Accelerating LINPACK with MPI-OpenCL on Clusters of Multi-GPU Nodes

Gangwon Jo, *Student Member, IEEE*, Jeongho Nah, Jun Lee, Jungwon Kim, *Member, IEEE*, and Jaejin Lee, *Member, IEEE*

Abstract—OpenCL is an open standard to write parallel applications for heterogeneous computing systems. Since its usage is restricted to a single operating system instance, programmers need to use a mix of OpenCL and MPI to program a heterogeneous cluster. In this paper, we introduce an MPI-OpenCL implementation of the LINPACK benchmark for a cluster with multi-GPU nodes. The LINPACK benchmark is one of the most widely used benchmark applications for evaluating high performance computing systems. Our implementation is based on High Performance LINPACK (HPL) and uses the blocked LU decomposition algorithm. We address that optimizations aimed at reducing the overhead of CPUs are necessary to overcome the performance gap between the CPUs and the multiple GPUs. Our LINPACK implementation achieves 93.69 Tflops (46 percent of the theoretical peak) on the target cluster with 49 nodes, each node containing two eight-core CPUs and four GPUs.

Index Terms—Cluster, GPU, heterogeneous computing, high performance LINPACK, OpenCL

1 Introduction

Accelerators, such as GPUs, Intel Xeon Phi coprocessors, Cell BE processors, and FPGAs, have become popular to achieve both high performance and power efficiency. Heterogeneous clusters using such accelerators are regarded as an efficient way to build high performance computing systems. A node in such heterogeneous clusters was typically built with multicore CPUs and a single accelerator (e.g., a GPU). Recently, more and more cluster systems have started to have multiple accelerators per node to deal with large size problems [1], [2], [3]. Multiple accelerators per node may enlarge the benefits of a heterogeneous system, especially for massively data-parallel applications. To take advantage of multiple accelerators, however, the applications should be optimized to maximize utilization of them.

Open Computing Language (OpenCL) [4] is an open standard to write parallel applications for heterogeneous computing systems. If an application is written in OpenCL, it can run on any processor or any mix of processors that supports OpenCL. Many industry-leading hardware vendors such as AMD [5], IBM [6], Intel [7], NVIDIA [8], and Samsung [9] provide OpenCL frameworks for a variety of processors. Although OpenCL serves as a programming model for a single heterogeneous system (i.e., a single OS instance), application developers for heterogeneous clusters may use a mix of programming models, such as MPI-OpenCL. Since both MPI and OpenCL are portable, applications written in a mix of MPI and OpenCL can be executed on any heterogeneous cluster that supports MPI and OpenCL, without any modification.

The LINPACK benchmark is one of the most widely used benchmark applications for evaluating high performance computing systems. It measures the performance of a computing system for solving dense linear algebra problems that are fundamental components of many scientific applications. Moreover, it is usually used as a yardstick of the performance of supercomputers because the TOP500 supercomputer list [10] ranks supercomputers by their performance on the LINPACK benchmark. The LINPACK benchmark requires $\frac{2}{3}n^3 + O(n^2)$ double-precision floating-point operations to solve a system of linear equations of order n. Reducing the operation count (e.g., using the Strassen algorithm for matrix multiplication) is not allowed. Under this constraint, any optimizations can be applied to the algorithm in order to achieve the best performance for the target system.

Since most of the floating-point operations in the LINPACK benchmark are from matrix multiplications that are relatively easy to parallelize, heterogeneous clusters can greatly improve the performance of the LINPACK benchmark by performing the matrix operations with accelerators. A further improvement is possible by overlapping computation performed by the accelerators with the computation/communication performed by CPUs. This helps to hide small but complicated computations that are not suitable for accelerators. As the number of GPUs per node increases, however, the performance gap between the CPUs and the GPUs widens. The workload of the CPUs also increases to provide enough computations to keep the multiple GPUs to them busy. This makes the CPUs a performance bottleneck.

- *G. Jo, J. Nah, J. Lee, and J. Lee are with the Center for Manycore Programming, Department of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea. E-mail: {gangwon, jeongho, jun}@aces.snu.ac.kr, jlee@cse.snu.ac.kr.*
- *J. Kim was with the Center for Manycore Programming, Department of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea. He is currently with Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA. E-mail: jungwon@aces.snu.ac.kr.*

Manuscript received 14 Sept. 2013; revised 5 Mar. 2014; accepted 3 Apr. 2014. Date of publication 29 June 2014; date of current version 5 June 2015. Recommended for acceptance by S. Ranka. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2014.2321742

In this paper, we introduce an efficient implementation of the LINPACK benchmark for heterogeneous CPU/GPU clusters that have multiple GPUs per node. Our implementation is based on High Performance LINPACK (HPL) [11]. We use a mix of MPI and OpenCL as a programming model for heterogeneous clusters. We focus on optimization techniques aimed at reducing the workload of CPUs and overlapping the computation/communication of the CPUs with the OpenCL kernel execution on the multiple GPUs. The proposed optimization techniques can be adopted not only for the LINPACK benchmark but also for various scientific applications.

We evaluate the performance of our implementation with a 49-node heterogeneous cluster. This target cluster is a part of the 'Chundoong' supercomputer that contains 56 compute nodes. Chundoong was ranked 277th in the TOP500 [10] list and 32nd in the Green500 [21] list of November 2012. Our result shows that MPI-OpenCL can achieve comparable performance than other programming models, and thus it is promising for heterogeneous clusters.

The contributions of the paper are as follows:

- We introduce an efficient implementation of the LINPACK benchmark for clusters of multi-GPU nodes. We present an efficient multi-GPU algorithm that combines DTRSM and DGEMM routines in the Basic Linear Algebra Subprograms (BLAS) library, and dynamically balances workloads between multiple GPUs. We also present optimization techniques that reduce the overhead of CPUs to eliminate the bottleneck. Optimizations for the CPU side is necessary for applications running on clusters with multiple accelerators per node, to overcome the performance gap between the CPUs and the multiple accelerators.
- We implement the LINPACK benchmark and its optimization techniques using a mix of MPI and OpenCL. OpenCL is used to handle multiple GPUs in each node. MPI is used for communication between nodes. To the best of our knowledge, this is the first work to implement the LINPACK benchmark for heterogeneous clusters using only portable programming models. Prior studies use vendor-specific programming models such as CUDA [12] and AMD CAL [13] for GPUs.
- We show that using more GPUs per node improves power efficiency of the LINPACK benchmark significantly, even if the performance of multiple GPUs is not fully exploited because of the bottleneck in the CPU side. Power consumption is one of the most important design constraints for today's high-performance computing systems. Our result shows that heterogeneous clusters of multi-accelerator nodes are beneficial in terms of the power efficiency, for the LINPACK benchmark and other scientific applications.

The remainder of this paper is organized as follows. Section 2 briefly describes the HPL algorithm. Section 3 explains the OpenCL implementations of the DTRSM and DGEMM routines. Section 4 describes our baseline implementation and optimization techniques for reducing the execution time of the CPU side. Section 5 discusses and analyzes the evaluation result of our LINPACK implementation. Section 6 briefly describes the related work. Finally, Section 7 concludes the paper.

2 HPL

Our work is based on HPL [11] that is a well-known implementation of the LINPACK benchmark for distributed-memory clusters. This section briefly describes the algorithm used in HPL and characterizes its workload.

HPL obtains the solution of a linear system of order n, $Ax = b$ using the LU decomposition method. Most of its execution time is spent on the LU decomposition stage that requires $O(n^3)$ double-precision floating-point operations. After the LU decomposition, obtaing the solution requires only $O(n^2)$ double-precision floating-point operations [14].

HPL uses MPI for communication between nodes, and the BLAS library for matrix and vector operations. HPL performs most of its floating-point operations for DTRSM and DGEMM routines in BLAS.

2.1 HPL Algorithm

HPL uses the right-looking blocked LU decomposition algorithm [15]. It adopts the block-cyclic data distribution scheme [16]. The coefficient matrix is logically divided into $n_b \times n_b$ blocks, as shown in Fig. 1a. Nodes are arranged as a two-dimensional $P \times Q$ grid. The blocks of the coefficient matrix are cyclically dealt onto the grid. It means that the block (i, j) in the coefficient matrix is allocated to the node $(i \bmod P, j \bmod Q)$. Fig. 1b illustrates how the 8×8 blocks of the coefficient matrix are distributed across the nodes in a 2×3 grid. Note that blocks in the same row (or column) are handled by the nodes in the same row (or column).

