A Scalable Framework for Heterogeneous GPU-Based Clusters *

(Regular Submission)

Fengguang Song
Innovative Computing Laboratory
University of Tennessee
Knoxville TN 37996-3450
song@eecs.utk.edu

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

Abstract

GPU-based heterogeneous clusters continue to draw attention from vendors and HPC users due to their high energy efficiency and much improved single-node computational performance, however, there is few parallel software that can utilize all CPU cores and all GPUs on the heterogeneous system efficiently. On a heterogeneous cluster, the performance of a GPU (or a compute node) increases in a much faster rate than the performance of the PCI-Express connection (or the interconnection network) such that communication eventually becomes the bottleneck of the entire system. To overcome the bottleneck, we developed a multi-level partitioning and distribution method that guarantees a near-optimal communication volume. We have also extended heterogeneous tile algorithms to work on distributed-memory GPU clusters. Our main idea is to execute a serial program and generate hybrid-size tasks, and follow a dataflow programming model to fire the tasks on different compute nodes. We devised a distributed dynamic scheduling runtime system to schedule tasks, and transfer data between hybrid CPU-GPU compute nodes transparently. The runtime system employs a novel distributed task-assignment protocol to solve data dependencies between tasks without coordination between processing units. The runtime system on each node consists of a number of CPU compute threads, a number of GPU compute threads, a task generation thread, an MPI communication thread, and a CUDA communication thread. By overlapping computation and communication through dynamic scheduling, we are able to attain a high performance of 75 TFlops for Cholesky factorization on the heterogeneous Keeneland system [23] using 100 nodes, each with twelve CPU cores and three GPUs. Moreover, our framework can also deliver high performance on distributed-memory clusters without GPUs, and shared-system multiGPUs.

keywords: Dense linear algebra, distributed runtime systems, hybrid CPU-GPU, heterogeneous clusters.

*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.

1 Introduction

Based on the November 2011 Green500 list [20], twenty-three out of the top thirty greenest supercomputers are GPU-based. However, there is few software that can take advantage of the large-scale heterogeneous systems efficiently, especially to utilize all CPU cores and all GPUs. Considering many operations of scientific computing applications are carried out through numerical linear algebra libraries, we focus on providing fundamental linear algebra operations on the new heterogeneous architectures.

A great amount of effort has gone into the implementation of linear algebra libraries. LAPACK [5], Intel MKL, AMD ACML, and PLASMA [4] are mainly designed for shared-memory multicore machines. ScaLAPACK [10] is intended for distributed memory CPU-based machines. CUBLAS [18], MAGMA [22], and CULA [15] provide a subset of the LAPACK subroutines but work on a single GPU. So far these libraries do not support computations using multiple CPU cores and multiple GPUs on a single node, not to mention distributed GPU-based clusters. Moreover, with an increasing number of cores on the host whose performance continues to keep up with the GPU speed, new parallel software should not ignore either GPUs or CPUs.

Our work aims to provide a unified framework to solve linear algebra problems on any number of CPU cores, any number of GPUs, and for either shared-memory or distributed-memory systems. Our solution consists of three essential components: (1) a static multi-level data distribution method, (2) heterogeneous tile algorithms, and (3) a distributed dynamic scheduling runtime system. The solution works as follows. Given a matrix input, we first split it into tiles of hybrid sizes. Then we distribute the tiles to host main memories and GPU device memories on a cluster with a static method. Each compute node runs a runtime system (launched as an MPI process) that schedules tasks within the node dynamically. Different nodes communicate with each other by means of MPI messages. Our runtime system follows the data-flow programming model and builds a partial directed acyclic graph (DAG) dynamically, where a completed task will trigger a set of new tasks in the DAG.

We use a static multi-level distribution method to allocate data to different hosts and GPUs. Each compute node is heterogeneous since it has both CPUs and GPUs, but different nodes have the same performance. Therefore, we design a multi-level distribution method. On the top (i.e., inter-node) level, we use a 2-D block cyclic distribution method. On the second (i.e., intra-node between different GPUs) level, we allocate a node's local blocks to merely GPUs with a 1-D or 2-D block cyclic method. On the third (i.e., intra-node between CPUs and GPUs) level, we cut a slice from each GPU's local block and put it to the host. The output of the multi-level distribution method is that each matrix block is uniquely assigned to the host or a GPU on a specific node.

We also use heterogeneous tile algorithms to handle the difference between CPUs and GPUs [21]. In the algorithm, there are a great number of small tasks for CPU cores, and a great number of large tasks for GPUs, to compute concurrently, at any time. While a heterogeneous tile algorithm is based on tiles, it has two types of tiles: small ones for CPU cores, and large ones for GPUs. Our work combines the heterogenous tile algorithms and the multi-level distribution method together so that the algorithms are applicable to heterogeneous clusters with hybrid CPUs and GPUs.

We design a distributed scheduling runtime system for heterogeneous clusters. Each compute node is executing a runtime system that can solve data dependencies dynamically, and transfer data from a parent task to its children transparently. All runtime systems (one per node) proceed in parallel, and execute a task-assignment protocol to build subsets (or partitions) of a DAG dynamically. There is no communication required when building the DAG. The protocol guarantees that all runtime systems make a unanimous decision without coordinating with each other such that every task is computed by one and only one processing unit (on a host or a GPU).

Our experiments with double-precision Cholesky and QR factorizations, on the heterogeneous Keeneland system [23] at the Oak Ridge National Laboratory, demonstrate great scalability from one to 100 nodes using all CPUs and GPUs. In addition, we apply our framework to the other two possible environments: clusters without GPUs, and a shared system with CPUs and multiple GPUs. Compared with vendor-optimized and open source libraries (i.e., In-

tel MKL, and StarPU [6]), our framework is able to provide better performance thank Intel MKL by up to 66% on clusters without GPUs, and up to 250% better performance than StarPU on shared-system multiGPUs.

2 Background

We place greater emphasis on communications in our framework design. On a host that is attached with multiple GPUs through PCI-Express connections, the ratio of computation to communication on the GPUs keeps increasing. Eventually the communication time on a PCI-Express connection will become the bottleneck of the entire system.

