Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems [*]

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

Stanimire Tomov
EECS Department
University of Tennessee
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 approach to utilizing all CPU cores and all GPUs on heterogeneous multicore and multi-GPU systems to support dense matrix computations efficiently. The main idea is that we treat a heterogeneous system as a distributed-memory machine, and use a heterogeneous multi-level block cyclic distribution method to allocate data to the host and multiple GPUs to minimize communication. We design heterogeneous algorithms with hybrid tiles to accommodate the processor heterogeneity, and introduce an auto-tuning method to determine the hybrid tile sizes to attain both high performance and load balancing. We have also implemented a new runtime system and applied it to the Cholesky and QR factorizations. Our approach is designed for achieving four objectives: a high degree of parallelism, minimized synchronization, minimized communication, and load balancing. Our experiments on a compute node (with two Intel Westmere hexa-core CPUs and three Nvidia Fermi GPUs), as well as on up to 100 compute nodes on the Keeneland system [31], demonstrate great scalability, good load balancing, and efficiency of our approach.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*SIMD*

General Terms

Algorithms, Design, Performance

Keywords

Heterogeneous algorithms, hybrid CPU-GPU architectures, numerical linear algebra, runtime systems

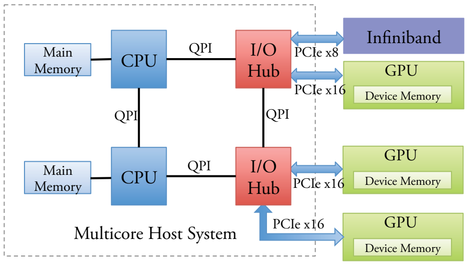


Figure 1: Architecture of a heterogeneous multi-core and multi-GPU system. The host and the GPUs have separate memory spaces.

1. INTRODUCTION

As the performance of both multicore CPUs and GPUs continues to scale at a Moore's law rate, it is becoming more common to use heterogeneous multi-core and multi-GPU architectures to attain the highest performance possible from a single compute node. Before making parallel programs run efficiently on a distributed-memory system, it is critical to achieve high performance on a single node first. However, the heterogeneity in the multi-core and multi-GPU architecture has introduced new challenges to algorithm design and system software.

Figure 1 shows the architecture of a heterogeneous multicore and multi-GPU system. The host system has two multicore CPUs and is connected to three GPUs each with a dedicated PCI-Express connection. To design new parallel software on this type of heterogeneous architectures, we must consider the following features: (1) Processor heterogeneity between CPUs and GPUs; (2) The host and the GPUs have different memory spaces and an explicit memory copy is required to transfer data between them; (3) As the performance gap between a GPU and its PCI-Express connection becomes larger, network is eventually the bottleneck for the entire system; (4) GPUs are optimized for throughput and expect a larger input size than CPUs which are optimized for latency [25]. We take into account these aspects and strive to meet the following objectives: a high degree of parallelism, minimized synchronization, minimized communication, and load balancing. In this paper, we present heterogeneous tile algorithms, heterogeneous multi-level block cyclic data distribution, an auto-tuning method, and an extended runtime system to achieve the objectives.

The heterogeneous tile algorithms build upon the previous tile algorithms [12], which divide a matrix into square

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.
Copyright 2012 ACM 978-1-4503-1316-2/12/06 ...$10.00.

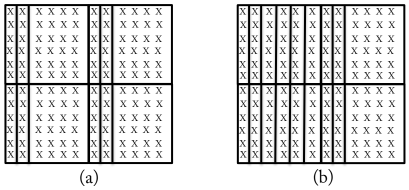


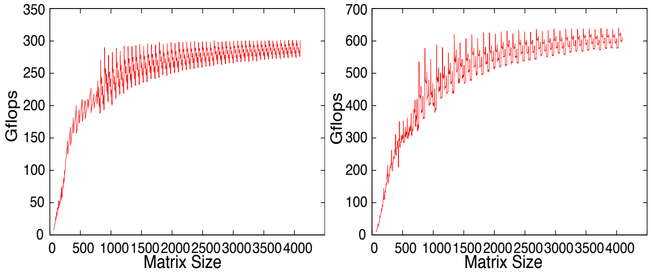
Figure 2: Matrices consisting of a mix of small and large rectangular tiles. (a) A 12×12 matrix is divided into eight small tiles and four large tiles. **(b)** A 12×12 matrix is divided into sixteen small tiles and two large tiles.

tiles and exhibit a high degree of parallelism and minimized synchronizations. A unique tile size, however, does not work well for both CPU cores and GPUs simultaneously. A large tile will clobber a CPU core, and a small tile cannot attain high performance on a GPU. Therefore, we have extended the tile algorithms so that they consist of two types of tiles: smaller tiles for CPU cores, and larger tiles for GPUs. Figure 2 depicts two matrices consisting of a set of small and large tiles. The heterogeneous tile algorithms execute in a fashion similar to the tile algorithms such that whenever a task computing a tile of [I, J] is completed, it will trigger a set of new tasks in [I, J]'s neighborhood.

We statically store small tiles on the host, and large tiles on the GPUs, respectively, to cope with processor heterogeneity and reduce redundant data transfers. We also place greater emphasis on communication minimization by viewing the multicore and multi-GPU system as a distributed memory machine. In order to allocate the workload to the host and different GPUs evenly, we propose a two-level 1-D block cyclic data 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 an appropriate size slice from each block and assign it to the host CPUs. Our analysis shows that the static distribution method is able to reach a near lower-bound communication volume. Furthermore, we have designed an auto-tuning method to determine the best size of the slice such that the amount of work on the host is equal to that on each GPU.

We design a runtime system to support dynamic scheduling on multicore and multi-GPU systems. The runtime system allows programs to be executed in a data-availability-driven model where a parent task always tries to trigger its children. In order to address the specialties of the heterogeneous system, we extend a centralized runtime system to a new one. The new runtime system is "hybrid" in the sense that its scheduling and computing components are centralized and resident in the host system, but its data, pools of buffers, communication components, and task queues are distributed for the host and different GPUs.