The blocked LU decomposition algorithm consists of n/n_b iterations. In the first iteration, the coefficient matrix is divided into three submatrices L^1, U^1, and A^1. In the $(i+1)$th iteration, a submatrix A^i in the preceding iteration is divided into L^{i+1}, U^{i+1}, and A^{i+1}. For example, Fig. 2a shows L^1, U^1, and A^1, and Fig. 2b shows L^3, U^3, and A^3. The ith iteration consists of the following four steps:

- *Factorization.* The LU factorization is performed on L^i. One of the three recursive variants of the LU decomposition algorithm (Crout, left-looking, and right-looking) can be used. P nodes containing the blocks in L^i cooperate for the LU factorization. These 'factorizing' nodes communicate with each other to choose pivot rows and swap them with some other rows in L^i. Indices of the pivot rows are recorded in array D^i_{PIV}. In addition, they invoke BLAS routines many times to perform small-sized matrix or vector operations.

  Let L^i_1 be the lower triangular matrix of the topmost block of L^i and L^i_2 be the rest of the blocks in L^i. As a result of the factorization phase, all factorizing nodes have the same copies of D^i_{PIV} and L^i_1. Also, they have their own blocks of L^i_2.
- *Broadcast.* Each factorizing nodes in a row broadcast the copies of D^i_{PIV}, L^i_1, and its blocks in L^i_2 to other nodes in the same row.

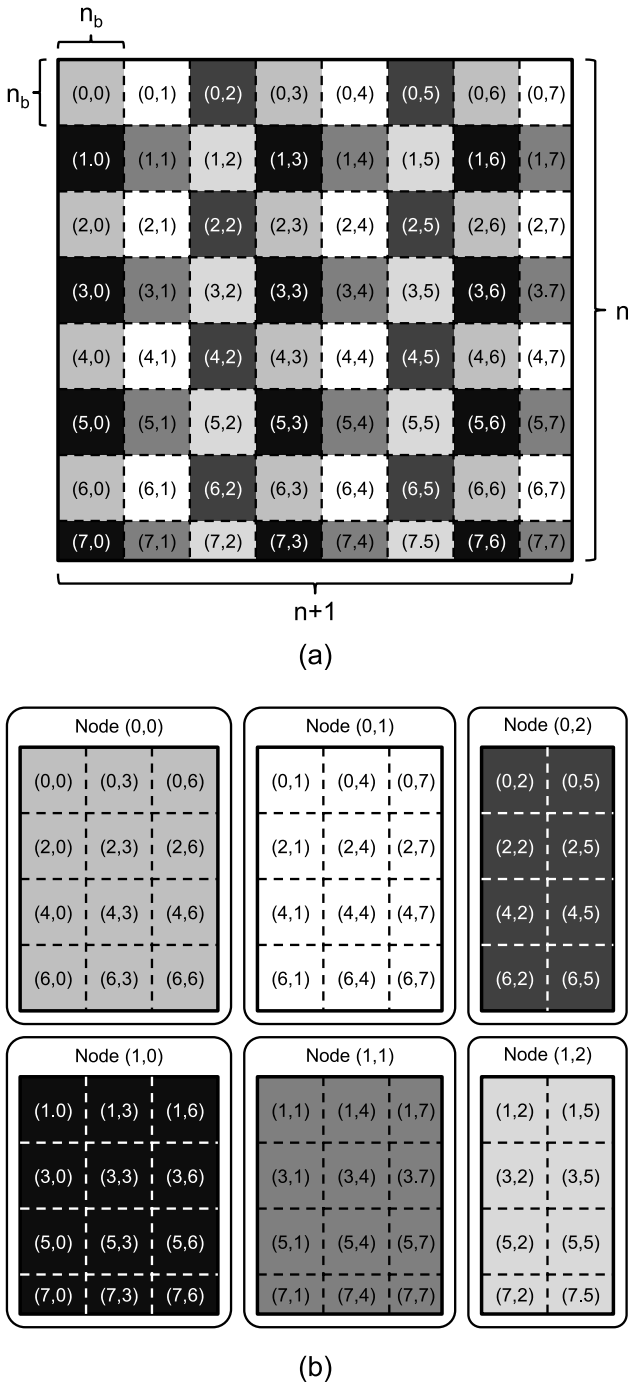

Fig. 1. The coefficient matrix is divided into blocks and those blocks are distributed across the nodes in a 2×3 grid.

- *Swap*. Rows in U^i and A^i are swapped using D^i_{PIV}. Swapping occurs between the nodes in the same column. After swapping, the node that has the blocks of U^i in a column broadcasts the blocks to other nodes in the same column. Note that HPL provides two swapping algorithms, *binary-exchange* and *long*, which perform swapping and broadcasting at the same time. Users may choose one of them.
- *Update*. Each node independently updates its own blocks of U^i and A^i without any communication. The matrix U^i is updated with X by solving the matrix equation $L^i_1 X = U^i$ using the DTRSM routine in the BLAS library. Then, the matrix A^i is updated with the result of $A^i - L^i_2 U^i$ using the DGEMM routine.

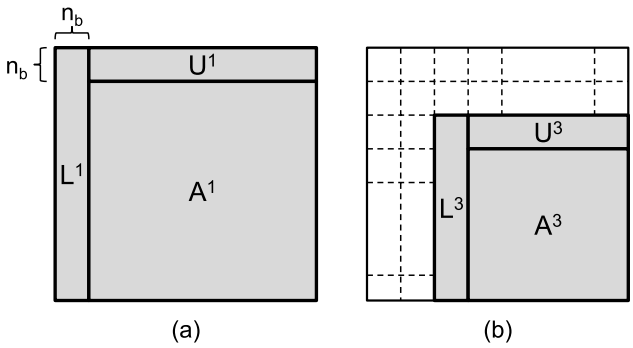

Fig. 2. The submatrices L^1, U^1, A^1, L^3, U^3, and A^3 for the blocking LU decomposition.

The LU decomposition process performs approximately $n_b^2 \cdot m$ double-precision floating-point operations in the factorization phase of the ith iteration, where m is the height of submatrix A^i (i.e., $n - n_b \cdot i$). In the update phase of the ith iteration, DTRSM and DGEMM routines perform $n_b^2 \cdot (m+1)$ and $m \cdot (m+1) \cdot (2n_b + 2)$ operations, respectively. Note that only the factorization and update phases perform double-precision floating-point operations.

We used HPL as a baseline of our implementation to easily evaluate the performance. However, the blocked LU decomposition algorithm and the block-cyclic data distribution scheme are also used in ScaLAPACK [16]. It is a well-known linear algebra library for distributed-memory systems and originated from LAPACK [17]. This implies that our work can also be applied to a wide range of scientific applications.

2.2 Workload Characterization

For simplicity of discussion, assume that the double-precision floating-point operations required in the ith iteration of the blocked LU decomposition are uniformly distributed on and taken care of by a $P \times P$ grid of nodes. This assumption is reasonable when m (the height of submatrix A^i) is much bigger than n_b. Then, P factorizing nodes take care of $(n_b^2 \cdot m)/P$ operations each in the factorization phase. In addition, each node in the grid takes care of $n_b^2 \cdot (m+1)/P$ operations for DTRSM and $m \cdot (m+1) \cdot (2n_b + 2)/P^2$ operations for DGEMM in the update phase. Although all $P \times P$ nodes execute the DTRSM routine, the operation count of DTRSM is divided by only P (the number of columns). This is because each node in the same column redundantly computes the same section of U^i with the DTRSM routine. In the broadcast and swap phases, each node sends and receives on average $O(m \cdot n_b)$ data.

Fig. 3 shows the breakdown of the number of double-precision floating-point operations taken care of by each factorizing node for different values of m when $n_b = 2{,}000$ and $P = 7$. As the size of A^i decreases, the fraction of the update phase (DTRSM + DGEMM) decreases. Since our LINPACK implementation executes the update phase (both DTRSM and DGEMM) on GPUs, this implies that more iterations in the LU decomposition fully utilize the GPUs when the size of the coefficient matrix is bigger. However, we cannot make the

Height of A^i	Factorization	DTRSM	DGEMM
800,000	0.86%	0.86%	98.28%
600,000	1.14%	1.14%	97.72%
400,000	1.69%	1.69%	96.62%
200,000	3,27%	3.27%	93.46%

Fig. 3. The breakdown of the number of double-precision floating-point operations taken care of by each factorizing node for different sizes of submatrix A^i when $n_b = 2{,}000$ and $P = 7$.

coefficient matrix indefinitely bigger because the size of main memory in each node is limited.

Fig. 4 shows the breakdown for different values of n_b when $m = 600{,}000$ and $P = 7$. As n_b decreases, the fraction of the update phase (DTRSM + DGEMM) increases. This implies that a smaller n_b gives more work to the GPUs and may better utilize them. In addition, since both of the number of operations in the factorization phase and the amount of inter-node communication in other phases are proportional to $m \cdot n_b$, a bigger n_b implies more operations in the factorization phase and more inter-node communication. This may make the CPUs a performance bottleneck because CPUs are in charge of the factorization phase and inter-node communication in our approach. Unlike the size of the coefficient matrix, n_b can be easily made small enough by the user. However, using a smaller n_b degrade the performance of DGEMM, especially when multiple GPUs are used in each node. We will discuss the trade-off between n_b and the performance of DGEMM in Section 5.3.