Researchers most often use dynamic scheduling methods to support heterogeneous systems, where each CPU core or GPU picks up a ready task from task queues independently whenever a processing unit becomes idle. At the beginning of our design, we have also implemented dynamic scheduling runtime systems. In our dynamic runtime system, all the CPU cores and GPUs share a global ready task queue, and each GPU owns a software cache on its device memory. Whenever a GPU reads a block of data from the host, it stores the data to its software cache. All the data in the GPUs' software caches are also backed up by the main memory on the host. We have used two cache writing policies: write-through and write-back. With the write-through policy, every modification to the software cache must be updated to the host main memory immediately. With the write-back policy, a modification to the software cache is updated to the host main memory only when a different device wants to access the modified data. To achieve the best performance, our software cache size on each GPU is configured as large as the input matrix size to eliminate capacity cache misses.

Figure 1 shows our experiments with the double-precision Cholesky factorization on a single node of the Keeneland system using 12 cores and 3 Fermi GPUs. In the figure, we compare our software-cache based dynamic scheduling runtime system, the generic dynamic scheduling runtime system of StarPU [6], and our distributed-GPUs framework that builds upon a static data distribution method. By changing from the write-through policy to the write-back policy, we can improve the program performance significantly because of the reduced communication.

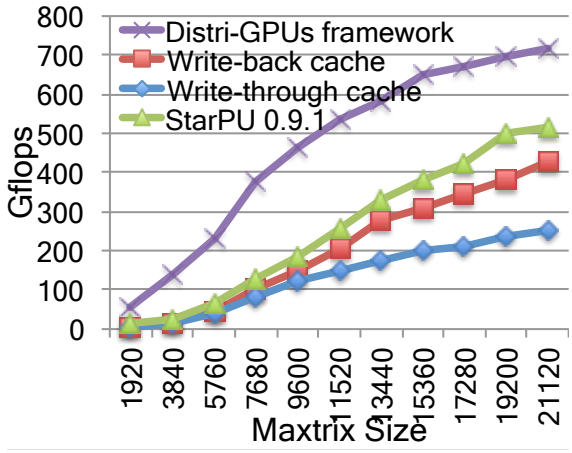


Figure 1: A comparison between dynamic scheduling systems and our distributed-GPU framework.

Differently, StarPU consists of profiling, performance modeling, and various scheduling policies to achieve load balancing and reduce data transfers. However, since our static data distribution method can guarantee a near lower-bound communication cost and has less scheduling overhead, it is faster than StarPU by up to 200% for small to medium matrix sizes. This has inspired us to use a static data distribution method on GPU-based clusters. Here we emphasize that despite its better performance, our framework is intended for solving linear algebra problems, while StarPU is more generic and can support other applications.

3 Heterogeneous Tile Algorithms

Our previous work has designed a class of heterogeneous tile algorithm and applied them to shared-system multiGPUs [21]. Here we mention it briefly for completeness. In the algorithm, every task takes several individual tiles as input and output. On a heterogeneous system with CPUs and GPUs, we create two different tiles.

Figure 2 shows a matrix that is stored in a tile data layout with two different tiles. All the tasks that modify small tiles are to be executed by CPU cores, and those that modify large tiles are to be ex-

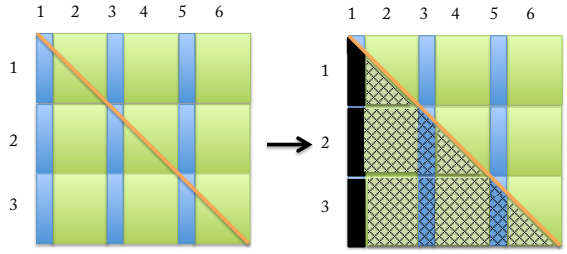


Figure 2: An example of the heterogeneous tile algorithm for Cholesky factorization.

ecuted by GPUs. Given a matrix with 3 tile rows and 6 tile columns, Cholesky factorization can be computed recursively as follows. At the first iteration, (1) we solve Cholesky factorizations on tiles A_{11}, A_{21}, and A_{31} in the first column. That is, $L_{11} \leftarrow Cholesky(A_{11})$, and $L_{ij} \leftarrow A_{ij}(L_{11}^T)^{-1}$; (2) then we update the trailing submatrix located between the second and the last sixth column. That is, $A_{ij} \leftarrow A_{ij} - L_{i1}L_{j1}$. The Cholesky factorization can be completed by recursively applying the above two steps to the trailing submatrix that starts from the j-th tile column. If a matrix has n tile columns, the factorization takes n iterations to complete. The same idea can be applied to other matrix factorizations (e.g., QR and LU factorizations). For more details, please refer to our report [21] for the exact algorithms of Cholesky and QR factorizations.

3.1 Static Multi-Level Block Cyclic Data Distribution

This section presents a new multi-level partitioning scheme to create small and large tiles on distributed-memory heterogeneous clusters. (1) At the top level, we divide a matrix into $p \times p$ large tiles, each of which is of size $B \times B$. (2) We distribute the $p \times p$ large tiles to a $P_r \times P_c$ process grid using a 2-D block cyclic method. There is one process per node. (3) At the bottom level (i.e., within each node), we vertically cut every large tile of size $B \times B$ on each node into a number of $(s-1)$ small tiles of size $B \times b$, and one remaining large tile of size $B \times (B-(s-1)\cdot b)$. We allocate the small tiles to the entire set of CPU cores on the host, meanwhile allocate the remaining tiles to GPUs using a 1-D or 2-D block cyclic method. So far we use a 1-D method because of the small number of GPUs (i.e., ≤ 4) on each compute node.

After the multi-level block cyclic data distribution, each node is assigned to $\frac{p}{P_r} \times \frac{p \cdot s}{P_c}$ rectangular tiles. Given a tile indexed by [I, J], we first map it to node N_i then to device D_j, where D_0 denotes the host, and $D_{j \geq 1}$ denotes the j-th GPU located on node N_i. Assuming each node has G GPUs, we can compute node N_i ($0 \leq i \leq P_r P_c$-1), and device D_j as follows:

$$N_i = (I \mod P_r) \cdot P_c + (\frac{J}{s} \mod P_c),$$

$$D_j = \begin{cases} 0 & : (J \mod s) < s-1 \\ (\frac{J/s}{P_c} \mod G) + 1 & : (J \mod s) = s-1 \end{cases}$$

In other words, Step (2) distributes the large tiles across $P_r \times P_c$ nodes (*for* N_i). Next, within each node, the tile columns whose indices are multiples of $(s-1)$ are mapped to the node's G GPUs in a 1-D cyclic way, and the rest of the columns are mapped to all CPUs on the host (*for* D_j). Although small tasks are assigned to all the CPUs, each CPU core can pick up any small task independently (i.e., not in a fork-join manner). We also tune the tile sizes of B and b to achieve the highest performance. Appendix A.1 describes how we choose the best tile sizes.