To our best knowledge, this is the first work to utilize multi-node, multi-core, and multi-GPU to solve fundamental matrix computation problems. Our work makes the following contributions: (i) making effective use of all CPU cores and all GPUs, (ii) new heterogeneous algorithms with hybrid tiles, (iii) heterogeneous block cyclic data distribution based upon a novel multi-level partitioning scheme, (iv) an auto-tuning method to achieve load balancing, and (v) a new runtime system to accommodate the features of the heterogeneous system (i.e., a hybrid of a shared- and distributed-memory system).



(a) Double precision (b) Single precision

Figure 3: Matrix multiplication with CUBLAS 4.0 on an Nvidia Fermi M2070 GPU. (a) The max performance in double precision is 302 Gflops and the distance between the peaks is 64. **(b)** The max performance in single precision is 622 Gflops and the distance between the peaks is 96.

We conducted experiments on the Keeneland system [31] at the Oak Ridge National Laboratory. On a compute node with two Intel Westmere hexa-core CPUs and three Nvidia Fermi GPUs, both our Cholesky factorization and QR factorization exhibit scalable performance. In terms of weak scalability, we attain a nearly constant Gflops-per-core and Gflops-per-GPU performance from one core to nine cores and three GPUs. And in strong scalability, we reduce the execution time by two orders of magnitude from one core to nine cores and three GPUs.

Furthermore, our latest results demonstrate that the approach can be applied to distributed GPUs directly, and is able to scale efficiently from one node (0.7 Tflops) to 100 nodes (75 Tflops) on the Keeneland system.

2. MOTIVATION

2.1 Optimization Concerns on GPUs

Although computation performance can be increased by adding more cores to a GPU, it is much more difficult to increase network performance at the same rate. We expect the ratio of computation performance to communication bandwidth on GPUs will continue to increase, hence one of our objectives is to reduce communication. In ScaLAPACK [9], the parallel Cholesky, QR, and LU factorizations have been proven to reach the communication lower bound to within a logarithmic factor [7, 14, 17]. This has inspired us to adapt these efficient methods to optimize communication on the new multicore and multi-GPU architectures.

Another issue is that GPUs cannot reach their high performance until given a sufficiently large input size. Figure 3 shows the performance of matrix multiplication on an Nvidia Fermi GPU using CUBLAS 4.0 in double precision and single precision, respectively. In general, the bigger the matrix size, the better the performance is. The double-precision matrix multiplication does not reach 95% of its maximum performance until the matrix size $N = 1088$. In single precision, it does not reach 95% of its maximum until $N = 1344$. Unlike GPUs, it is very common for CPU cores to reach 90% of their maximum performance when $N \geq 200$ for matrix multiplications. However, solving a large matrix of size $N \simeq 1000$ by a single CPU core is much slower than dividing it into submatrices and solving them by multiple cores in parallel. One could still use several cores to solve the large

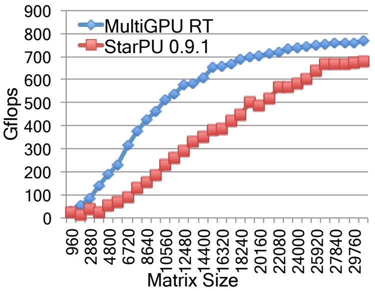


Figure 4: A comparison between our static scheduling approach and the dynamic scheduling approach of StarPU on Cholesky factorization (double precision) using three Fermi GPUs and twelve cores.

matrix in a fork-join manner, but it will introduce additional synchronization overhead and more CPU idle time on multicore architectures [3, 11, 12]. Therefore, we are motivated to design new heterogeneous algorithms to expose different tile sizes suitable for CPUs and GPUs, respectively.

2.2 Choosing a Static Strategy

We have decided to use a static strategy to solve dense matrix problems due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead [18]. In fact, the de facto standard library of ScaLAPACK [9] also uses a static distribution method on distributed-memory supercomputers.

We performed experiments to compare our static distribution strategy to an active project called StarPU [5] that uses a dynamic scheduling approach. Both our program and StarPU call identical computational kernels. Figure 4 shows the performance of Cholesky factorization in double precision on a Keeneland compute node with three Fermi GPUs and two Intel Westmere 6-core CPUs. Our performance data shows that the static approach can outperform the dynamic approach by up to 250%.

Dynamic strategies are typically more general and can be applied to many other applications. However, they often result in sophisticated scheduling systems that are required to make good decisions on load balancing and communication minimization on-the-fly. It is also non-trivial to guarantee that the on-the-fly decisions are globally good. Penalties for mistakes include extra data transfers, delayed tasks on the critical path, higher cache miss rate, and more idle time. For instance, we have found that StarPU sent critical tasks to slower CPUs to compute such that the total execution time has been increased. By contrast, a well-designed static method for certain domains can be proven to be globally near-optimal [18].

3. HETEROGENEOUS TILE ALGORITHMS

We extend tile algorithms [12] to heterogeneous algorithms and apply them to the Cholesky and QR factorizations. We then design a two-level block cyclic distribution method to support the heterogeneous algorithms, as well as an auto-tuning method to determine the hybrid tile sizes.

3.1 Hybrid Tile Data Layout

Heterogeneous tile algorithms divide a matrix into a set of small and large tiles. Figure 2 shows that two matrices are divided into hybrid rectangular tiles. Constrained by the correctness of the algorithm, hybrid 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 heterogeneous systems with two types of processors (i.e., CPUs and GPUs), we use two tile sizes: small tiles for CPUs and large tiles for GPUs. It should be easy to extend the algorithms to include more tile sizes.