3 Multi-GPU DTRSM-DGEMM

Most of the double-precision floating-point operations in the blocked LU decomposition are from the update phase. Thus, our LINPACK implementation performs DTRSM and DGEMM of the update phase on multiple GPUs. Some non-OpenCL DGEMM implementations for AMD GPUs are publicly available [13], [18], [19]. We develop an OpenCL routine that is suitable and optimized for the target cluster with multiple GPUs.

In our approach, we do not implement these two routines separately. Instead, we embed the DTRSM kernel in the DGEMM workflow and execute them together. We call the resulting routine as DTRSM-DGEMM. This section describes our DTRSM-DGEMM implementation for multiple GPUs.

3.1 Workflow of DTRSM-DGEMM

In the update phase of the ith iteration, each node has its own blocks of L_2^i, U^i, and A^i, and independently updates them. Hence, just consider the update phase in a single node. Let $[L_2]$, $[U]$, and $[A]$ be the blocks of L_2^i, U^i, and A^i in the node, respectively. The node solves $L_1^i X = [U]$ using DTRSM and updates $[U]$ with X. Then, $[A]$ is updated with $[A] - [L_2][U]$ using DGEMM.

n_b	Factorization	DTRSM	DGEMM
4,000	2.23%	2.23%	95.54%
3,000	1.69%	1.69%	96.62%
2,000	1.14%	1.14%	97.72%
1,000	0.58%	0.58%	98.85%

Fig. 4. The breakdown of the number of double-precision floating-point operations taken care of by each factorizing node for different n_b when the height of $A^i = 600{,}000$ and $P = 7$.

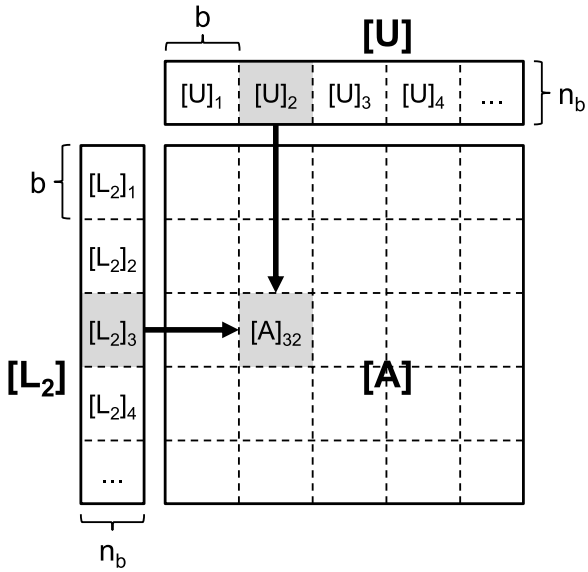

Fig. 5. Blocks of matrices $[L_2]$, $[U]$, and $[A]$ used in DTRSM-DGEMM.

As described in Section 2.2, the coefficient matrix should be as large as possible to achieve better performance. However, due to the memory size limitation of the GPU, GPUs cannot compute the entire $[U]$ or $[A]$ at a time. Instead, each GPU compute a submatrix of $[U]$ or $[A]$ at a time. As shown in Fig. 5, $[A]$ is divided into blocks $[A]_{pq}$ whose size is $b \times b$. $[L_2]$ and $[U]$ are divided into blocks $[L_2]_p$ and $[U]_q$ whose sizes are $b \times n_b$ and $n_b \times b$, respectively. Note that b may be different from n_b. The value of $b = 3{,}968$ for our DTRSM-DGEMM implementation is determined empirically to achieve the best performance of the DGEMM kernel.

In the update phase, L_1^i is transferred to all GPUs first, because the same triangular matrix L_1^i is used for all blocks of $[U]$. Then $[U]_q$ can be computed as follows. First, $[U]_q$ is copied into a single contiguous chunk of main memory by the CPUs (CopyU step). Then the copy is transferred to the memory of the GPU that is in charge of computing $[U]_q$ (WriteU step). The OpenCL kernel for DTRSM compute a new $[U]_q$ on the GPU (DTRSM step), and the result is transferred back to the contiguous chunk of main memory from the GPU (ReadU step). Finally, the result is written back to $[U]_q$ (UpdateU step).

In addition, after the DTRSM step, the new $[U]_q$ is stored in the GPU and each $[A]_{pq}$ can be computed as follows. $[L_2]_p$ is copied into a single contiguous chunk of main memory (CopyL2 step), and the copy is transferred to the memory of the GPU (WriteL2 step). The OpenCL kernel for DGEMM computes $\alpha[L_2]_p[U]_q$ on the GPU (DGEMM step), and the result is transferred back (ReadA step). The result is added to $[A]_{pq}$ in the main memory by the CPUs (Sum step).

The CopyU, CopyL2, UpdateU, and Sum steps are performed on the CPUs. Thus, they can be overlapped with other steps that are performed on GPUs in a pipelined manner. Moreover, kernel execution (DTRSM and DGEMM

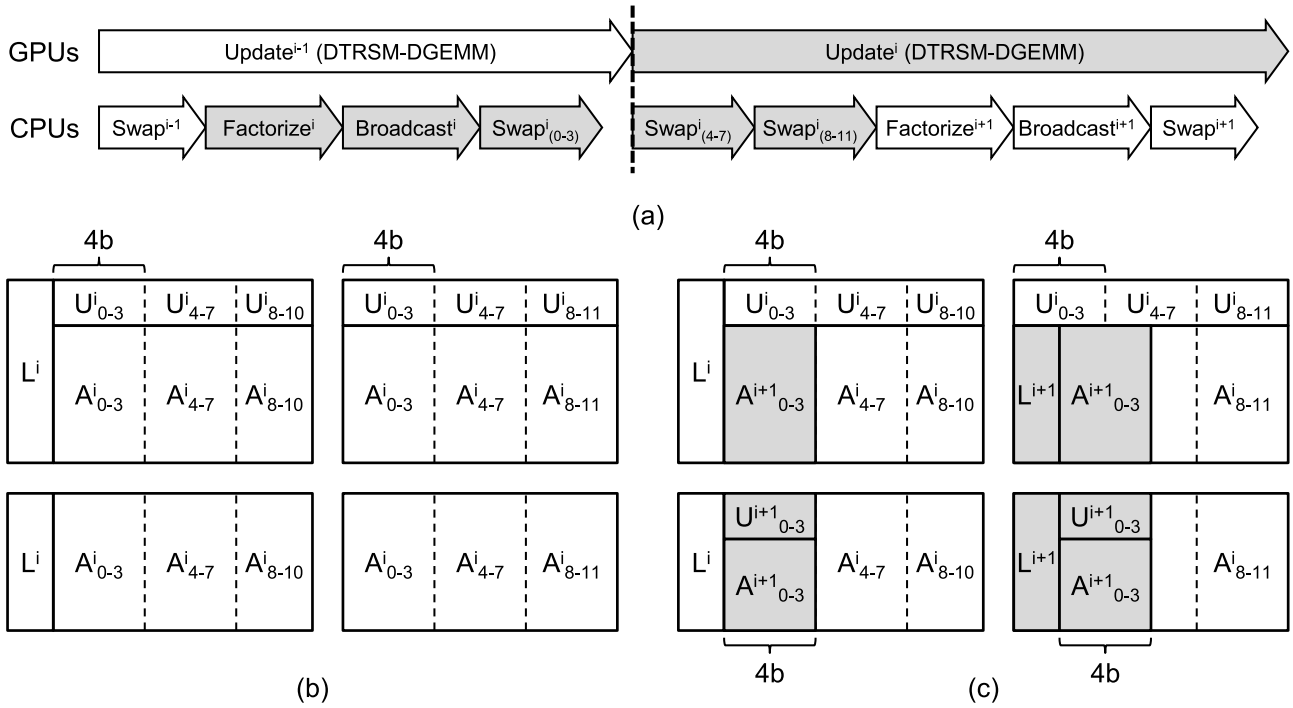

Fig. 6. The key idea of the baseline implementation. The CPUs perform the factorization, broadcast, and swap phases of the $(i+1)$th iteration, and these phases are overlapped with the update phase of the ith iteration performed by the GPUs.

steps) can also be overlapped with data transfer (WriteU, ReadU, WriteL2, and ReadA steps). For example, assume that a GPU performs DGEMM for $[A]_{pq}$, $[A]_{(p+1)q}, \ldots$, and $[A]_{(p+k)q}$ in turn, where $k \geq 4$. While the DGEMM steps of the current block $[A]_{(p+l)q}$ $(2 \leq l \leq k-2)$ are performed, the WriteL2 step of the next block $[A]_{(p+l+1)q}$ and the ReadA step of the previous block $[A]_{(p+l-1)q}$ are also performed in parallel. In addition, the CPU performs the Sum step of the block $[A]_{(p+l-2)q}$ and the CopyL2 step for the block $[A]_{(p+l+2)q}$ in parallel. To achieve this, one CPU thread is assigned to each GPU.

