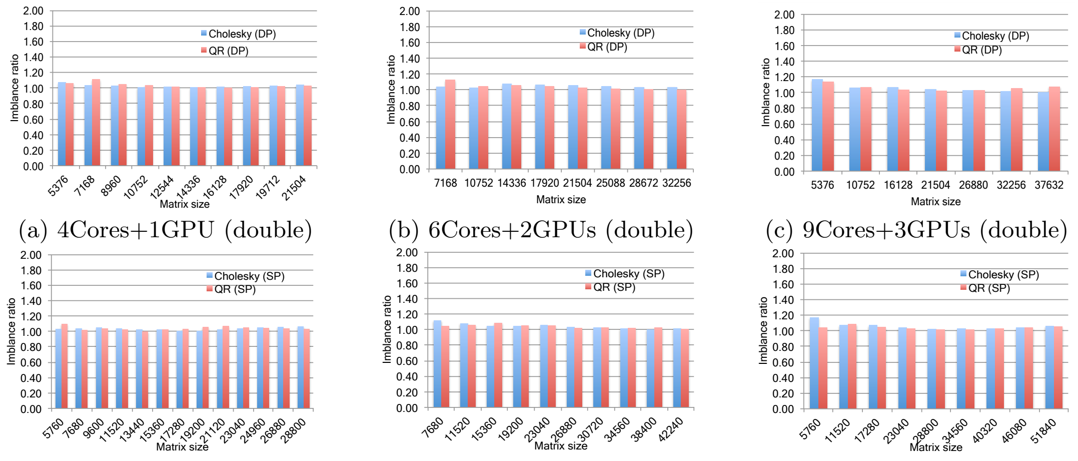
Figure 10 shows the execution time breakdown of the Cholesky and QR factorization experiments that use 9 cores and 3 GPUs. The time breakdown data is actually collected from the `9cores+3GPUs` experiment in the weak scalability experiments shown in Fig. 8. The corresponding load imbalance ratio for the two experiments are 3% and 0.1%, respectively. Note that we show the time breakdown for one of the GPUs because of their balanced load.



(a) Cholesky factorization (b) QR factorization

**Figure 10: Execution time break down on a GPU for double precision Cholesky and QR factorizations.**

As shown in Fig. 10 (a), the double precision Cholesky factorization spends 73% of its time on kernel `GEMM`, 14% on `TRSM`, 7% on kernel `SYRK`, and 5% on communication. There is only 1% of idle time. In (b), all the QR computational kernels take 95% of the total execution time, the communication time takes 3%, and the idle time is 2%. From the analysis, we can see that our runtime system works efficiently and the communication time occupies a small percentage of time. To further improve the performance, we will need a better implementation of TRSM and TSQRT kernels for GPUs. In addition, the maximum performance of SSRFB for GPUs is only 74% of the maximum GEMM performance and it needs to be improved too.



(a) 4Cores+1GPU (double) (b) 6Cores+2GPUs (double) (c) 9Cores+3GPUs (double)

(d) 4Cores+1GPU (single) (e) 6Cores+2GPUs (single) (f) 9Cores+3GPUs (single)

**Figure 11: Load imbalance.** The metric $imbalance\_ratio = \frac{MaxLoad}{AvgLoad}$. The closer the ratio is to 1.0 the better.

## 6. RELATED WORK

There are a few linear algebra libraries developed for GPU devices. CUBLAS has implemented the standard BLAS (Basic Linear Algebra Subroutines) library on GPUs [25]. MAGMA and CULA have also implemented a subset of the standard LAPACK library on GPUs [27, 19]. But currently they do not support computations using multiple cores and multiple GPUs.

Demmel et al. have developed the communication-avoiding QR factorization entirely on a single GPU to minimize communication for tall and skinny matrices [5]. By contrast, other GPU implementations (including ours) send panel factorizations back to the host CPUs to compute. Fogue et al. presented a strategy to port the existing PLAPACK library to GPU-accelerated clusters [17]. They require that GPUs compute most of the compute-intensive work and store all data in GPU memories to reduce communication.

StarSs is a programming model using directives to annotate a sequential source code to execute on various architectures such as SMP, CUDA, and Cell [7]. A programmer is responsible for specifying which piece of code should be executed on a GPU. Then its runtime can execute the annotated code in parallel on the host and GPUs. It uses a software cache mechanism to reduce data transfer time. Charm++ is an object-oriented parallel language that uses a dynamic load balancing runtime system to map objects to processors dynamically [21]. StarPU also develops a dynamic load balancing framework to execute a sequential code on the host and GPUs in parallel and has been applied to the QR factorization [1]. Differently, we use a simple static data distribution to minimize communication and attain high performance simultaneously for matrix computations.