Algorithm 1 Heterogeneous Tile Cholesky Factorization

```
for t ← 1 to p do
   for d ← 1 to s do
      k ← (t - 1) * s + d /* the $k$-th tile column */
      Δ ← (d - 1) * b /* local offset within a tile */
      POTF2'($A_{tk}$[Δ,0], $L_{tk}$[Δ,0])
      for j ← k + 1 to t * s /* along the $t$-th tile row */ 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 /* along the $k$-th tile column */ do
         TRSM($L_{tk}$[Δ,0], $A_{ik}$, $L_{ik}$)
      end for
      /* trailing submatrix update */
      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
```

We create hybrid tiles with the following two-level partitioning scheme: (1) At the top level, we divide a matrix into large square tiles of size $B \times B$. (2) Then 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 allows us to use a simple auto-tuning method to achieve load balancing, but also results in a regular code structure. For instance, as shown in Fig. 2 (a), we first divide the 12×12 matrix into four 6×6 tiles, then we divide each 6×6 tile into two 6×1 and one 6×4 rectangular tiles. How to partition the top-level large tiles is dependent on the performance of the host and the performance of each GPU. Section 3.6 will introduce our method to determine a good partitioning.

3.2 Heterogeneous Tile Cholesky Factorization

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

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

a_{ij} represents a small rectangular tile of size $B \times b$, and A_{ij} represents an often larger tile of size $B \times (B - b(s-1))$. Note that $\overbrace{a_{11} a_{12} \ldots A_{1s}}$ forms a square tile of size $B \times B$. We also assume $n = pB$ and $B > b$.

Algorithm 1 shows the heterogeneous tile Cholesky factorization. Here we do not differentiate a_{ij} and A_{ij} and always use A_{ij} to denote a tile. We also denote A_{ij}'s submatrix that starts from its local x-th row and y-th column, to its original

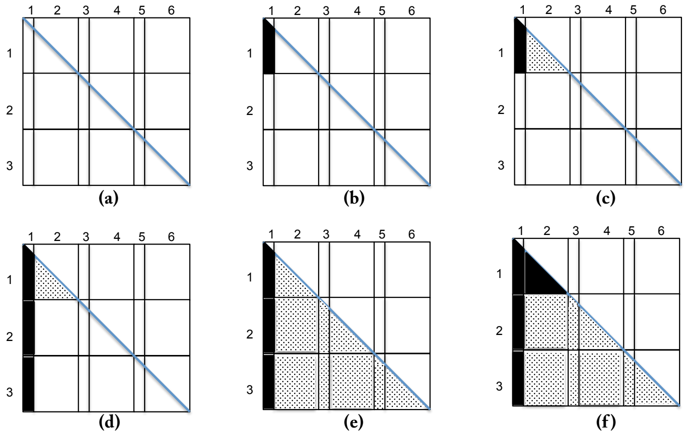


Figure 5: The operations of heterogeneous tile Cholesky factorization. **(a)** The symmetric positive definite matrix A. **(b)** Compute POTF2' to solve L_{11}. **(c)** Apply L_{11} to update its right A_{12} by GSMM. **(d)** Compute TRSMs for the two tiles below L_{11}. **(e)** Apply GSMMs to update all tiles on the right of the first tile column. **(f)** At the second iteration, we repeat performing (b), (c), (d), (e) on the trailing submatrix that starts from the second tile column.

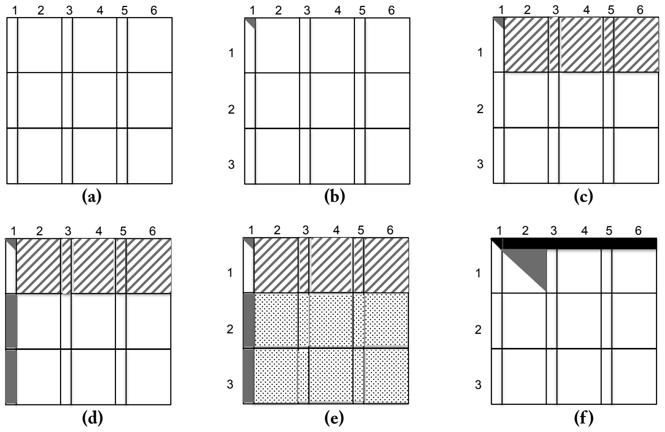


Figure 6: The operations of heterogeneous tile QR factorization. **(a)** The matrix A. **(b)** Compute the QR factorization of A_{11} to get R_{11} and V_{11}. **(c)** Apply V_{11} to update all tiles on the right of A_{11} by calling LARFB. **(d)** Compute TSQRTs for all tiles below A_{11} to solve V_{21} and V_{31}. **(e)** Apply SSRFBs 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 second iteration, we repeat performing (b), (c), (d), (e) on the trailing submatrix.

bottom right corner by $A_{ij}[x, y]$. We denote $A_{ij}[0, 0]$ by A_{ij} for short. As shown in the algorithm, at the k-th iteration (corresponding to the k-th tile column), we first factorize the diagonal tile on the k-th tile column by POTF2', then solve the tiles located below the diagonal tile by TRSMs, followed by updating the trailing submatrix on the right side of the k-th tile column by GSMMs.

Figure 5 illustrates the operations to factorize a matrix of 3×3 top-level large tiles (i.e., $p = 3$), each of which is divided into one small and one large rectangular tiles (i.e., $s = 2$). The factorization goes through six ($= p{\cdot}s$) iterations, where the k-th iteration works on a trailing submatrix that starts from the k-th tile column. Since all iterations apply the same operations to A's trailing submatrices recursively, Fig. 5 only shows the operations of the first iteration.

We list the kernels called by Algorithm 1 as follows:

- POTF2'(A_{tk}, L_{tk}): Given a matrix A_{tk} of $m \times n$ and $m \geq n$, we let $A_{tk} = \binom{A_{tk1}}{A_{tk2}}$ such that A_{tk1} is of $n \times n$, and A_{tk2} is of $(m - n) \times n$. We also let $L_{tk} = \binom{L_{tk1}}{L_{tk2}}$. POTF2' computes $\binom{L_{tk1}}{L_{tk2}}$ by solving $L_{tk1} =$ Cholesky(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 $A_{ij} = A_{ij} - L_{ik} L_{jk}^{T}$.