4 Our Framework Overview

We design a distributed dynamic scheduling runtime system for the heterogeneous GPU-based clusters. Given a cluster with P nodes, we launch P MPI processes (one process per node), each of which executes an instance of the runtime system. We also assume a matrix is stored in the hybrid tile data layout that uses two different tile sizes.

Not only do we distribute data to hosts and GPUs on different nodes statically, but also we distribute tasks to hosts and GPUs statically. We require that the location of a task be the same as the location of the task's output. Although a task's allocation is static, we schedule tasks dynamically within a host or GPU, in order to reduce synchronization points and to overlap computation with communication.

Our runtime system follows a dataflow programming model and is essentially data-availability driven. Whenever a parent task completes, it triggers its child tasks immediately. The runtime system can identify data dependencies between tasks and unroll

a Directed Acyclic Graph (DAG) dynamically. Note that a DAG has never been created and stored in our runtime system explicitly. A parallel program starts with an entry task and finishes with an exit task of the DAG, respectively. The runtime system is also responsible for transferring data from a parent task to its children transparently.

Each runtime system instance is multi-threaded. It creates five types of threads: a task-generation thread, a set of CPU compute threads for CPU cores, a set of GPU management threads for GPUs, an inter-node MPI communication thread, and an intra-node CUDA communication thread. If a machine is not distributed, the MPI communication thread is not created. Similarly, if there is no GPU, the CUDA communication thread is not created.

A task-generation thread create tasks (similar to CPUs issuing instructions) and drives the execution of a parallel program. There are actually P task-generation threads running on P nodes. All the task-generation threads execute the same sequential code independently and create task instances for the program without communication. They execute a distributed task-assignment protocol. Based on the common knowledge of the static multi-level distribution, each task generation thread is able to decide by itself which task it should compute and where the task's children are located without exchanging any messages.

5 The Framework Implementation

As shown in Fig. 3, our runtime system consists of seven components listed as follows. (1) Task window: a fixed-size task queue that stores all the generated but not finished tasks. (2) Ready task queues: a few lists of ready tasks. (3) Task-generation thread: a single thread that executes a serial program, and generates new tasks. (4) CPU compute threads: there is a compute thread running on every CPU core. (5) GPU management (or compute) threads: there is a GPU management thread for each GPU. (6) MPI communication thread: a single thread that transfers data between different nodes. (7) CUDA communication thread: a single thread that transfers data among the host and multiple GPUs within the same node using `cudaMemcpyAsync`.

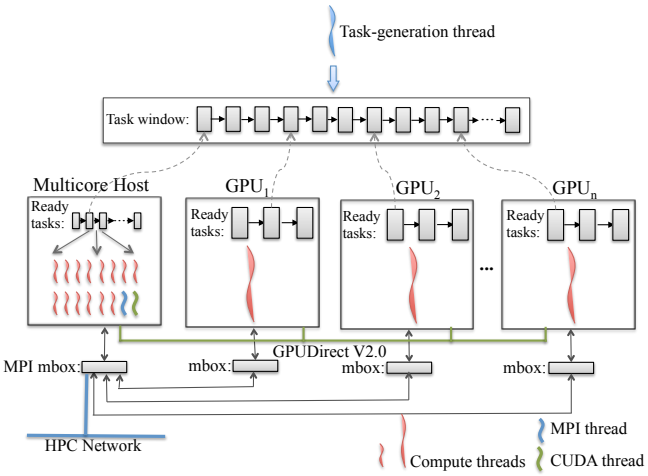


Figure 3: Architecture of the distributed-GPU runtime system on each node.

5.1 Different Task Queues

A *task window* stores generated tasks in a single-linked list. It also keeps the original sequential order between tasks. Each task consists of the information of a task's input and output. Based on the input and output information, when a task modifies its output, the runtime system scans the list to search for the tasks who are waiting for the output. However, a global task list is often too long to search and can result in severe contention between threads.

To make accesses to the task window faster, we use 2-D task lists to implement the task window. As illustrated in Fig. 4, each tile in a matrix has its own task list. If a task's input or output is tile [I, J], the runtime system will add an instance of the task to [I, J]'s task list. When a matrix is distributed to different compute nodes, we partition the 2-D task lists into different nodes according to the location of the tiles. That is, if tile [I, J] is allocated to node P_k, tile [I, J]'s task list is also assigned to node P_k.

A *ready task queue* stores "ready-to-go" tasks whose inputs are all available. If a task modifies a tile that belongs to the host or a GPU, it is added to that host or GPU's ready task queue correspondingly. We don't allow host and GPUs to steal tasks from each other to avoid unnecessary data transfers and to increase data reuse. In addition, a ready task in our implementation is simply a pointer that points to a task stored in the task window.

5.2 Solving Data Dependencies

A tile's task list maintains the serial semantic order between tasks that either read or write the tile. Whenever two tasks access the same tile and one of them is write, the runtime system detects a data dependency and stalls the successor till the predecessor is finished. Here we only consider the true dependency RAW (read after write), and use renaming to avoid the WAR (write after read) and WAW (write after write) dependencies. Figure 5 shows an example of a task list that is used for tile A[i, j], where tasks 1-3 are waiting for task 0's output.

There are only two operations to access a task list: `FIRE` and `APPEND`. After a task completes and modifies its output tile [I, J], the `FIRE` operation searches [I, J]'s task list for the tasks that want to read [I, J]. It scans the list from the just completed task to the end to find which tasks are waiting for [I, J]. The scan process will exit when confronting a task that wants to write to [I, J]. If a task is visited before the scan process exits, the FIRE operation marks the task's input as "ready". When all the inputs of a task become ready, the task evolves into a ready task.

The `APPEND` operation is invoked by the task-generation thread. After generating a new task, the generation thread inspects every input and output of the task. Given an input that reads tile [I, J], before appending the input task instance, APPEND scans [I, J]'s task list from the list head to find if there exists a task that writes to tile [I, J]. If none of the tasks writes to [I, J], the input task instance is marked as "ready". Otherwise, it is "unready". By contrast, given an output [I, J], APPEND puts an output task instance to the end of [I, J]'s task list immediately.