3.2 Optimizing Kernels

To reduce the data access latency by the OpenCL DGEMM kernel, we use the read-only texture memory of GPUs to store the blocks of $[L_2]$ and the updated blocks of $[U]$ [19]. The OpenCL DGEMM kernel assumes that $[U]_q$ is in the transposed form, since this assumption makes the kernel faster when matrices are stored in column-major order. HPL uses column-major order and stores the submatrix $[U]$ in the transposed form by default. Our approach also works when the submatrix $[U]$ is not in the transposed form because the blocks of $[U]$ can be transposed in the CopyU step.

3.3 Dynamic Load Balancing

Our DTRSM-DGEMM dynamically assigns the columns of blocks in $[U]$ and $[A]$ to the GPUs from left to right, one column for a GPU at a time. If a GPU finishes the assigned column and there is no column left to be scheduled, the GPU may steal some blocks that have been assigned to another GPU. Such dynamic scheduling achieves load balancing across GPUs resulting in performance improvement and better GPU utilization.

4 LINPACK Implementation

In this section, we describe our implementation of the LINPACK benchmark for the heterogeneous cluster with multi-GPU nodes. It uses DTRSM-DGEMM described in the previous section. To reduce the overhead of the CPUs, we apply four different optimizations to our baseline implementation.

4.1 Baseline Implementation

The key idea of the baseline implementation is that CPUs perform the factorization, broadcast, and swap phases of the $(i+1)$th iteration in the blocked LU decomposition, and these phases are overlapped with the update phase of the ith iteration performed by the GPUs. Fig. 6a illustrates the key idea. It shows three consecutive iterations, the $(i-1)$th, ith, and $(i+1)$th of the blocked LU decomposition. We distinguish the phases in different iterations with a superscript that denotes the iteration.

We assume that there are four GPUs in each node. Fig. 6b shows an example that the submatrices L^i, U^i, and A^i are distributed across 2×2 nodes. As shown in Fig. 6b, each node logically divides its own blocks of U^i and A^i into $4b$ columns, where b is the size of blocks used in DTRSM-DGEMM. Similarly, Fig. 6c shows the submatrices L^{i+1}, U^{i+1}, and A^{i+1} on a 2×2 grid of nodes.

When the GPUs start $Update^i$, the leftmost $4b$ columns of U^i (U^i_{0-3}) and A^i (A^i_{0-3}) in each node have been swapped by the CPUs already. Each b columns of U^i and A^i are assigned to each GPU at the beginning of $Update^i$. While the GPUs execute DTRSM-DGEMM on U^i_{0-3} and A^i_{0-3}, the next $4b$ columns in each node, U^i_{4-7} and A^i_{4-7}, are swapped by the CPUs. After swapping, those columns are available for

GPUs in the update phase. Swapping $4b$ columns by the CPUs is usually faster than performing DTRSM-DGEMM on $4b$ columns with GPUs.

Three types of synchronization are required between the CPUs and the GPUs. To start executing DTRSM-DGEMM on $U^i_{p-(p+3)}$ and $A^i_{p-(p+3)}$, $Swap^i_{p-(p+3)}$ must be finished. To start $Factorize^{i+1}$, $Update^i$ on matrix L^{i+1} must be finished (Fig. 6c). $Update^i$ on matrix U^{i+1}_{0-3} and A^{i+1}_{0-3} must be finished before $Swap^{i+1}$ swaps them.

To achieve the ordering, our DTRSM-DGEMM maintains a *working window*, $[w_l, w_r)$. The working window is the interval of columns in $[U]$ and $[A]$ that the GPUs may update. The columns at the left side of the w_lth column are already updated by the GPUs. The columns at the right side of the w_rth column are not yet available for the update phase (i.e., the swap phase on those columns are not finished yet). Whenever a GPU completely updates a column of blocks, the CPU thread assigned to the GPU updates w_l. $Factorize^{i+1}$ and $Swap^{i+1}$ can be started after w_l becomes greater than or equal to both of n_b and $4b(+\,n_b)$. Whenever the CPUs finish $Swap^i_{p-(p+3)}$, they set w_r to $(p+4)b$. DTRSM-DGEMM does not assign the column of blocks outside the working window (i.e., at the right side of the w_rth column) to the GPUs. If a GPU finishes the assigned column and there is no column left in the working window, the GPU waits until w_r increases and some columns of blocks become available.

When the total execution time of the factorization, broadcast, and swap phases is smaller than that of the update phase, our baseline implementation may completely hide the execution time of CPUs and fully exploit the capability of multiple GPUs. Note that our overlapping scheme is more aggressive than the original HPL [11] and the previous studies [1], [2], [13]. The *look-ahead* algorithm used in the original HPL overlaps the broadcast phase of the iteration $(i+1)$th iteration and the update phase of the ith iteration. But it does not overlap the factorization and swap phases with the update phase. Rohr et al. [13] overlaps both the factorization and broadcast phases with the update phase. However, the swap phase and the update phase are still performed in order.

The factorization phase requires BLAS routines for small-sized matrix and vector operations. We use Intel Math Kernel Library [20] for multicore CPUs because the factorization phase is performed by CPUs. Since each node in the target cluster has 16 cores, a CPU core is assigned to each GPU to manage DTRSM-DGEMM, 1 core is assigned for MPI communication, and the remaining cores are assigned to the OpenCL host thread and Intel Math Kernel Library.

4.2 Overlapping Factorization and Swap

The factorization phase of the $i+1$th iteration and the swap phase of the ith iteration are performed on different parts of the coefficient matrix. Thus, these two phases can be overlapped on the CPU side. Consider Fig. 6 again. Since $Factorize^{i+1}$ on L^{i+1} has a dependence to $Update^i$ on L^{i+1}, $Factorize^{i+1}$ can start immediately after $Update^i$ on L^{i+1} finishes without regard to the remaining $Swap^i$.

Both the factorization and the swap use MPI functions to communicate with other nodes. However, the amount of communication occurred in the factorization phase is much less than that in the swap phase. In addition, the swap phase requires only a single CPU core for communication. Thus, these two can be overlapped with each other.

This optimization implies that the time taken by $Update^i$ on L^{i+1} determines how fast $Factorize^{i+1}$ can start. Having n_b equal to b (the size of blocks used in DTRSM-DGEMM), we make L^{i+1} to be a column of $b \times b$ blocks and assign all GPUs to the blocks in L^{i+1} in $Update^i$. Note that a GPU is assigned to a single column of blocks in the update phase of the baseline implementation. We apply this optimization adaptively only when the CPUs become a performance bottleneck because it slightly hurts the performance of the update phase by making the data transfers to the GPUs bursty.

4.3 Distributing the Factorization Phase

Assume that our baseline implementation is executed on $P \times Q$ nodes. For each iteration, only P nodes cooperate in the factorization phase and other $P \times (Q-1)$ nodes just wait until the factorization phase is finished by factorizing nodes. Utilizing those idle nodes in the factorization phase improves the performance of the factorization phase.

Fig. 7 illustrates the idea with a 2×3 grid of nodes. First, when all the blocks of L^i have been updated by the GPUs and are ready for the factorization phase, all nodes stop performing the swap phase. Each factorizing node distributes its blocks of L^i to other $Q-1$ nodes in the same row as shown in Fig. 7a. Then, the swap phase resumes. All nodes have some blocks of L^i and can cooperate in the factorization phase. (Fig. 7b). As a result, they have the same copies of D^i_{PIV} and L^i_1, and their own resulting blocks of L^i_2. Thus, D^i_{PIV} and L^i_1 do not need to be broadcasted in the broadcast phase. Instead, each node broadcasts its blocks of L_2 to other nodes in the same row in the broadcast phase (Fig. 7c).

This optimization incurs an extra overhead to stop/resume the swap phase and to distribute blocks of L^i. Thus, if the communication between nodes is relatively slow, it may not be beneficial.