3.3 Heterogeneous Tile QR Factorization

Algorithm 2 shows our heterogeneous tile QR factorization. It is able to utilize the same kernels as those used by the tile QR factorization [12], except that each kernel requires an additional GPU implementation. For completeness, we list them briefly here:

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

Similar to the Cholesky factorization, Fig. 6 illustrates the operations of the heterogeneous tile QR factorization. It shows a matrix of three tile rows by six tile columns. Again the algorithm goes through six iterations for the six tile columns. Since every iteration performs the same operations on a different trailing submatrix, Fig. 6 shows the operations of the first iteration.

3.4 Two-Level Block Cyclic Distribution

We divide a matrix A into $p \times (s{\cdot}p)$ rectangular tiles using the two-level partitioning scheme (in Section 3.1), which first partitions A into $p \times p$ large tiles, then partitions each large

Algorithm 2 Heterogeneous Tile QR Factorization

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

tile into s rectangular tiles. On a hybrid CPU and GPU machine, we allocate the $p \times (s \cdot p)$ tiles 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 device P_x, where P_0 denotes the host system and $P_{x \geq 1}$ denotes 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, those columns whose indices are multiples of s are mapped to the P GPUs in a cyclic way, and the remaining columns go to all CPU cores on the host. Note that all CPU cores share the same set of small tiles assigned to the host, but each of them can pick up any small tile and compute on it independently.

Figure 7 (a) illustrates a matrix that is divided into hybrid tiles with the two-level partitioning scheme. Since we always map an entire tile column to a device, the figure omits the boundaries between rows. Figure 7 (b) displays how a matrix with twelve tile columns can be allocated to one host and three GPUs using the heterogeneous 1-D column block cyclic distribution. The ratio of the s-1 small tiles to their remainder controls the workload on the host and GPUs.

3.5 Communication Cost

We consider the hybrid CPU/GPU 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 [9].

On a system with P GPUs, and given 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 is equal to:

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

which reaches the lower bound of $\Omega(\frac{n^2}{\sqrt{P}})$ [20] to within a factor of $\sqrt{P} \log(P)$. We can also use a 2-D block cyclic distribution instead of the 1-D block cyclic distribution to attain the same communication volume as ScaLAPACK (i.e., $O(\frac{n^2}{\sqrt{P}} \log P)$ [7, 14]). The reason we use the 1-D distribution method is because it can result in less messages for the class of tile algorithms, and keep load balancing between processes as long as P is small. For distributed GPUs for which P is large, we indeed use a 2-D block cyclic distribution method.

The number of messages sent or received by at least one process in the heterogeneous tile QR (or Cholesky) factorization is equal to:

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

The number of messages is greater than that of ScaLAPACK [7, 14] by a factor of $O(p)$, but the heterogeneous tile algorithms have a much smaller message size and exhibit a higher degree of parallelism. Note that we want to have a higher degree of parallelism in order to reduce synchronization points and hide communications particularly on many-core systems [3, 4, 11].

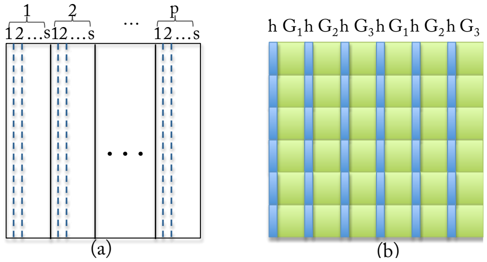


Figure 7: Heterogeneous 1-D column block cyclic data distribution. **(a)** The matrix A divided by a two-level partitioning method. (p, s) determines a matrix partition. **(b)** Allocation of a matrix of 6×12 rectangular tiles (i.e., p=6, s=2) to a host and three GPUs: `h`, `G`$_1$, `G`$_2$, and `G`$_3$.

3.6 Tile Size Tuning

Load imbalance could happen either between different GPUs, or between the host and each GPU. We use a 1-D block cyclic distribution method to achieve load balancing between different GPUs. Meanwhile we adjust the ratio of CPU tile to GPU tile to achieve load balancing between the host and the GPUs. We go through the following three steps to determine the best tile sizes to attain high performance:

1. We use the two-level partitioning scheme to divide a matrix assuming that we have known the top-level large tile size B (later we show how to decide 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_{core} and Perf_{gpu} denote the maximum performance of a dominant computational kernel (in Gflops) of the algorithm on a CPU core or on a GPU, respectively.

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 the 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 execution 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 decreases 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 all our experiments, it took at most three attempts to find B_h^*.

The top-level tile of 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 (e.g., `GEMM` for Cholesky and `SSRFB` for QR). Our search ranges from 128 to 2048 and is performed only once for every new kernel library and every new GPU architecture.

Unlike Step 1, Steps 2 and 3 depend on the number of CPU cores and number of GPUs used in a computation. Note that there are at most ($\#Cores \cdot \#GPUs$) configurations on a given machine, but not every configuration is useful in practice (e.g., we often use all CPU cores and all GPUs in scientific computing applications). Our experimental results (Fig. 12) show that the auto-tuning method can keep the system load imbalance under 5% in most cases.

4. THE RUNTIME SYSTEM

We allocate a matrix's tiles to the host and multiple GPUs statically with the two-level block cyclic distribution method. After the data are distributed, we further require that a task that modifies a tile be executed by the tile's owner (either host or GPU) to reduce data transfers. In other words, the allocation of a task is decided by the task's output location.

Although we use a static data and task allocation, the execution order between tasks within the host or GPU is decided at runtime. For instance, on the host, a CPU core will pick up a high-priority ready task to execute whenever it becomes idle. We design a runtime system to support dynamic scheduling, automatic data-dependency solving, and transparent data transfers between devices. The runtime system follows the data-flow programming model, and drives the task execution by data-availability.