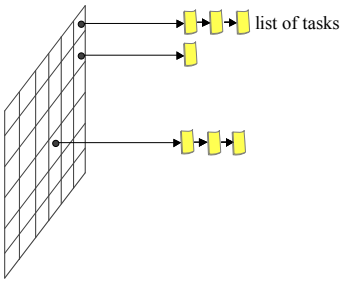


Figure 4: The 2-D task window implementation. Each tile has its own task list whose tasks either read or write the tile.

5.3 Computation Components

Each CPU core runs a CPU compute thread. Whenever a CPU compute thread becomes idle, it picks up a ready task from the host's shared ready task queue and computes it by itself. After finishing the task, the thread invokes the `FIRE` operation to determine which tasks are the children of the finished task, and moves them to a ready task queue if possible.

Similarly, each GPU has a GPU compute thread. A GPU compute thread is essentially a GPU management thread, which is running on the host but can start GPU kernels quickly. For convenience, we think of the GPU management thread as a powerful compute thread. If a node has g GPUs and n CPU cores, our runtime system launches g GPU compute threads to represent (or manage) the g GPUs, and $(n-g-2)$ CPU compute threads to represent the remaining CPU cores. The remaining number of CPU cores is not equal to $(n-g)$ because we use one core for MPI communication and another core for CUDA memory copies.

5.4 Communication Components

There are two types of communications on a GPU-based cluster: communication between nodes, and communication within a node. On each node, we create a thread to perform MPI operations to transfer data between nodes, and another thread to copy memories among the host and different GPUs on the same node.

The technique of CUDADirect V2.0 supports direct memory copies between GPUs on the same node. It may also send or receive GPU buffers on different nodes directly if an MPI library supports CUDADirect. To make our framework more portable, we choose to move data from GPU to host on the source node first, then send it to a destination node.

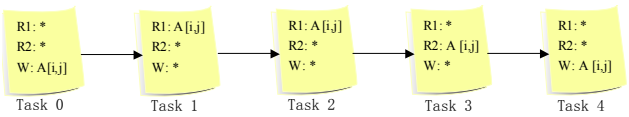


Figure 5: Solving data dependencies for a set of tasks that read or write tile A[i, j]. The completion of Task 0 will `fire` Tasks 1-3 that want to read A[i,j].

After the destination host receives the data, it copies the data from host to one or more of its GPUs.

An MPI communication thread runs on a dedicated CPU core. It calls nonblocking MPI point-to-point operations to send and receive messages. At the beginning, the thread posts an `MPI_Irecv` operation and an `MPI_Isend` operation. Next, it checks if the pending receive or send operation has finished with busy-polling. When an operation is finished, the thread posts a new operation to replace the finished one so that there are always two operations (one receive and one send) ongoing at the same time. Figure 9 in Appendix A shows the pseudocode to implement the MPI communication thread. In the code, `wait4send` and `wait4recv` indicate if there exists a pending send or receive operation.

A CUDA communication thread also uses a dedicated CPU core on the host. Each GPU has two mail boxes: `out_mbox` and `in_mbox`. The messages in an out_mbox are intended from the GPU to other devices, and the messages in an in_mbox are intended from other devices to the GPU itself. We create two streams for each GPU: one for outgoing traffic and the other for incoming traffic. Similar to the MPI communication thread, the CUDA communication thread tries to start one incoming memory copy and one outgoing memory copy for each GPU simultaneously. If there are a number of g GPUs, there will be $2g$ `cudaMemcpyAsync` operations happening concurrently in which each GPU has two operations. To implement the CUDA communication thread, `wait4send` and `wait4recv` have been changed to bitsets, where the i-th bit denotes the status of the i-th GPU. We have also implemented `select_GPU_streams` to substitute `MPI_Test` so that we can test in which GPU streams the asynchronous cudaMemcpy operations have finished.

5.5 Data Storage and Management

The host and each GPU have an indirect data structure to store matrices. Given a matrix with $p \times q$ tiles, the indirect data structure consists of $p \times q$ pointers each pointing to a tile. A pointer is null if the corresponding tile is not stored in the host or GPU. We store a GPU's indirect data structure to the host memory, but the pointers in the GPU's indirect structure actually point to GPU device memories. By using the indirect data structure, a GPU compute thread can simply look it up and pass correct arguments (i.e. GPU device pointers) to launch a GPU kernel.

Our runtime system transfers data from a parent task to its children transparently, however, it does not know how long the data should persist in the destination device. We provide programmers with a special function of `Release_Tile()` to free data. Release_Tile does not free any memory, but sets up a marker in the task window. The marker tells the runtime system that the tile will not be needed by the future tasks that are generated after the marker, and it is safe to free the tile whenever possible. When a programmer writes a sequential program, he or she can add Release_Tile() to the program just like calling the ANSI C function `free`. The task-generation thread keeps track of the expected number of visits for each tile. Meanwhile the compute threads count the actual number of visits for each tile. The runtime system will free a tile if and only if: i) Release_Tile has been called to mark the tile, and ii) the actual number is equal to the expected number of visits to the tile. In essence, this is an asynchronous deallocation method with which a dynamic runtime system can decide when it is safe to free data.

6 Distributed Task Assignment Protocol

Numerous runtime systems (one per node) execute the same code and generate the same set of tasks so that a task may be duplicated on each node. We design a protocol to guarantee that a task is computed by one and one one processing unit (CPU or GPU), and an input is sent to a waiting task only once.

Given a task with k_1 inputs, all the runtime systems across the cluster will generate k_1 input task instances in total. It is exactly k_1 instances because each input belongs to exactly one node and only that node will claim ownership of the input. Also we define that the first output of a task is the *main* output, and the rest outputs are *minor* outputs.

Our runtime system generates eight types of task instances using a set of rules. The rational behind the rules is that when all runtime systems look at the same input or output, they should make an unanimous decision merely based on a predefined distribu-

tion (i.e., the static multi-level block cyclic distribution) without any communication. Note that the following rules of 1, 2-4 and 5-8 are applied to a task's main output, inputs, and minor-outputs, respectively.

1. *Owner.* Each runtime system looks at a newly generated task's main output. If the main output is assigned to a certain host or GPU (e.g., on node$_i$) based upon a static data distribution, only node$_i$'s runtime system will create an owner task instance. An owner task instance stores the complete information of the task (i.e., input, output, the ready status of each input).