4.4 Optimizing the Swap Phase

The *long* swapping algorithm used in the baseline implementation is originally optimized for large matrices [11]. It consists of five steps. First, nodes containing the blocks in U^i rearrange and copy the rows in U^i to a contiguous buffer using D^i_{PIV} ($\mathsf{Copy}_{\mathsf{swap}}$). Then, these nodes spread their rows to other nodes in the same column ($\mathsf{Spread}_{\mathsf{swap}}$). After that, each node swaps the rows in the received U^i and its own A^i ($\mathsf{Swap}_{\mathsf{swap}}$), and broadcast the swapped rows to other nodes in the same column ($\mathrm{Roll}_{\mathsf{swap}}$). Finally, all nodes rearrange the received rows and complete blocks of U^i ($\mathsf{Permute}_{\mathsf{swap}}$).

Although we apply the optimization technique described in Section 4.2 to the baseline implementation, the swap phase cannot be completely hidden by the factorization phase. For example $Swap^i_{0-3}$ cannot be hidden by $Factorize^{i+1}$ in Fig. 6a. We can further optimize it. We divide the leftmost $4b$ columns of the ith iteration, U^i_{0-3} and A^i_{0-3} in Fig. 6b, into two $2b$ columns, (U^i_{0-1}, A^i_{0-1}) and (U^i_{2-3}, A^i_{2-3}). Then, we swap the rows in these two parts concurrently. The $\mathsf{Copy}_{\mathsf{swap}}$ step can be started immediately after D^i_{PIV} is completed. Thus, it can be overlapped with

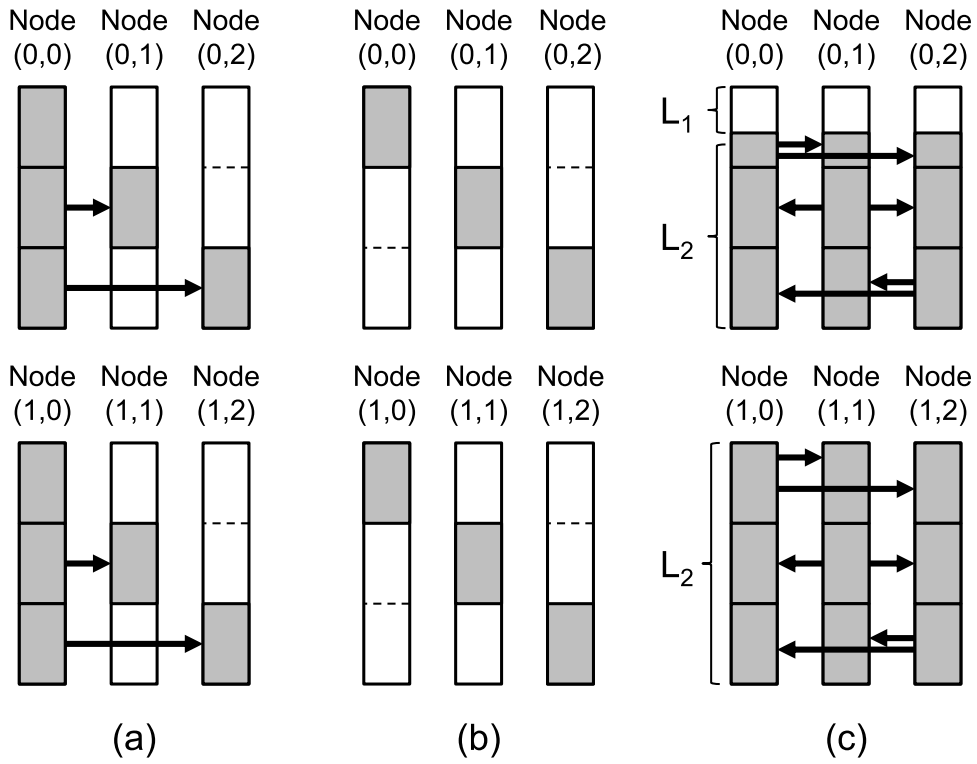

Fig. 7. Distributing the workload in the factorization phase between nodes in a 2×3 grid.

broadcasting L_2^i ($Broadcast^i$). $\text{Copy}_{\text{swap}}$ and $\text{Permute}_{\text{swap}}$ are performed by CPUs and can be overlapped with the communication occurred in $\text{Spread}_{\text{swap}}$ and $\text{Roll}_{\text{swap}}$. Consequently, these four steps are interleaved as shown in Fig. 8.

5 Evaluation

In this section, we evaluate our LINPACK implementation and its optimization techniques.

5.1 The Target Cluster

The target cluster consists of 49 nodes. Each node contains two Intel Xeon E5-2650 CPUs (i.e., 16 CPU cores), four AMD Radeon HD 7970 graphics cards, and 128 GB of main memory. GPUs are overclocked from 925 to 960 MHz. Thus, the theoretical peak performance of the GPU is 983 Gflops for double-precision floating-point operations, while that of two CPUs in a node is 256 Gflops. The theoretical peak performance of the entire cluster is 205.2 Tflops. Each GPU has 3 GB of memory, but only 2 GB is available for OpenCL applications. GPUs are connected to CPU cores by four PCI-E Gen. 3 links with 16 lanes each. The nodes are connected by Mellanox InfiniBand QDR switches.

Each node runs Red Hat Enterprise Linux Server 6.3. AMD Accelerated Parallel Processing (APP) SDK 2.8 together with the ATI Catalyst driver 13.01 is used for OpenCL support. Open MPI 1.6.4 is used as the MPI library. We run our LINPACK implementations on a 7×7 grid of nodes.

The target cluster is a part of the 'Chundoong' supercomputer that has 56 compute nodes. Chundoong was ranked 277th in the TOP500 [10] list and 32nd in the Green500 [21] list of November 2012.

5.2 DTRSM-DGEMM Performance

Fig. 9 shows the performance of our DTRSM-DGEMM implementation described in Section 3 in a single node. We vary the size of the matrix $[A]$ in Fig. 5. We measure the performance of DGEMM (i.e., only $[A]$ is computed) for different number of GPUs, and the performance of DTRSM-DGEMM using 4 GPUs.

DGEMM achieves 656 Gflops (66.7 percent of the theoretical peak) with a single GPU. This low efficiency is

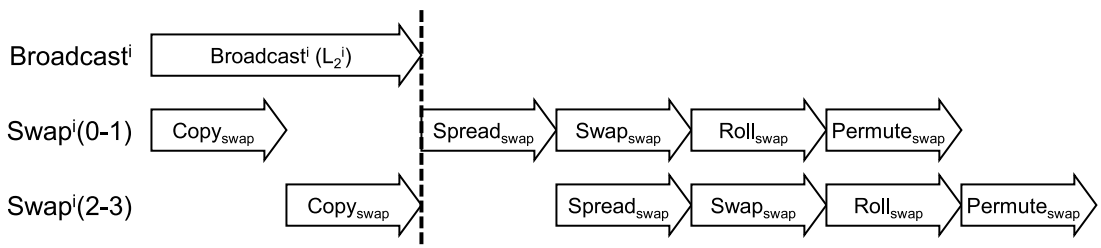

Fig. 8. The idea to optimize the swap phase. The swap phase for the two $3b$ columns are interleaved.

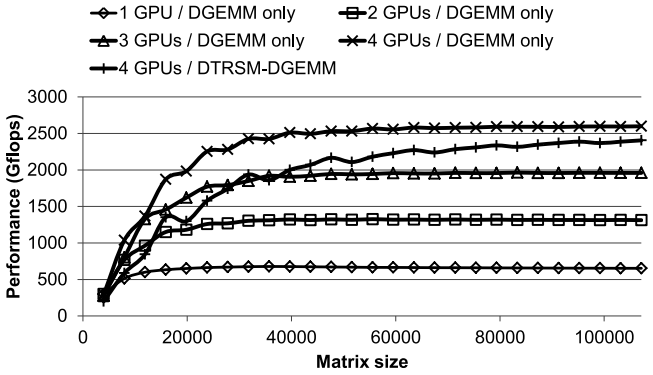

Fig. 9. The performance of DGEMM and DTRSM for different size of $[A]$ in Fig. 5 ($n_b = 3{,}968$).

caused by the low performance of the DGEMM OpenCL kernel. The performance of the kernel is 717 Gflops without counting the data transfer overhead. That is, the computation in the DGEMM OpenCL kernel utilizes only 72.9 percent of the theoretical peak. We cannot increase the kernel-computation-only performance with additional low-level optimization techniques because actual code generation is performed by the AMD OpenCL C compiler, and the low-level SDK is not publicly available for the user.

DGEMM with four GPUs is 3.95 times faster than that with a single GPU. The performance of DTRSM-DGEMM with four GPUs is slightly worse than DGEMM because it performs more computation (i.e., DTRSM) and PCI-E data transfer than the case of DGEMM only.