4.1 Data-Availability-Driven Execution

A parallel program consists of a number of tasks (or instances of compute kernels). It is the runtime system's responsibility to identify the data dependencies between tasks. Whenever identifying a parent-child relationship, it is also the runtime system's responsibility to transfer the parent's output data to the child.

Our runtime system keeps track of generated tasks in a list. The information of a task contains the input and output location of the task. By scanning the task list, the runtime system is able to find out which tasks are waiting for the finished task based on their input and output information. For instance, after a task is finished and has modified a tile [I,J], the runtime system searches the list for the tasks who want to read the tile [I,J]. Those tasks that were just found will be the children of the finished task. For instance, Fig. 8 shows that tasks 1-3 are waiting for the completion of task 0, and will be fired after task 0 is finished. In our runtime system, we use a hash table to speed up the matching between tasks by using [I,J] as hash keys.

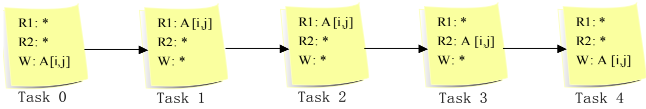


Figure 8: An example of solving data dependencies between tasks.

In essence, by solving data dependencies, our runtime system is gradually unrolling or constructing a DAG (Directed Acyclic Graph) for a user program. However, we never store the whole DAG or create it in one shot because whenever a task is finished, our runtime system can search for its children and trigger them on-the-fly dynamically. In addition, users do not need to provide communication codes to the runtime system, since the runtime system knows where the tasks are located based on the static allocation method, and will send data from the parent task to its children before actually notifying the children. This is why we call the scheme "data-availability-driven".

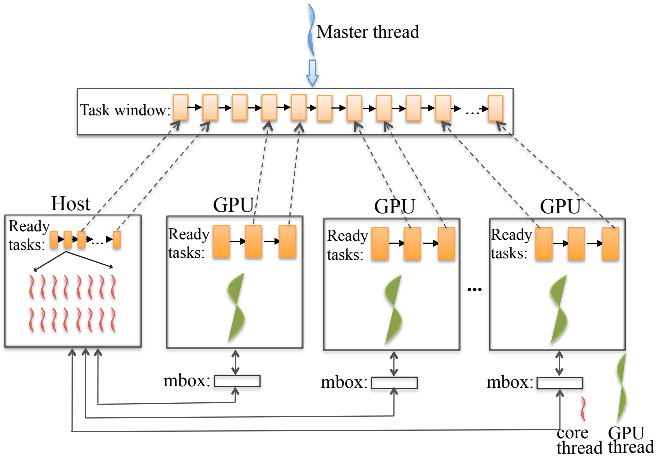


Figure 9: The runtime system for heterogeneous multicore and multi-GPU systems.

4.2 System Infrastructure

We extend the centralized-version runtime system of our previous work [29] to a new runtime system that is suitable for heterogeneous multicore and multi-GPU systems. The centralized-version runtime system works only on multicore architectures, and consists of four components (see Fig. 9 as if it had no GPUs):

- Task window: a fixed-size task queue that stores all the generated but unfinished tasks. It is an ordered list that keeps the serial semantic order between tasks. This is where the runtime system scans or searches for the children of a finished task.
- Ready task queue: a list of tasks whose inputs are all available.
- Master thread: a single thread that executes a serial program, and generates and inserts new tasks to the task window.
- Computational (core) threads: there is a computational thread running on every CPU core. Each computational thread picks up a ready task from the shared ready task queue independently whenever it becomes idle. After finishing the task, it scans the task window to determine which tasks are the children of the finished task, and fires them.

Figure 9 shows the architecture of our extended runtime system. Note that the master thread and the task window have not been changed. However, since the host and the GPUs own disjoint subsets of the matrix data and tasks, we allocate to the host and each GPU a private ready task queue to reduce data transfers. If a ready task modifies a tile that belongs to the host or a GPU, it is added to the host or the GPU's ready task queue accordingly.

We have also extended the computational threads. The new runtime system has two types of computational threads:

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 4.0, MAGMA
Compilers	Intel compilers 11.1	CUDA toolkit 4.0
OS	CentOS 5.5	Driver 270.41.19

one for CPU cores and the other for GPUs. If a multicore and multi-GPU system has P GPUs and n CPU cores, the runtime system will launch P computational threads to represent (or manage) the P GPUs, and $(n-P)$ computational threads to represent the remaining CPU cores. A GPU computational thread is essentially a GPU management thread, which is running on the host but can invoke GPU kernels quickly. For convenience, we think of the GPU management thread as a powerful GPU computational thread.

We also create a message box for each GPU computational (or management) thread in the host memory. When moving data either from host to a GPU, or from a GPU to host, the GPU computational (or management) thread will launch the corresponding memory copies. In our implementation, we use GPUDirect V2.0 to copy data between different GPUs.

4.3 Data Management

It is the runtime system's responsibility to move data between the host and GPUs. The runtime system will send a tile from a parent task to its children transparently when the tile becomes available. However, the runtime system does not know how long the tile should persist in the destination device. Similar to the ANSI C function `free()`, we provide programmers with a special routine `Release_Tile()` to free tiles. `Release_Tile()` does not release any memory, but sets up a marker in the task window. While adding tasks, the master thread keeps track of the expected number of visits for each tile. Meanwhile the computing thread records the actual number of visits for each tile. The runtime system frees a tile if and only if: i) the actual number of visits is equal to the expected number of visits to the tile, and ii) `Release_Tile` has been called to free the tile. In our runtime system, each tile maintains three data members to support the dynamic memory deallocation: `num_expected_visits`, `num_actual_visits`, and `is_released`.

5. PERFORMANCE EVALUATION

We conducted experiments on a single node of the Keeneland system [31] at the Oak Ridge National Laboratory. The Keeneland system has 120 nodes and each node has two Intel Xeon 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` that are used by Cholesky and QR factorizations, respectively. In addition, we show our experiments with the Cholesky factorization using up to 100 nodes.