2. *Native input.* Each input of a new task will be checked by every runtime system. If the input and the task's main output are assigned to the same host or GPU (e.g., on node$_i$), only node$_i$'s runtime system will create a native-input task instance. The native-input instance also stores a pointer pointing to the task's owner instance.

3. *Intra-node alien input.* Unlike Rule 2, if the input and the task's main output belong to the same node (e.g., on node$_i$) but on different devices, the runtime system on node$_i$ will create an intra-node alien-input task instance. The intra-node alien-input instance also stores a pointer pointing to the task's owner instance.

4. *Inter-node alien input.* Unlike Rule 3, if the input and the task's main output belong to different nodes, and suppose the input belongs to node$_i$, the runtime system on node$_i$ will create an inter-node alien-input task instance. The inter-node alien-input instance stores the location of the task's main output.

5. *Native minor-output.* All runtime systems look at each minor output of a newly generated task. If the minor output and the task's main output belong to the same host or GPU (e.g., on node$_i$), the runtime system on node$_i$ will create a native minor-output task instance. The task's owner instance stores a pointer pointing to the native minor-output instance.

6. *Sink minor-output.* Unlike Rule 5, if the minor output and the main output belong to different devices (regardless of nodes), and suppose the minor output is assigned to node$_j$, the runtime system on node$_j$ will create a sink minor-output task instance. The sink instance is expecting its correponding source to send data to it.

7. *Intra-node source minor-output.* Unlike Rule 5, if the minor output and the main output belong to different devices but on the same node (e.g., node$_i$), the runtime system on node$_i$ will create an intra-node source minor-output task instance. The intra-node source minor-output stores a pointer pointing to its corresponding sink instance on the same node.

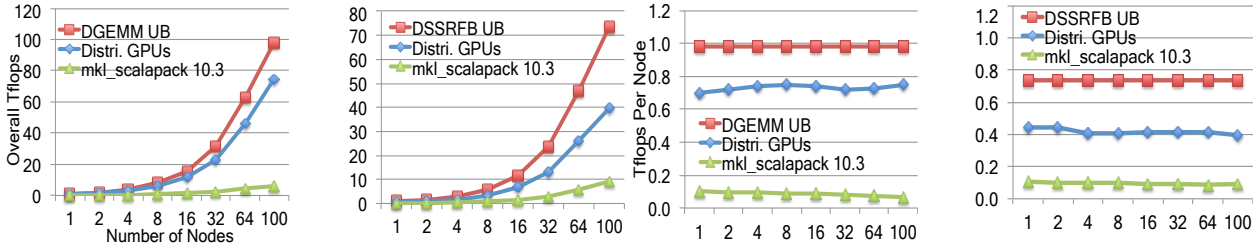
8. *Inter-node source minor-output.* If the minor output and the main output belong to different nodes, and suppose the main output is assigned to node$_i$, the runtime system on node$_i$ will create an inter-node source minor-output task instance. The inter-node source minor-output stores the location of its corresponding sink instance on a remote node.

Since we require the location of an owner task instance be where a computation occurs, our runtime systems is designed to support linking a task's input instances, minor-output instances, and its owner instance together so that the availability of an input triggers the owner. In our runtime system, the linking information is either a direct pointer or the location of the owner task instance. Also by distinguishing *intra-node* from *inter-node*, the runtime system can decide if it needs to copy data to a different device, or send an MPI message to a different node in order to fire a child task.

A distinctive feature of the protocol is that all the runtime systems can follow the same rules to generate tasks and solve data dependencies in an embarrassingly parallel manner without any communication (except for the actual data transfers).

7 Performance Evaluation

We conducted experiments with the Cholesky and QR factorization in double precision on the heterogeneous Keeneland system [23] at the Oak Ridge National Laboratory. The Keeneland system has 120 nodes and is connected by a Qlogic QDR InfiniBand



(a) Cholesky (b) QR (c) Cholesky (d) QR

Figure 6: Weak scalability on distributed GPUs. (a) and (b) show the overall performance, while (c) and (d) show the performance-per-node for the Cholesky and QR factorizations (in double precision), respectively. Every experiment uses all 12 CPU cores and 3 GPUs on each node.

network. Each node on the system runs CentOS 5.5, and has two Intel Xeon 2.8 GHz 6-core processors and three Nvidia Fermi 1.15 GHz M2070 GPUs. The host on each node has 24 GB memories, and each GPU has 6 GB device memories. There is a full PCI-Expresse bandwidth to every GPU on the system. All the nodes have installed CUDA 4.0, Intel Compilers 12.0, Intel MKL 10.3.5, and OpenMPI 1.5.1.

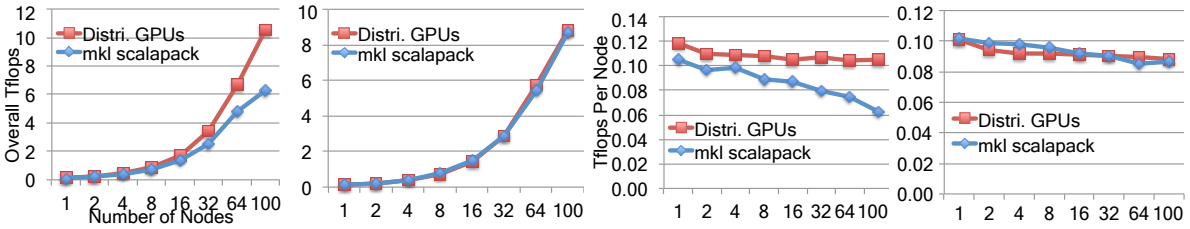
In the following experiments, we perform weak scalability experiments to measure the capability of our programs to solve potentially larger problems if there are more computing resources. While we are focused on clusters with distributed GPUs, our framework is also able to achieve high performance in the other two environments: clusters without GPUs, and shared-system multiGPUs (i.e., a node with both CPUs and GPUs).

7.1 Distributed GPUs

We did experiments on Keeneland using all twelve CPU cores and all three GPUs on each node. Figure 6 shows how the performance of our distributed-GPU framework scales as we increase the number of nodes and the matrix size simultaneously. Although there are 120 nodes on Keeneland, its batch scheduler only allows a job to use 110 nodes in maximum. Considering one or two nodes are unavailable sometimes, we increase the number of nodes from one to 100. As the number of nodes is increased by k, we increase the matrix size by $\sqrt{k}$. The single-node experiment takes an input matrix of size 34,560.

