PAPER Special Section on Parallel and Distributed Computing and Networking

Computation-Communication Overlap of Linpack on a GPU-Accelerated PC Cluster

Junichi OHMURA[†a)], ***Nonmember***, **Takefumi MIYOSHI**[†b)], **Hidetsugu IRIE**[†c)], ***and*** **Tsutomu YOSHINAGA**[†d)], ***Members***

SUMMARY In this paper, we propose an approach to obtaining enhanced performance of the Linpack benchmark on a GPU-accelerated PC cluster connected via relatively slow inter-node connections. For one node with a quad-core Intel Xeon W3520 processor and a NVIDIA Tesla C1060 GPU card, we implement a CPU–GPU parallel double-precision general matrix–matrix multiplication (*dgemm*) operation, and achieve a performance improvement of 34% compared with the GPU-only case and 64% compared with the CPU-only case. For an entire 16-node cluster, each node of which is the same as the above and is connected with two gigabit Ethernet links, we use a computation-communication overlap scheme with GPU acceleration for the Linpack benchmark, and achieve a performance improvement of 28% compared with the GPU-accelerated high-performance Linpack benchmark (HPL) without overlapping. Our overlap GPU acceleration solution uses overlaps in which the main inter-node communication and data transfer to the GPU device memory are overlapped with the main computation task on the CPU cores. These overlaps use multi-core processors, which almost all of today's high-performance computers use. In particular, as well as using a CPU core for communication tasks, we also simultaneously use other CPU cores and the GPU for computation tasks. In order to enable overlap between inter-node communication and computation tasks, we eliminate their close dependence by breaking the main computation task into smaller tasks and rescheduling. Based on a scheme in which part of the CPU computation power is simultaneously used for tasks other than computation tasks, we experimentally find the optimal computation ratio for CPUs; this ratio differs from the case of parallel *dgemm* operation of one node.

key words: *parallel processing, multi-core processor, GPU, computation-communication overlap*

1. Introduction

The graphic processing unit (GPU) has become an integral part of today's mainstream computing systems. Recently, there has been a substantial improvement in the performance and capabilities of GPUs. The modern GPU is a powerful graphics engine as well as a highly parallel programmable processor featuring peak arithmetic and memory bandwidths that are substantially superior to those of its CPU counterpart. Many studies on the use of GPUs for numerical computations, such as matrix–matrix multiplications, have been reported.

Organizing PC clusters with multi-core processors and GPUs is also getting popular. This makes the efficient parallel use of CPUs and GPUs more important. Meanwhile, in respect of inter-connects, faster ones, such as Inifiniband, are commonly used in highend supercomputers. However, compared to the GPUs, which are also widely shipped as graphics engines in the consumer market, the latest interconnects are relatively costly. So, our experimental environment utilizes two gigabit Ethernet links to connect 16 GPU-accelerated PCs, each of which has a quad-core Intel Xeon W3520 processor and a NVIDIA Tesla C1060 GPU card. For this type of clusters connected via relatively slow inter-connects, its bandwidth might be a potential bottleneck. This bottleneck in our experimental environment can be recognized from Fig. 1.

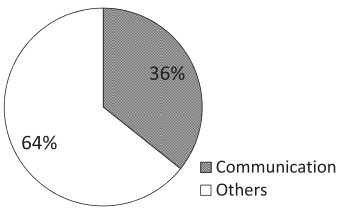


Fig. 1 A ratio of execution time occupied by communication tasks to overall elapsed time when using HPL in our experimental environment (see Tables 1 and 2, and Fig. 11 for details).

As we discuss in 3.3, CPUs provide 35% of computation power in our experimental environment. Therefore, it is important to overlap computation and communication. In our present work [1], we examined an efficient implementation of Linpack. Our approach is based on the *Hybrid MPI-OpenMP with thread-to-thread communication (Hybrid TC)* model introduced by [9].

This paper show further performance improvement of Linpack by carefully tuning the overlap not only among nodes but also between CPU and GPU on a single node. Our Linpack implementation results in 28% performance improvement compared with the nonoverlap GPU acceleration case introduced by Fatica [5] on the above-mentioned cluster environment.

The rest of the paper is organized as follows. Section 2 introduces related studies that also examine GPU-accelerated PC clusters. Section 3 describes the local *dgemm* operation that uses CPU–GPU parallel processing. In Sect. 4, we discuss computation-communication overlap of the Linpack benchmark and explain our proposed ap-

Manuscript received January 6, 2011.
Manuscript revised June 9, 2011.