5.1 Weak Scalability

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

Figure 10 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 and Gflops-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 CPU core or from a GPU, one "`max GPU-kernel`" to denote the maximum GPU kernel performance of the Cholesky or QR factorization which is the upper bound of the program, "`our perf per GPU`" to denote our program performance on each GPU, and "`our perf per core`" to denote our program performance on each CPU core.

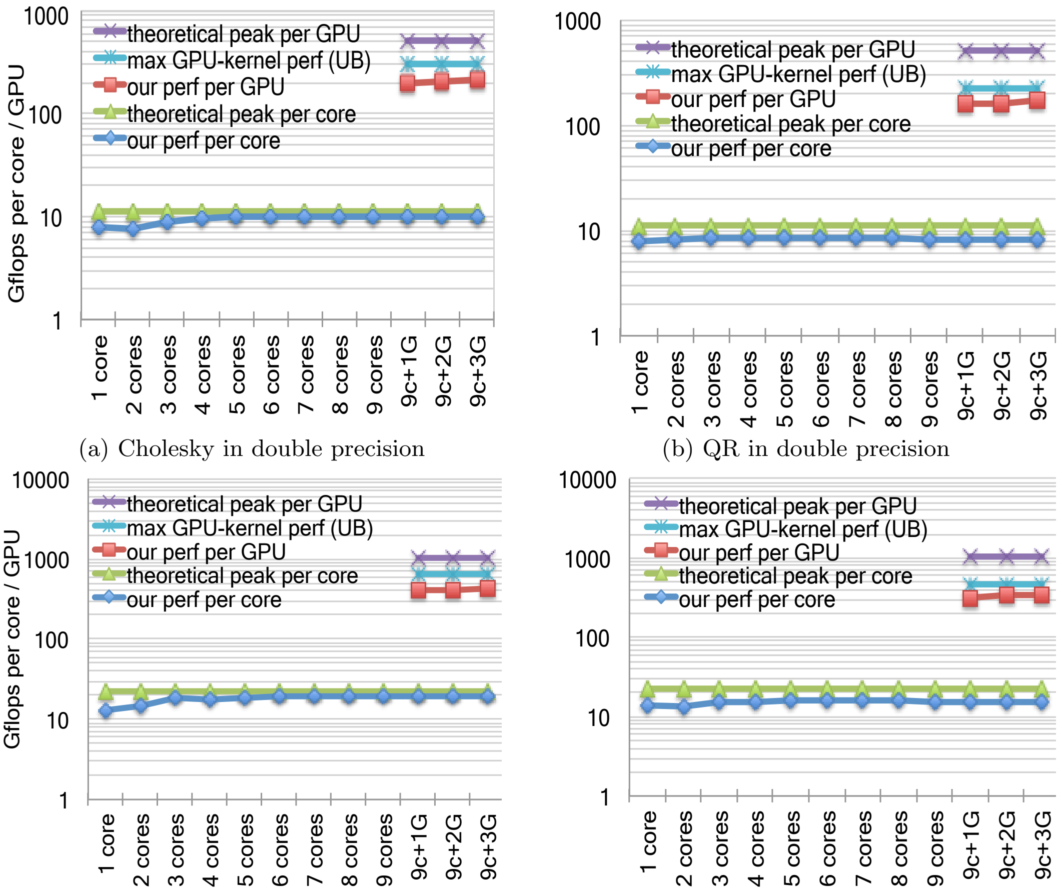
In the experiments, we first increase the number of cores from one to nine. Then we add one, two, and three GPUs to the nine cores. The input sizes for the double precision experiments (i.e., (a), (b)) are: 1000, 2000, . . . , 9000, followed by 20000, 25000, and 34000. The input sizes for single precision (i.e., (c), (d)) are the same except for the last three input sizes which are 30000, 38000, and 46000. Figure 10 shows that our Cholesky and QR factorizations are scalable on both CPU cores and GPUs. Note that performance per core or performance per GPU should be a flat line ideally.

The overall performance of Cholesky factorization and QR factorization can be derived by summing up (perf-per-core × NumberCores) and (perf-per-gpu × NumberGPUs). For instance, the double precision Cholesky factorization using nine cores and three 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 delivers an overall performance of 1.44 Tflops, which is 69% of the upper bound and 44% of the theoretical peak. On the other hand, 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 our program can solve a given problem if we are provided with additional computing resources. In the experiment, we measure the wall clock execution time to solve a number of matrices. Given a fixed-size matrix, we keep adding more computing resources to solve it.

Figure 11 shows the wall clock execution time of Cholesky and QR factorizations in both double and single precisions. Each graph has four or five curves, each of which corresponds to a matrix of a fixed 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. 11 (a), we can 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 can reduce the execution time of QR factorization



(a) Cholesky in double precision

(b) QR in double precision

(c) Choleksy in single precision

(d) QR in single precision

Figure 10: Weak scalability. The input size increases too while adding more cores 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.

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 performances for the single precision experiments. 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 Balancing

We employ the metric `imbalance_ratio` to evaluate the quality of our load balancing, where `imbalance_ratio` = $\frac{MaxLoad}{AvgLoad}$ [22]. We let computational time 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.