In our experiments, we measure the total number of TeraFlops to solve the Cholesky factorization and the QR factorization, as shown in Figure 6 (a) and (b), respectively. To show the possible maximum performance (i.e., upper bound) of our programs, we also depict the curves of *DGEMM* and *DSSRFB* that are the dominant computational kernels of Cholesky factorization and QR factorization, respectively. We calculate an upper bound by the following formula: `kernel UB` = serial_cpu_kernel_perf $\times$ #cores + gpu_kernel_perf $\times$ #gpus. To show the benefits of using GPUs, we also present the performance of the Intel MKL 10.3.5 ScaLAPACK library which uses CPUs only. In (a), the overall performance of our distributed-GPU Cholesky factorization reaches 75 TFlops on 100 nodes, while MKL ScaLAPACK reaches 6.3 TFlops. In (b), the overall performance of our distributed-GPU QR factorization reaches 40 TFlops on 100 nodes, while MKL ScaLAPACK reaches 9.2 TFlops.

Figure 6 (c) and (d) show another view (i.e., Performance Per Node) for the same experiments as displayed in (a) and (b). That is, TFlops-Per-Node = $\frac{Overall\ TFlops}{Number Nodes}$ on a given number of nodes. Ideally, the performance-per-node is a constance number in a weak scalability experiment. From (c), we can see that our distributed-GPU Cholesky factorization does not lose any performance from one node to 100 nodes. In (d), our distributed-GPU QR factorization scales well again from four nodes to 100 nodes. The performance-per-node on four nodes drops from 0.44 TFlops to 0.41 TFlops because the four-node experiment uses a 2×2 process grid and has a larger communication overhead than a process grid with $P_r = 1$ (Appendix A.2 analyzes the related communication cost).



(a) Cholesky (b) QR (c) Cholesky (d) QR

Figure 7: Weak scalability on clusters with CPUs only. Every experiment uses 12 CPU cores on each node.

7.2 Clusters without GPUs

We did another set of experiments to test whether our framework is capable of delievering high performance when the system is a conventional cluster without GPUs. We only use the 12 CPU cores on each node to do the experiments, and compare our Cholesky and QR factorization implementation with the Intel MKL 10.3.5 ScaLAPACK library.

We performed weak scalability experiments again, where the input size increases by $\sqrt{2}$ whenever we double the number of nodes. In Fig. 7 (a), the overall performance of the ScaLAPACK Cholesky factorization is slower than our Cholesky factorization by 43% on 100 nodes. In (b), our QR factorization program and the ScaLAPACK QR factorization have comparable overall performance. Figure 7 (c) and (d) show the performance per node. In (c), our CPU-only Cholesky factorization scales well from 2 to 100 nodes. Its curve has a dip from one to two nodes since the runtime system on each node uses a dedicated core to do MPI communication (i.e., $\frac{1}{12}$ less computing power) if there are more than one node. Similar to Cholesky factorization, in (d), our QR factorization scales well from 4 to 100 nodes. Because of its good scalability, our QR program eventually outperforms the Intel MKL ScaLAPACK QR factorization by 5% when the number of nodes is greater than 32. Note that we use 11 out of 12 cores on each node to do the real computation, while ScaLAPACK uses all 12 cores, however we are still 5% faster.

7.3 Shared-System MultiGPUs

To evaluate our framework on a shared-system with multicore CPUs and multiple GPUs, we compare our Cholesky factorization to StarPU 0.9.1 [6] on a single node of the Keeneland system.

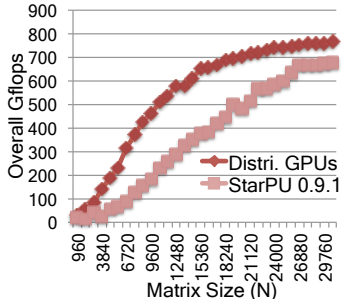


Figure 8: Cholesky factorization (in double precision) on a shared-system with 12 cores and 3 GPUs.

StarPU uses a dynamic scheduling runtime system to assign tasks to CPUs and GPUs to keep load balancing and reduce data transfers. The StarPU implementation of Cholesky factorization uses the same computational kernels as ours, which calls subroutines from the Intel MKL 10.3.5, CUBLAS 4.0, and MAGMA 1.0 libraries. With the help from one of the StarPU developers, we ported the StarPU Cholesky factorization to Keeneland and also tuned its performance thoroughly. Porting the StarPU QR factorization to Nvidia Fermi GPUs is not successful so far due to numerical errors in the result.

Figure 8 shows the overall performance of our framework and StarPU 0.9.1 to solve double-precision Cholesky factorizations. All the StarPU experiments use 9 CPU cores and 3 GPUs to do the real computation, and use the remaining three cores to manage the three GPUs. By contrast, our implementation uses 8 CPU cores and 3 GPUs to do the real computation since we also use an additional core to do CUDA communications. The performance data shows that our framework can rise to high performance more quickly than the StarPU program. When the input size is not very large, our framework is faster than StarPU (i.e., 250% faster when $N \leq 7680$,

and 100% faster when N $\leq$ 12480). When the input size is sufficiently large (i.e., N $\geq$ 26,880), StarPU starts to be close to our framework.

8 Related Work

There are a number of runtime systems developed to support multiple GPU devices on a shared-memory system. StarPU develops a dynamic scheduling runtime system to execute a sequential code on the host CPUs and GPUs in parallel [6], and has been applied to the Cholesky, QR, and LU factorizations [3, 2, 1]. SuperMatrix is another runtime system that supports shared-memory systems with multiple GPUs [19]. It uses several software-cache schemes to maintain the coherence between the host RAM and the GPU memories. While SuperMatrix requires that GPUs take most of the computations, our framework utilizes all CPU cores and all GPUs on both shared-memory and distributed-memory systems.

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 [7]. A programmer is responsible for specifying which piece of code should be executed on a GPU. Its runtime then executes the annotated code in parallel on the host and GPUs. It is also possible to use the hybrid MPI/SMPSs approach to support clusters with multicore CPUs [17].