Fig. 10 shows the sensitivity of our DGEMM to n_b in Fig. 5. To perform DGEMM for $[A]_{pq}$, $b^2 \cdot n_b$ floating point operations, transferring $8 \cdot 2b \cdot n_b$ bytes to a GPU, and transferring $8b^2$ bytes from a GPU are required. As n_b decreases, the amount of computation and the amount of the CPU-to-GPU transfer decrease linearly. However, the amount of the GPU-to-CPU transfer remains constant. This implies that the time taken to perform the CPU-to-GPU transfer and GPU-to-CPU transfer for a single floating point operation increases as n_b decreases. The slow data transfer between CPUs and GPUs typically does not affect performance because the data transfer is completely hidden by the kernel execution. However, if the value of n_b is less than some threshold value, the transfer time becomes large and cannot be hidden. For multiple GPUs, the threshold value is much larger because data transfer becomes slow due to PCI-E bus contention between multiple GPUs. Thus, a large enough value of n_b is required

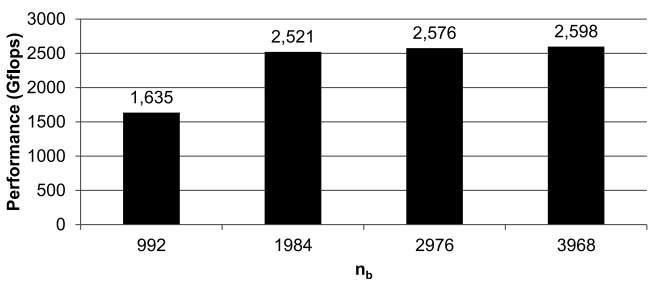

Fig. 10. The performance of DGEMM for different n_b using 4 GPUs (The size of $[A]$ is $111,104$).

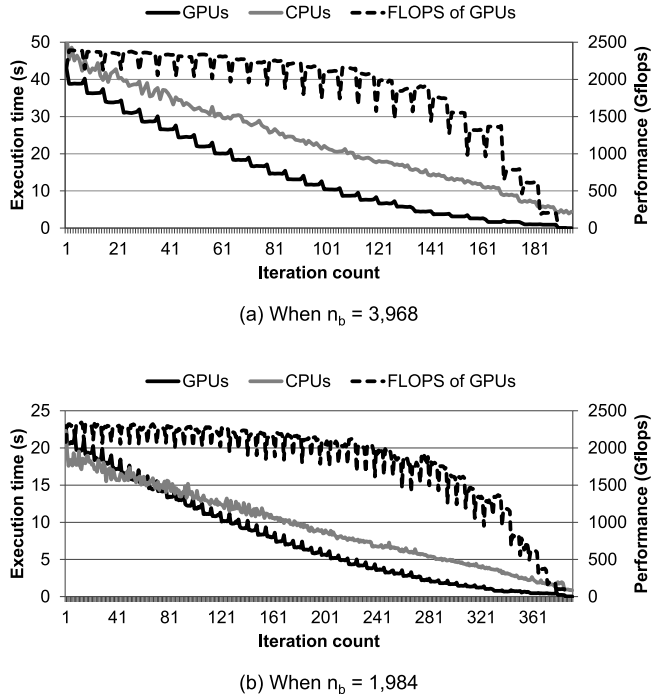

Fig. 11. The execution time of the CPUs, the execution time of the GPUs, and the performance of the GPUs in Gflops in each iteration of the baseline implementation.

for the LINPACK implementation to fully utilize multiple GPUs in a node.

5.3 Baseline Implementation

We evaluate the performance of our baseline implementation (described in Section 4.1) on the target cluster. The coefficient matrix is a $777{,}728 \times 777{,}729$ matrix (with a size 92 GB per node). Two values of n_b, 3,968 and 1,984, are used for the evaluation. The target cluster achieves 68.43 Tflops with $n_b = 3{,}968$ and 82.61 Tflops with $n_b = 1{,}984$. The per-node performance is 1,397 Gflops and 1,686 Gflops, respectively.

Fig. 11 shows the execution time of the CPUs, the execution time of the GPUs, and the performance of the GPUs in Gflops in each iteration of the baseline implementation. The numbers are obtained from node (0, 0) in the 7×7 grid. When $n_b = 3{,}968$, the performance of the GPUs is more than 2.3 Tflops in the first one third of the iterations, but this is not exploited in the baseline implementation because the execution time of the CPUs is much larger than that of the GPUs in all iterations. On the other hand, when $n_b = 1{,}984$, the gap between the execution time of the CPUs and that of the GPUs is much smaller. However, using a small n_b slightly degrades the performance of GPUs.

To perform DGEMM for $[A]_{pq}$, $b^2 \cdot n_b$ floating point operations, transferring $8b \cdot n_b$ bytes to a GPU (i.e., CopyL2 and WriteL2 steps in Section 3), and transferring $8b^2$ bytes from a GPU (i.e., ReadA and Sum steps in Section 3) are required. As n_b decreases, the amount of computation and the amount of the CPU-to-GPU transfer decrease linearly. However, the amount of the GPU-to-CPU transfer remains constant. This implies that the time taken to perform the CPU-to-GPU transfer and GPU-to-CPU transfer for a single

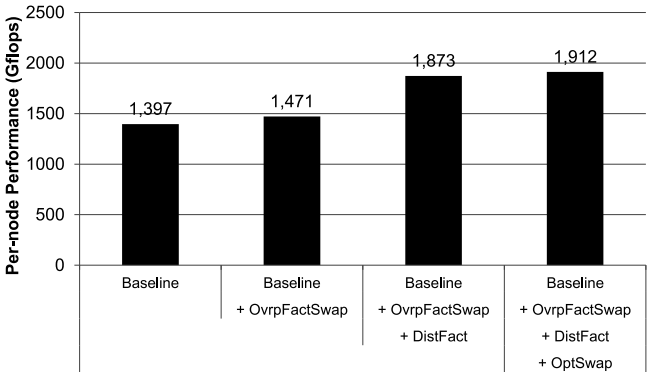

Fig. 12. The per-node performance of the baseline and optimized implementations ($n_b = 3{,}968$).

floating point operation increases as n_b decreases. The slow data transfer between CPUs and GPUs typically does not affect performance because the data transfer is completely hidden by the kernel execution. However, if the value of n_b is less than some threshold value, the transfer time becomes large and cannot be hidden. For multiple GPUs, the threshold value is much larger because data transfer becomes slow due to limited bandwidth of main memory and PCI-E bus contention between multiple GPUs. In our case, the threshold is about 1,984 and using a smaller n_b may hurt the performance of GPUs significantly.

The evaluation result tells us that to achieve good performance, a large n_b needs to be used to fully utilize multiple GPUs. In addition, we need to optimize the phases performed by the CPUs. None of the factorization, broadcast, and swap phases have negligible execution time on the CPUs. Thus, all of them can be optimized to improve the performance of the CPUs.

5.4 Effect of Optimizations

We measure the performance of our LINPACK implementation by incrementally applying the three optimization techniques, OvrpFactSwap in Section 4.2, DistFact in Section 4.3, and OptSwap in Section 4.4, to the baseline implementation (Baseline). Fig. 12 shows the per-node performance in Gflops for the optimized implementations. The performance is improved as more optimizations are applied. Finally, our optimized LINPACK implementation achieves 93.69 Tflops for the target cluster. The per-node performance is 1,912 Gflops, which is 74 percent of the DGEMM performance with four GPUs and 46 percent of the theoretical peak performance of a node.

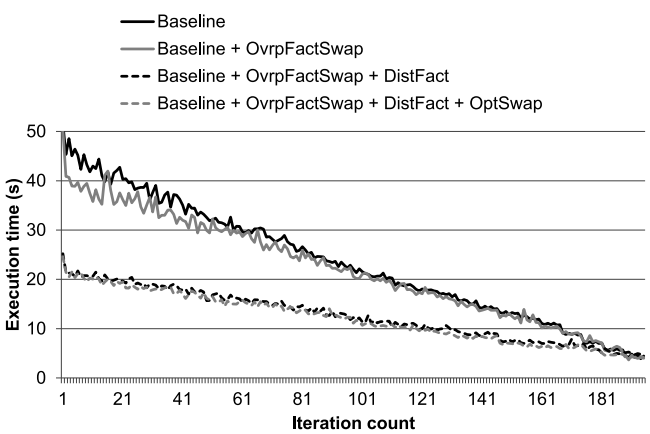

Fig. 13. The execution time of the CPUs in each iteration of the baseline and optimized implementations ($n_b = 3{,}968$).