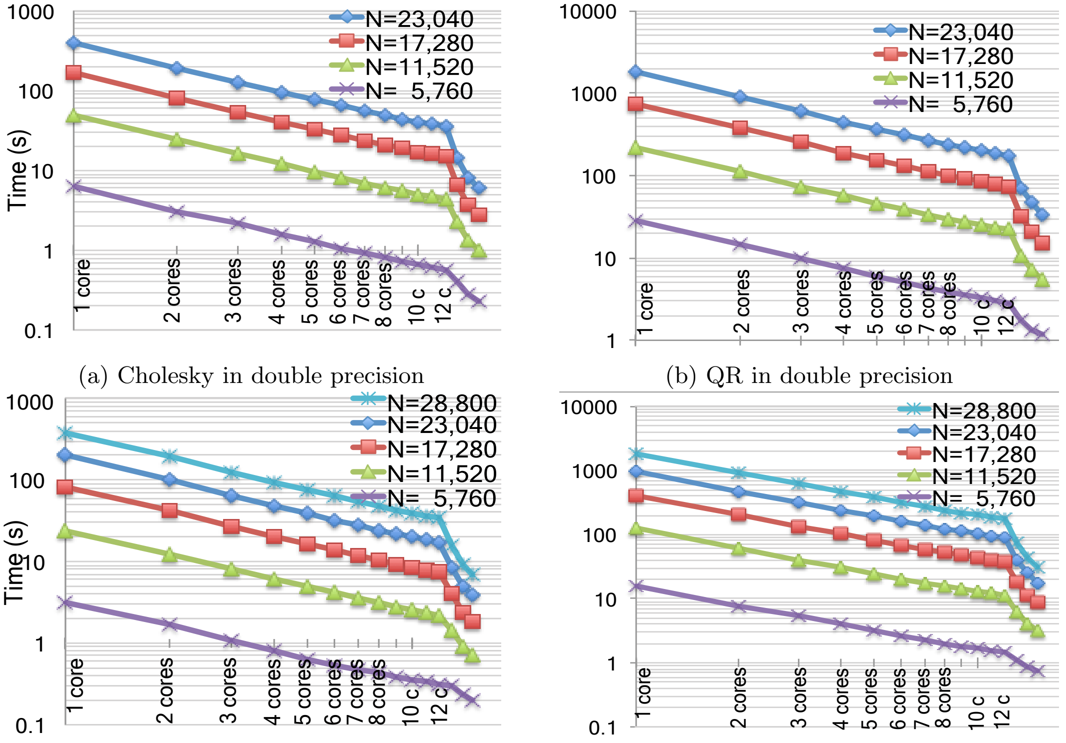
Our experiments use three different configurations as examples: 3 cores + 1 GPU, 6 cores + 2 GPUs, and 9 cores + 3 GPUs. Given an algorithm, we first determine the top-level tile size, B, for the algorithm; then we determine the partitioning size, B_h^*, for each configuration using our autotuning method. 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 the number of tile columns is not divisible by the number of GPUs, we can divide its remainder ($\leq$ NumberGPUs-1) among all GPUs using a smaller chunk size.

Figure 12 shows the measured imbalance ratio for double and single precision matrix factorizations on three configurations. An imbalance ratio of 1.0 indicates a perfect load balancing. We can see that most of the imbalance ratios are within 5% of the optimal ration 1.0. A few of the first columns have an imbalance ratio that is within 17% of the optimal ratio. This is because their corresponding matrices have too few top-level tiles. For instance, the first column of the `9Cores+3GPUs` configuration (Fig. 12 (c), (f)) has a matrix of three top-level tiles for three GPUs. We could increase the number of tiles to alleviate this problem by reducing the top-level tile size.

5.4 Applying to Distributed GPUs

Although this paper is focused on a shared multicore and multi-GPU system, we are able to apply our approach to distributed GPUs directly.

Given a cluster with lots of multicore and multi-GPU nodes, we first divide a matrix into $p \times p$ large tiles. Next, we distribute the large tiles to the nodes in a 2-D block cyclic way. Finally, after each node is assigned a subset of the large tiles, we partition each of the local large tiles into s-1



(a) Cholesky in double precision

(b) QR in double precision

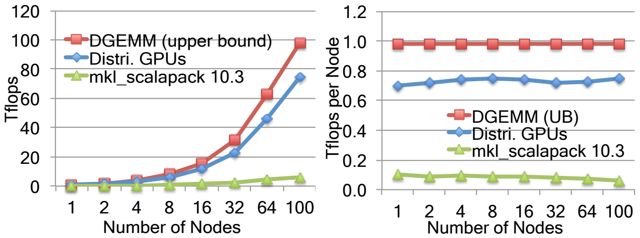
(c) Cholesky in single precision

(d) QR in single precision

Figure 11: Strong scalability. An ideal strong-scalability curve were to be a straight line in a log-log graph. The last three ticks on the x-axis (after `12c`) 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 the x-axis and the y-axis are presented on a logarithmic scale.

small rectangular tiles and one large rectangular tile, and map them to the node's host and multiple GPUs using the two-level block cyclic distribution (in Section 3.4).

Our experiments with Choleksy factorization demonstrate that our approach can also scale efficiently on distributed GPUs. Figure 13 shows the weak scalability experiments on Keeneland using one to 100 nodes, where each node uses 12 cores and 3 Fermi GPUs. The single-node experiment takes as input a matrix of size 34,560. If an experiment uses P nodes, the matrix input size is $\sqrt{P} \times 34560$. The overall performance on 100 nodes reaches 75 Tflops (i.e., 45% of the theoretical peak). Also the workload difference on 100 nodes between the most-loaded node and the least-loaded node is just 5.2%. For reference, we show the performance of the Intel MKL ScaLAPACK library that uses CPUs only.



(a) Overall Performance

(b) Per-Node Performance

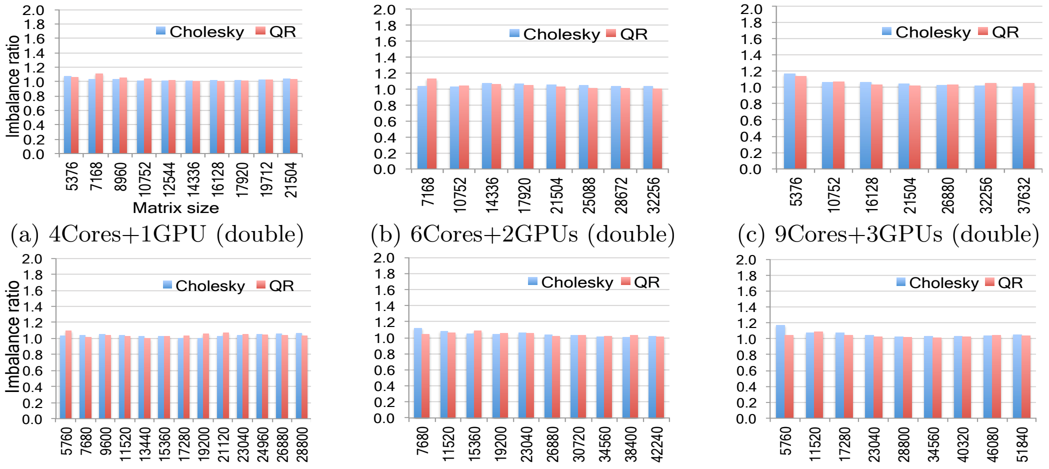
Figure 13: Weak scalability of the distributed-GPU Cholesky factorization (double precision) on Keeneland.