There is research work that supports scientific computations on distributed GPUs. Factica uses CUDA to accelerate the LINPACK Benchmark [13] on heterogeneous clusters by modifying the original source code slightly. The revised code intercepts every DTRSM or DGEMM call, and splits it into two calls to execute on both CPUs and GPUs, respectively. Those calls to CPUs relies on setting `OMP_NUM_THREADS` to utilizes all CPU cores on the host. Differently, our distributed GPU framework allows every CPU core to compute tasks independently. On the other hand, we use one MPI process per node, instead of one MPI process per GPU.

Fogue et al. ported the PLAPACK library to GPU-accelerated clusters [14]. They require that CPUs compute the diagonal block factorizations while GPUs compute all the remaining operations. They also store all data in GPU memories to reduce communication. In our method, we distribute a matrix across the host and GPUs, and can utilize all CPU cores and all GPUs. Note that it is possible that the computational power of a host may be greater than that of a GPU such that the host needs to compute most of the work.

Many researchers have studied the static data distribution strategies on 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 [11]. Robert 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 [16]. Our work targets clusters of nodes that consist of multicore CPUs and multiple GPUs, and uses a novel static multi-level block cyclic strategy.

9 Conclusion and Future Work

As the trend of adding more GPUs to each node to deliver high performance continues, it is important to start to design new parallel software on the heterogeneous architectures. We present a new framework to solve dense linear algebra problems on large-scale GPU-based clusters. To attain high performance, we focus our design on minimizing communication, maximizing the degree of task parallelism, accommodating processor heterogeneity, hiding communication, and keeping load balance. The framework essentially consists of a static multi-level data distribution method, a class of heterogeneous tile algorithms, and a distributed scheduling runtime system.

Our experiments with the Cholesky and QR factorizations on the heterogeneous Keeneland system demonstrate good scalability in various environments: clusters with or without GPUs, and shared-systems with multi-GPUs. Our future work along this line is to apply the approach to two-sided factorizations and sparse matrices. Another future work is to add NUMA support to our runtime system to improve performance on each node that has hundreds or even thousands of CPU cores as well as a great number of NUMA memory nodes.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. LU factorization for accelerator-based systems. ICL Technical Report ICL-UT-10-05, Innovative Computing Laboratory, University of Tennessee, 2010.

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *IPDPS 2011*, Alaska, USA, 2011.

[3] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, USA, 2010.

[4] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Julie Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. PLASMA Users' Guide. Technical report, ICL, UTK, 2011.

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

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper., Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[7] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique 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] Grey Ballard, James Demmel, Olga Holtz, and Oded 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] Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves 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] Pierre Boulet, Jack Dongarra, Yves Robert, and Frdric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25(5):547 – 568, 1999.

[12] James W. Demmel, Laura Grigori, Mark Frederick Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. LAPACK Working Note 204, UTK, August 2008.

[13] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:803–820, 2003.

[14] Manuel Fogu, Francisco D. Igual, Enrique S. Quintana-ort, and Robert Van De Geijn. Retargeting PLAPACK to clusters with hardware accelerators. FLAME Working Note 42, 2010.

[15] 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.

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

[17] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

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

[19] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert 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, New York, NY, USA, 2009. ACM.

[20] Sushant Sharma, Chung-Hsing Hsu, and Wu chun Feng. Making a case for a Green500 list. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)/ Workshop on High Performance - Power Aware Computing*, 2006.

[21] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Efficient support for matrix computations on heterogeneous multi-core and multi-GPU architectures. LAPACK Working Note 250, UTK, June 2011.

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

[23] J.S. 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.

```
wait4send = wait4recv = 0;
while(!is_done || wait4send) {
  if(!is_done && !wait4recv) {
    call MPI_Irecv(recv_buf, MPI_ANY_SOURCE, &recv_req);
    wait4recv = 1;
  }
  if(!wait4send) {
    msg = get_msg(host's out_mbox);
    call MPI_Isend(msg->data, msg->dst_pid, &send_req);
    wait4send = 1;
  }
  if(wait4send) {
    call MPI_Test(&send_req);
    if(success) wait4send = 0;
  }
  if(wait4recv) {
    call MPI_Test(&recv_req);
    if(success) {
      store recv_buf to the host's local matrix;
      wait4recv = 0;
    }
  }
}
```

Figure 9: Pseudocode of the MPI communication thread in our distributed runtime system.

A Appendix

A.1 Tile Size Auto-Tuning

We adjust the ratio of the small tiles on the host to the large tiles on GPUs, to keep load balancing between CPUs and GPUs within each node. The load balancing between nodes is attained automatically by the 2-D block cyclic distribution method.

Given a tile of size $B \times B$, we partition it into two parts. Suppose $B_h = b(s-1)$ and $B_g = B - B_h$. The left partition of size $B \times B_h$ is allocated to the host, and the right partition of size $B \times B_g$ is allocated to a GPU. We let $T(m \times n)$ denote the number of floating operations to compute a matrix of size $m \times n$. f_{core} and f_{gpu} denote the speed (i.e., flop/s) on a CPU core or GPU, respectively.

In a block cyclic data distribution (1-D or 2-D), a number of G tiles are allocated to a number of G GPUs such that each GPU has one tile and it takes $T(B \times B_g)/f_{gpu}$ time to compute. Differently, the CPU cores on the host receive G small partitions (each is of size $B \times B_h$) corresponding to the G tiles, and it takes $G \times T(B \times B_h)/(f_{core} \times NumCores)$ time to compute.

To achieve load balancing, we determine B_h by the following formula:

$$\frac{T(B \times B_g)}{f_{gpu}} = \frac{G \times T(B \times B_h)}{f_{core} \times NumCores}$$

$$\Rightarrow \frac{T(B \times B_h)}{T(B \times B_g)} = \frac{f_{core} \times NumCores}{f_{gpu} \times G}$$

$$\Rightarrow B_h = B \times \frac{f_{core} \times NumCores}{f_{core} \times NumCores + f_{gpu} \times G}$$

In practice, f_{core} or f_{gpu} is the maximum performance of the dominant computational kernel in an algorithm. In

addition, we fine-tune the value of B_h automatically. We start from the estimated size B_h and search for an optimal B_h^* near B_h. We wrote a script to execute a matrix factorization with an input of size $N = c_0 \cdot B \cdot G$, where we set $c_0 = 3$ to reduce the tuning time. The script adjusts the value of B_h to search for the minimum difference between the CPU and the GPU computation time. Note that B_h is dependent only on the number of CPU cores and the number of GPUs, assuming the system and the computational kernels are not changed.