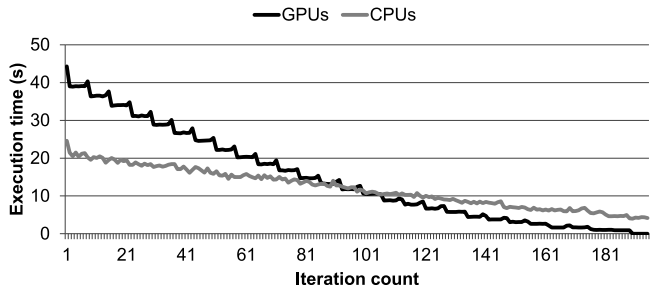

Fig. 14. The execution time of the CPUs and the execution time of the GPUs in Gflops in each iteration of the final optimized implementation ($n_b = 3{,}968$).

Fig. 13 shows the execution time of the CPUs in each iteration for the baseline and optimized implementations. Overlapping between the factorization and swap phase reduces the execution time of the CPUs by 6.3 percent. Distributing the factorization phase dramatically reduces the execution time of the CPUs by 42 percent. The optimized swap phase reduces the execution time by 4.9 percent. Using our optimization techniques, the execution time of the CPUs is reduced by overall 48.5 percent compared to the baseline implementation.

Fig. 14 shows the execution time of the CPUs and the execution time of the GPUs in each iteration of the final optimized implementation. Execution parameters are the same as those in Fig. 11a. We see that the execution time of the CPUs is completely hidden by that of GPUs in 48 percent of the iterations that occupy 74 percent of the total execution time. In Fig. 11a, we see that the execution time of the CPUs is not hidden at all by that of the GPUs with the baseline implementation.

5.5 Power Efficiency of Multiple GPUs

We measure the actual power consumption while our LINPACK implementation performs the blocked LU decomposition, under the rule of Level 1 power measurement for the Green500 list [22]. The power efficiency can be computed by dividing the LINPACK performance (Flops) by the power consumption (Watt). Fig. 15 shows the power efficiency of

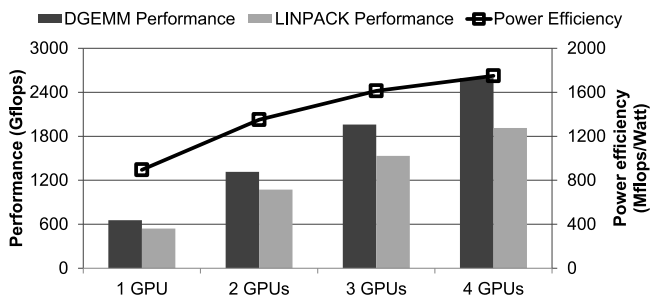

Fig. 15. The performance of DGEMM, the per-node performance of LINPACK, and the power efficiency of LINPACK in Mflops/Watt for different number of GPUs.

TABLE 1
Comparison of LINPACK Implementations

Implementation	Accelerators per Node in the Target Cluster	Accelerated Routine	n_b	Performance of DGEMM (Gflops)		Per-node Performance of HPL (Gflops)
Rohr *et al.* [13]	1 AMD GPU	DGEMM	1024	1 GPU 1 GPU + CPUs	465 623	526
Fatica [12]	2 NVIDIA GPUs	DTRSM and DGEMM	960	1 GPU 1 GPU + CPUs	≈ 70 ≈ 110	157
Yang *et al.* [3]	2 AMD GPUs	DGEMM	1216	2 GPUs	≈ 210	110
Endo *et al.* [1]	2 NVIDIA GPUs + 1 ClearSpeed, or 1 ClearSpeed	DTRSM and DGEMM	1152	1 GPU 1 ClearSpeed	76 60	120 (on average)
Heinecke *et al.* [23]	2 Intel Xeon Phis	DGEMM	1200	1 Xeon Phi	944	1758
Kistler *et al.* [2]	4 Cell BE processors	DTRSM and DGEMM	128	1 Cell processor	102	350
Our work	4 AMD GPUs	DTRSM and DGEMM	3968	4 GPUs	2592	1912

our LINPACK implementation. We vary the maximum number of GPUs per node used for DTRSM-DGEMM. As the number of GPUs per node increases, the performance gap between DGEMM and the LINPACK implementation widens. This is because the execution time of CPUs becomes a performance bottleneck as the iterations go on. As a result, the performance improvement by the GPUs is not fully exploited. With a single GPU, the LINPACK implementation achieves 82 percent of the DGEMM performance while it achieves just 74 percent of the DGEMM performance with four GPUs. However, the power efficiency of LINPACK is increased as more GPUs are used per node. This tells us that using multiple GPUs per node can be more power efficient although their performance is not fully exploited because the CPUs become the performance bottleneck.

6 Related Work

Some LINPACK implementations for heterogeneous clusters have been proposed and evaluated in their target clusters. Table 1 summarizes the prior implementations of the LINPACK benchmark for heterogeneous clusters, and compares them with our approach. All of them are based on HPL. They execute DGEMM or both DGEMM and DTRSM on accelerators. All implementations except that by Kistler et al. use a small n_b between 960 and 1,216. Since each node in their target clusters has only one or two accelerators, such small n_b does not degrade the performance of the accelerators. Kistler et al. [2] uses a much smaller $n_b(= 128)$ because they store the coefficient matrix in the device memory of accelerators (Cell BE processors). Then, the data transfer between the main memory and the device memory is not required for DGEMM. Thus, this extremely small n_b does not hurt the performance of DGEMM.

Since the prior implementations use a small n_b and also the performance gap between CPUs and accelerators in their target clusters is not too big, the execution time of the CPU side is not a performance bottleneck. Thus, they are focused on efficient execution of DGEMM. Rohr et al. [13] proposes an efficient DGEMM implementation called CALDGEMM running on AMD Cypress GPUs, and optimizes HPL using CALDGEMM. Since CALDGEMM uses AMD CAL that is a low-level programming environment for AMD GPUs, it can achieve 465 Gflops on a single GPU, which is 85.5 percent of the theoretical peak. Rohr et al. [13], Fatica [12], Yang et al. [3], and Heinecke et al. [23] use both CPUs and accelerators to execute DGEMM. They divide the matrix A into two parts and compute one of them on accelerators and the other on CPUs. Similarly, Endo et al. [1] proposes a load balancing method to utilize all accelerators and CPUs for executing DGEMM. However, optimizations for the CPU side are not the main concern of the prior implementations. Rohr et al. [13], Endo et al. [1], Kistler et al. [2], and Heinecke et al. just overlap a part of the CPU-side execution with the execution of accelerators. Fatica [12] and Yang et al. [3] just replace the BLAS library with the accelerated version and do not optimize HPL further.

Our approach is significantly different from those prior implementations. Each node in the cluster has multiple GPUs as accelerators, and a large value of n_b is required to fully utilize the GPUs. Hence, the CPU side becomes a performance bottleneck. Consequently, optimizations for the CPU side are necessary. We show that exploiting multiple GPUs in a node is more efficient in terms of the power efficiency, as described in Section 5.5.

Du et al. [24] introduce an implementation of the blocked LU decomposition algorithm for GPU clusters. Their work is based on ScaLAPACK [16]. Each node in their target cluster contains 3 NVIDIA GPUs, but their implementation is not optimized for multi-GPUs. As a result, it only achieves 1.7-1.8$\times$ speedup compared to the original CPU-only ScaLAPACK. Note that our per-node LINPACK performance is 1,912 Gflops, and it is 7.5 times higher than the theoretical peak performance of two Intel Xeon E5-2650 CPUs installed in a single node.

Rohr et al. [25] presents a DGEMM implementation for multiple GPUs. They argue that using multiple GPUs may improve the energy-efficiency of HPL. They also show that a large n_b is required to achieve high performance in HPL if multiple GPUs are used. However, they do not actually

TABLE 2
Comparison of the State-of-the-Art CPU/GPU Clusters in the TOP500 List of November 2012 [10]

Name	Rank in TOP500	CPU	GPU	Interconnection Network	Per-node Performance of HPL (Gflops)
SANAM	#52	2 × Intel Xeon E5-2650	2 × AMD FirePro S10000 (4 GPUs)	InfiniBand FDR	2006
HA-PACS	#51	2 × Intel Xeon E5-2670	4 × NVIDIA M2090 (4 GPUs)	InfiniBand QDR	1622
The target cluster	#277	2 × Intel Xeon E5-2650	4 × AMD Radeon HD 7970 (4 GPUs)	InfiniBand QDR	1912

integrate their DGEMM implementation into HPL and evaluate the performance.

Table 2 describes the state-of-the-art CPU/GPU clusters in the TOP500 list of November 2012. The per-node performance of SANAM and HA-PACS are the first and the third among the CPU/GPU clusters in the TOP500 list, respectively. Note that their CPUs (Intel Xeon E5-2670 of HA-PACS) or InfiniBand switches (InfiniBand FDR of SANAM) are better than those of the target cluster. As we mentioned before, the performance of the LINPACK benchmark largely depends on the performance of the computation of CPUs and the inter-node communication. Table 2 shows that the target cluster (Chundoong) achieves comparable or better performance than other clusters using our LINPACK implementation. Since the LINPACK implementations for SANAM and HA-PACS are not open to public, we cannot analyze their algorithms deeply.