6. RELATED WORK

There are a few dense linear algebra libraries developed for GPU devices. CUBLAS has implemented the standard BLAS (basic linear algebra subroutines) library on GPUs [26]. CULA [19] and MAGMA [30] have implemented a subset of the standard LAPACK library on GPUs. However, their current releases do not support computations using both CPUs and GPUs.

Quintana-Ortí et al. adapted the SuperMatrix runtime system to shared-memory systems with multiple GPUs [27]. They also recognized the communication bottleneck between host and GPUs, and designed a number of software cache schemes to maintain the coherence between the host RAM and the GPU memories to reduce communication. Fogue et al. presented a strategy to port the existing PLAPACK library to GPU-accelerated clusters [16]. They require that GPUs take most of the computations and store all data in GPU memories to minimize communication. Differently, we distribute a matrix across host and GPUs, and can utilize all CPU cores and all GPUs.

StarSs is a programming model that uses directives to annotate a sequential source code to execute on various architectures such as SMP, CUDA, and Cell [6]. 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. Charm++ is an object-oriented parallel language that uses a dynamic load balancing runtime system to map objects to processors dynamically [21]. StarPU develops a dynamic load bal-



(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 12: Load imbalance. The metric $imbalance\_ratio = \frac{MaxLoad}{AvgLoad}$. The closer the ratio is to 1.0, the better.

ancing framework to execute a sequential code on host and GPUs in parallel, and has been applied to Cholesky and QR factorizations [2, 1]. By contrast, we use a simple static distribution method to minimize communication and keep load balancing to attain high performance.

Fatica has implemented the Linpack Benchmark for GPU clusters by splitting the trailing submatrix into a single GPU and one CPU socket statically [15]. Yang et al. introduces an adaptive partitioning technique to split the trailing submatrix into a single GPU and individual CPU cores to implement Linpack [32]. Qilin is a generic programming system that can automatically map computations to GPUs and CPUs through off-line trainings [24]. Ravi et al. design a dynamic work distribution scheme for the class of Map-Reduce applications [28]. Differently, we use a heterogeneous multi-level 2D block cyclic method to distribute work to multiple GPUs, multiple CPU cores, and on many nodes.

There are also many researchers who have studied how to apply static data distribution strategies to heterogeneous distributed memory systems. Dongarra et al. designed an algorithm to map a set of uniform tiles to a 1-D collection of heterogeneous processors [10]. Robert et al. proposed a heuristic 2-D block data allocation to extend ScaLAPACK to work on heterogeneous clusters [8]. Lastovetsky et al. developed a static data distribution strategy that takes into account both processor heterogeneity and memory heterogeneity for matrix factorizations [23].

7. CONCLUSION AND FUTURE WORK

Designing new software on heterogeneous multicore and multi-GPU architectures is a challenging task. In this paper, we present heterogeneous algorithms with hybrid tiles to solve a class of dense matrix problems that have affine loop structures. We treat the multicore and multi-GPU system as a distributed-memory machine, and deploy a heterogeneous multi-level block cyclic data distribution to minimize communication. We introduce an auto-tuning method to determine the best tile sizes. We also design a new runtime system for the heterogeneous multicore and multi-GPU architectures. Although we have applied our approach to matrix computations, the same methodology and principles such as heterogeneous tiling, multi-level partitioning and distribution, synchronization-reducing, distributed-memory perspective on GPUs, auto-tuning, are general and can be applied to many other applications (e.g., sparse matrix problems, image processing, quantum chemistry, partial differential equations, and data-intensive applications).

Our future work is to apply the approach to heterogenous clusters with an even greater number of multicore and multi-GPU nodes, and use it to build scientific applications such as two-sided factorizations and sparse matrix solvers. In our current approach, the largest input is constrained by the memory capacity of each GPU. A method to solve this issue is to use an “out-of-core” algorithm such as the left looking algorithm to compute matrices panel by panel [13].

Acknowledgments

We are grateful to Bonnie Brown and Samuel Crawford for their 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 *IPDPS 2011*, Alaska, USA, 2011.

[2] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, J. Roman, S. Thibault, and S. Tomov. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing*, Knoxville, USA, 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 *SC'09*, pages 20:1–20:12, 2009.

[4] 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.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper., Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[6] 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, 2009.

[7] 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, 2009.

[8] 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.

[9] 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.

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

[11] 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, 2007.

[12] 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.

[13] E. D'Azevedo and J. Dongarra. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurrency: Practice and Experience*, 12(15):1481–1493, 2000.

[14] 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.

[15] M. Fatica. Accelerating Linpack with CUDA on heterogenous clusters. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 46–51, 2009.

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

[17] 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, 2008.

[18] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15:1201–1226, September 1994.

[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, 2010.

[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] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

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

[26] NVIDIA. CUDA Toolkit 4.0 CUBLAS Library, 2011.

[27] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 121–130, 2009.

[28] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, 2010.

[29] 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, 2009.

[30] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA Users' Guide. Technical report, ICL, UTK, 2011.

[31] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science Engineering*, 13(5):90 –95, sept.-oct. 2011.

[32] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19 –28, sept. 2010.