The large tile size B is critical for the GPU performance. To determine the best B, we search for the minimal matrix size that provides the best performance for the dominant GPU kernel in an algorithm (e.g., `GEMM` for Cholesky factorization). Our search ranges from 256 to 2048, and is performed only once for every new computational library and every new GPU architecture.

A.2 Communication Cost Analysis

We count the number of messages and the number of words communicated by a process that has the most communication among all processes. Assume there are P processes (one process per node), and the broadcast between processes is implemented by a tree topology. We use P_r and P_c to denote a $P_r \times P_c$ process grid, where $P = P_r \cdot P_c$. In the multi-level block cyclic data distribution, we use n, B, s to denote the matrix size, the top-level tile size, and the number of partitions of each top-level tile, respectively.

A.2.1 Distributed Heterogeneous Tile Cholesky Factorization

For each iteration, we first broadcast the diagonal block down the panel (i.e. $\log P_r$ messages), next each process that owns data on the panel broadcasts to $P_c + P_r$ processes that are waiting for the panel data (i.e. $\frac{\#rows}{BP_r}\log(P_c+P_r)$ messages, where $\#rows$=$n - \lfloor \frac{j}{s} \rfloor B$ at the j-th iteration). The number of messages is expressed as follows:

$$\begin{aligned}
\text{msg}_{chol} &= \sum_{j=0}^{\frac{n}{B}s-1} logP_r + \frac{n - \lfloor \frac{j}{s} \rfloor B}{BP_r}\log(P_c+P_r) \\
&= \frac{ns}{B}\log P_r + \frac{n^2 s}{2B^2 P_r}\log(P_c+P_r) \\
&= \frac{ns}{2B}\log P + \frac{n^2 s}{4B^2\sqrt{P}}\log P + \frac{n^2 s}{2B^2\sqrt{P}}
\end{aligned}$$

And the number of words is:

$$\text{word}_{chol} = \frac{nB}{4}\log P + \frac{n^2}{4\sqrt{P}}\log P + \frac{n^2}{2\sqrt{P}}$$

A.2.2 Distributed Heterogeneous Tile QR Factorization

In the tile QR factorization, we can stack up v adjacent tiles to form a virtual tile, which is always allocated to the same host or GPU. At the j-th iteration, each process has $\frac{n-\lfloor \frac{j}{s} \rfloor B}{(vB)P_r} \times \frac{n-\lfloor \frac{j}{s} \rfloor B}{BP_c}$ virtual tiles of size $(vB) \times B$. Since one $B \times B$ tile out of every virtual tile will be sent down to its below process as a message (there is no message if P_r=1), and every tile on the panel will be broadcast right to P_c processes, the number of messages is expressed as follows:

$$\begin{aligned}
\text{msg}_{qr} &= \sum_{j=0}^{\frac{n}{B}s-1} \frac{n - \lfloor \frac{j}{s} \rfloor B}{vBP_r} \cdot \frac{n - \lfloor \frac{j}{s} \rfloor B}{BP_c} + \frac{n - \lfloor \frac{j}{s} \rfloor B}{BP_r}\log P_c \\
&= \sum_{j=0}^{\frac{n}{B}s-1} \frac{(n - \lfloor \frac{j}{s} \rfloor B)^2}{vB^2P} + (n - \lfloor \frac{j}{s} \rfloor B)\frac{\log P_c}{BP_r} \\
&= \frac{1}{vB^2P}\frac{n^3 s}{3B} + \frac{n^2 s}{2B^2 P_r}\log P_c \\
&= \frac{n^3 s}{3vB^3P} + \frac{n^2 s}{4B^2\sqrt{P}}\log P
\end{aligned}$$

And the number of words is:

$$\begin{aligned}
\text{word}_{qr} &= \frac{n^3}{3vBP} + \frac{n^2}{4\sqrt{P}}\log P \\
&= (\frac{n}{3vB\sqrt{P}\log P} + \frac{1}{4})\frac{n^2}{\sqrt{P}}\log P
\end{aligned}$$

If we set the virtual tile size v as n/B/P_r, $(vB\sqrt{P})$ is equal to n. Therefore, $\text{msg}_{qr} = \frac{n^2 s}{3B^2\sqrt{P}} + \frac{n^2 s}{4B^2\sqrt{P}}\log P$, and $\text{word}_{qr} = (\frac{1}{3\log P} + \frac{1}{4})\frac{n^2}{\sqrt{P}}\log P$.

A.2.3 Comparison with ScaLAPACK

Table 1 compares the heterogeneous tile algorithms with the communication lower bounds (LB), and the ScaLAPACK subroutines regarding the number of words (i.e., communication volume) and the number of messages. We can see that the heterogeneous tile Cholesky factorization has attained the communication volume lower bound to within a logarithmic factor. The communication volume of the heterogeneous tile QR factorization is greater than the communication volume lower bound by a factor of $(\frac{n}{3vB\sqrt{P}} + \frac{1}{4}\log P)$. Note that we can increase the value of v to minimize QR's communication volume to reach the lower bound to within a factor of $(\frac{1}{3} + \frac{1}{4}\log P)$.

Table 1: Communication cost of the heterogeneous tile algorithms. We assume square matrices, and $P_r = P_c = \sqrt{P}$.

	#words	#messages
Cholesky LB [[8]]	$\Omega(\frac{n^2}{\sqrt{P}})$	$\Omega(\sqrt{P})$
PDPOTRF [[8]]	$(\frac{nb}{4} + \frac{n^2}{\sqrt{P}}) \log P$	$\frac{3n}{2b} \log P$
Hetero. Cholesky	$(\frac{1}{4} \log P + \frac{1}{2}) \frac{n^2}{\sqrt{P}}$	$\frac{n^2 s}{4B^2\sqrt{P}}(\log P + 2)$
QR LB [[12]]	$\Omega(\frac{n^2}{\sqrt{P}})$	$\Omega(\sqrt{P})$
PDGEQRF [[12]]	$(\frac{3}{4}\frac{n^2}{\sqrt{P}} + \frac{3}{4}nb) \log P$	$(\frac{3}{2} + \frac{5}{2b})n \log P$
Hetero. QR	$(\frac{n}{3vB\sqrt{P}\log P} + \frac{1}{4}) \frac{n^2}{\sqrt{P}} \log P$	$\frac{n^3 s}{3vB^3 P} + \frac{n^2 s}{4B^2\sqrt{P}} \log P$