There are many researches that have studied how to apply static data distribution strategies to heterogeneous distributed memory systems. Boulet et al. designed an algorithm to map a set of uniform tiles to a 1-D collection of heterogeneous processors [11]. Beaumont et al. proposed a heuristic 2-D block data allocation to extend ScaLAPACK to work on heterogeneous clusters [9]. Lastovetsky et al. developed a static data distribution strategy that takes into account both processor heterogeneity and memory heterogeneity for dense matrix factorizations [23].

## 7. CONCLUSION AND FUTURE WORK

The heterogeneous multicore and multi-GPU architectures have imposed a challenging task to develop new parallel software owning to a variety of reasons. These reasons include processor heterogeneity, memory heterogeneity, many cores, distributed memory spaces, and an increasing gap between computational performance and communication bandwidth. In order to provide efficient support for matrix computations, we are focused on four objectives: fine-granularity (for a high degree of parallelism), minimized synchronization, minimized communication, and load balancing.

In this paper, we design rectangular tile algorithms with hybrid tiles to provide high degree of parallelism and cope with processor heterogeneity. We treat the multicore and multi-GPU system as a distributed-memory machine and deploy a heterogeneous 1-D column block cyclic data distribution to minimize communication. We also introduce an auto-tuning method to determine the best tile sizes that not only attain high performance, but also achieve load balancing. Furthermore, we have implemented a new runtime system for the heterogeneous multicore and multi-GPU architectures. Although we have applied our approach to the domain of matrix computations, the same methodology and principles can be applied to other scientific applications on multicore and multi-GPU architectures (e.g., heterogeneous algorithms with hybrid tasks, two-level partitioning, a distributed-memory perspective on GPUs, adapting efficient algorithms on clusters, auto-tuning, and so on).

In our approach, the largest matrix size is constrained by the memory capacity of each GPU since we use a static block cyclic distribution method. An approach to solving this issue is to use an "out-of-core" algorithm such as the left looking algorithm, to compute a matrix panel by panel [15]. Other future work includes applying the rectangular tile algorithms to distributed memory clusters with heterogeneous multicore and multi-GPU nodes by distributing the top-level tiles to nodes in a 2-D block cyclic way.

## Acknowledgment

We are very grateful to Bonnie Brown for all her assistance with this paper.

## 8. REFERENCES

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, Alaska, USA, 2011.

[2] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users' Guide. Technical report, ICL, UTK, 2010.

[3] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 20:1–20:12, New York, NY, USA, 2009. ACM.

[4] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[5] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. Technical Report UCB/EECS-2010-131, EECS Department, University of California, Berkeley, October 2010.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[7] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 245–252, New York, NY, USA, 2009. ACM.

[9] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Transactions on Computers*, 50:1052–1070, 2001.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[11] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25(5):547 – 568, 1999.

[12] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 1–10, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20(13):1573–1590, 2008.

[14] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, 2009.

[15] E. F. D'Azevedo, F. DÕazevedo, and J. J. Dongarra. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR and Cholesky factorization routines, 1997.

[16] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. LAPACK Working Note 204, UTK, August 2008.

[17] M. FoguŐ, F. D. Igual, E. S. Quintana-ortŠ, and R. V. D. Geijn. Retargeting PLAPACK to clusters with hardware accelerators. FLAME Working Note 42, 2010.

[18] L. Grigori, J. W. Demmel, and H. Xiang. Communication avoiding Gaussian elimination. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 29:1–29:12, Piscataway, NJ, USA, 2008. IEEE Press.

[19] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: Hybrid GPU accelerated linear algebra routines. In *SPIE Defense and Security Symposium (DSS)*, April 2010.

[20] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64:1017–1026, September 2004.

[21] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn. Scaling hierarchical N-body simulations on GPU clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Z. Lan, V. E. Taylor, and G. Bryan. A novel dynamic load balancing scheme for parallel systems. *J. Parallel Distrib. Comput.*, 62:1763–1781, December 2002.

[23] A. Lastovetsky and R. Reddy. Data distribution for dense factorization on computers with memory heterogeneity. *Parallel Comput.*, 33:757–779, December 2007.

[24] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, March 2010.

[25] NVIDIA. CUDA CUBLAS Library, 2010.

[26] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[27] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA Users' Guide. Technical report, ICL, UTK, 2010.