Volkov and Demmel [26], Quintana-Ortí et al. [27], and Tomov et al. [28] propose efficient algorithms for solving dense linear systems on multiple GPUs in a single node. They store the whole coefficient matrix in the global memory of GPUs and update the matrix by GPUs themselves. But these algorithms are not suitable for clusters because many parts of the coefficient matrix should be repeatedly copied to the main memory to communicate with other nodes, resulting in significant data transfer overhead between CPUs and GPUs.

7 Conclusions

We introduce an MPI-OpenCL implementation of the LINPACK benchmark for heterogeneous clusters of multi-GPU nodes. Our LINPACK implementation executes DTRSM and DGEMM of the update phase on GPUs, and the factorization, broadcast, and swap phases on CPUs. An efficient multi-GPU implementation of the DTRSM and DGEMM routines, called DTRSM-DGEMM, is used for our LINPACK implementation. It combines the DTRSM and DGEMM routines to minimize the data transfer between CPUs and GPUs. Also it dynamically balances the workload across GPUs.

We show that a bigger n_b is required to achieve better DGEMM performance, but it also increases the fraction of the CPU workload. When $n_b = 3,968$, the performance improvement by the GPUs in DGEMM is not reflected to the overall performance of our baseline implementation because the CPUs are the performance bottleneck in all iterations. We propose four optimization techniques that reduce the overhead of the CPUs. By applying these techniques to the baseline implementation, the execution time of the CPUs are significantly reduced by 48.5 percent and the per-node LINPACK performance is improved from 1,397 Gflops to 1,912 Gflops.

Even if those optimization techniques are applied, the execution time of the CPUs still remains as a major bottleneck. Thus, the performance improvement by the four GPUs is not still fully exploited in our LINPACK implementation. On the other hand, our experimental result indicates that using more GPUs per node for the LINPACK benchmark improves power efficiency significantly.

Compared to a small scale heterogeneous cluster, a large scale heterogeneous cluster that consists of thousands of nodes will slightly increase the amount of communication [29] while it makes the amount of computation performed by GPUs remain constant for LINPACK. Thus, proposed optimization techniques for the CPU side will be still important.

We use a portable mix of two programming models, MPI and OpenCL, for heterogeneous clusters. Our implementation of the LINPACK benchmark and its optimization techniques handle multiple GPUs in each node efficiently and achieve high performance on the target cluster. Our implementation is functionally portable to various heterogeneous clusters without any modification. In addition, the host program, which is the major part of our implementation and includes the proposed optimization techniques, does not suffer from the performance portability problem because it runs on CPUs. Two OpenCL kernels (DTRSM and DGEMM) may need to be modified to achieve better performance on different accelerators. Note that the length of these kernels is less than 200 lines. In addition, some previous approaches propose autotuning techniques to find the best BLAS kernel for the target accelerator automatically (e.g., [30]). This shows that MPI-OpenCL is a promising model for clusters of multi-GPU nodes to achieve high performance in HPL and other scientific applications.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2013R1A3A2003664). ICT at Seoul National University provided research facilities for this study.

References

[1] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–8.

[2] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," in *Proc. 14th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2009, pp. 241–250.

[3] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2010, pp. 19–28.
[4] Khronos Group. OpenCL-The open standard for parallel programming of heterogeneous systems [Online]. Available: http://www.khronos.org/opencl/
[5] AMD. AMD accelerated parallel processing (APP) SDK [Online]. Available: http://developer.amd.com/sdks/amdappsdk/pages/
[6] IBM. OpenCL developer kit for linux on power [Online]. Available: http://alphaworks.ibm.com/tech/opencl
[7] Intel. Intel SDK for OpenCL applications 2012 [Online]. Available: http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
[8] NVIDIA. NVIDIA OpenCL [Online]. Available: http://developer.nvidia.com/opencl
[9] Seoul National University and Samsung. SNU-SAMSUNG OpenCL framework [Online]. Available: http://opencl.snu.ac.kr
[10] TOP500 supercomputer sites [Online]. Available: http://top500.org/
[11] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL-A portable implementation of the high-performance linpack benchmark for distributed-memory computers [Online]. Available: http://www.netlib.org/benchmark/hpl/
[12] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," in *Proc. 2nd Workshop General Purpose Process. Graph. Process. Units*, 2009, pp. 46–51.
[13] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, "Optimized HPL for AMD GPU and multi-core CPU usage," *Comput. Sci.-Res. Develop.*, vol. 26, pp. 153–164, 2011.
[14] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency Comput.: Practice Exp.*, vol. 15, no. 9, pp. 803–820, 2003.
[15] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. V. Vorst, *Numer. Linear Algebra for High-Perform. Comput.* Philadelphia, PA, USA: SIAM, 1998.
[16] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines," *Sci. Program.*, vol. 5, no. 3, pp. 173–184, Aug. 1996.
[17] LAPACK-Linear algebra PACKage [Online]. Available: http://www.netlib.org/lapack/
[18] AMD. AMD core math library for graphics processors (ACML-GPU) [Online]. Available: http://developer.amd.com/libraries/acmlgpu/pages/default.aspx
[19] N. Nakasato, "A fast GEMM implementation on the cypress GPU," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 50–55, 2011.
[20] Intel, Intel Math Kernel Library (Intel MKL) [Online]. Available: http://software.intel.com/en-us/intel-mkl
[21] The green500 list [Online]. Available: http://green500.org/
[22] (2013, May). Energy efficient high performance computing power measurement methodology [Online]. Available: http://green500.org/sites/default/files/eehpcwg/EEHPCWG_PowerMeasurementMethodology.pdf, pp. 13–27.
[23] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. Shet, G. Chrysos, and P. Dubey, "Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 126–137.
[24] P. Du, S. Tomov, and J. Dongarra. (2012, Sep.). Providing GPU capability to LU and QR within the ScaLAPACK framework, LAPACK Working Note, Tech. Rep. 272 [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn272.pdf
[25] D. Rohr, M. Bach, M. Kretz, and V. Lindenstruth, "Multi-GPU DGEMM and high performance linpack on highly energy-efficient clusters," *IEEE Micro*, vol. 31, no. 5, pp. 18–27, Sep. 2011.
[26] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. ACM/IEEE Conf. Supercomput.*, 2008, pp. 31:1–31:11.
[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 *Proc. 14th ACM SIGPLAN Symp. Principles Practice Parallel Program*, 2009, pp. 121–130.
[28] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops and Phd Forum*, 2010, pp. 1–8.
[29] HPL scalability analysis [Online]. Available: http://www.netlib.org/benchmark/hpl/scalability.html
[30] C. Jang. GATLAS GPU automatically tuned linear algebra software [Online]. Available: http://golem5.org/gatlas/

Gangwon Jo received the BS degree in computer science from Seoul National University in 2010. He is working toward the PhD degree in the Department of Computer Science and Engineering at Seoul National University. He received the Graduate Student Fellowship from the Korea Foundation for Advanced Studies. His research interests include programming environments and runtime systems for heterogeneous systems, high-performance computing applications, and inter-node communication. He is a student member of the IEEE.

Jeongho Nah received the BA degree in maritime administration science from Korea Maritime University in 2006, and the MS degree in computer science from Seoul National University. He is working toward the PhD degree in the Department of Computer Science and Engineering at Seoul National University. His research interests include compiler, OS, and architecture interactions for multicores/manycores, compilers/runtimes for multicore/manycores, and analysis and optimization techniques for JavaScript.

Jun Lee received the BS degree in vocational education and workforce development from Seoul National University in 2008. He is working toward the PhD degree in the Department of Computer Science and Engineering at Seoul National University. His research interests include compilers, computer architectures, and parallel programming.

Jungwon Kim received the BS degree in computer science and engineering and the PhD degree in electrical engineering and computer science both from Seoul National University in 2006 and 2013, respectively. He is currently a postdoctoral research associate at Oak Ridge National Laboratory. His research interests include compilers, runtime systems for multicores/manycores, and scalable heterogeneous HPC computing. He is a member of the IEEE.

Jaejin Lee received the BS degree in physics from Seoul National University in 1991, the MS degree in computer science from Stanford University in 1995, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1999. He is a professor in the Department of Computer Science and Engineering at Seoul National University, Korea, where he has been a faculty member since September 2002. Before joining Seoul National University, he was an assistant professor in the Computer Science and Engineering Department at Michigan State University. He received the IBM Cooperative Fellowship and a fellowship from the Korea Foundation for Advanced Studies during his PhD study. His research interests include compilers, computer architectures, embedded computer systems, and systems in high-performance computing. He is a member of the IEEE and the ACM.