[†]The authors are with the Graduate School of Information Systems, University of Electro-Communications, Chofu-shi, 182–8585 Japan.
a) E-mail: ohmura@comp.is.uec.ac.jp
b) E-mail: misyohi@is.uec.ac.jp
c) E-mail: irie@is.uec.ac.jp
d) E-mail: yoshinaga@is.uec.ac.jp

DOI: 10.1587/transinf.E94.D.2319

proach in detail. Section 5 shows the differences between other Linpack implementations and ours, Sect. 6 the experimental results, and Sect. 7 concludes the paper.

2. Related Studies

The HPL is a well-known example of numerical computations; it solves random dense linear systems in double-precision arithmetic, and involve time-consuming tasks to deal with matrix–matrix multiplication. Recent studies dealing with general-purpose computing on GPUs (GPGPU) introduce accelerations by utilizing the GPUs on a GPU-accelerated PC clusters.

Ohshima et al. examined CPU and GPU parallel matrix–matrix multiplications on a single node [2], a procedure that improves the local *dgemm* performance. There are several frameworks and libraries to exploit the power of CPU and GPU, such as StarPU [3] and MAGMA [4]. Although these works successfully utilize computation power on a single node, computation resources distributed over many nodes connected via slow inter-connects are not considered.

Fatica showed the performance enhancement of HPL by only spreading each matrix–matrix multiplication operations (*dgemm* and *dtrsm*) over CPUs and GPUs [5]. Endo et al. presented implementations and evaluation results on TSUBAME 1.2 [6], [7] and TSUBAME 2.0 [8]; These works assumed fast but expensive inter-connects, such as Infiniband. Meanwhile, because one of the advantages of GPGPU is its low cost, it is important to consider good use of CPU computation powers during inter-node communications on clusters utilizing inexpensive but basic and relatively slow inter-connects. Unlike their approaches, we introduce a smart intra-node task assignment to exploit these CPU computation powers. More detailed differences are discussed in Sect. 5.

Meanwhile, in multi-core cluster systems without GPU accelerators, some other contributions have been made for improving the computing power by applying hybrid MPI-OpenMP models [9], [10]. It can exploit the power of CPU cores even if slow inter-communication tasks consume many CPU times. In our approach, we employ this model as a base strategy. In the case of clusters with GPU accelerators, we must consider the granularity of tasks. We describe it in Sect. 4.

3. Local Parallel *dgemm*

In basic linear algebra subprograms (BLAS), matrix–matrix multiplication is defined as a $C = \alpha \times A \times B + \beta \times C$ computation, where A, B, and C are matrices and α and β are scalars [11].

3.1 CPU-Only *dgemm*

As shown in Fig. 2, if A(M,K), B(K,N), and C(M,N) are input matrices, a *dgemm* call will compute $C = \alpha AB + \beta C$.

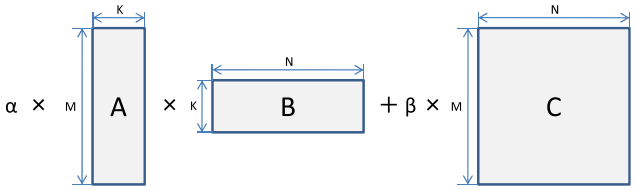


Fig. 2 Matrix–matrix multiplication: $C := \alpha * op(A) * op(B) + \beta * C$.

Several high-performance BLAS implementations are available: ATLAS [12], MKL [13], AMD ACML [14], and GotoBLAS2 [15]. All these procedures include implementation of matrix–matrix multiplication in level 3, with which users can develop their own applications using the library. We use GotoBLAS2 for our experiments because of its powerful performance and simplicity of installation.

3.2 GPU-Only *dgemm*

For GPU computing, compute unified device architecture (CUDA [16]) is a parallel programming model and software environment designed to expose the parallel capabilities of GPUs. CUDA extends *C* by allowing the programmer to define *C* functions, called *kernels*, which, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once as in the case of regular *C* functions. The software environment also provides a BLAS library (CUBLAS), which includes matrix–matrix multiplication. In this paper, GPU-only *dgemm* is performed by the CUBLAS library instead of by specialized *kernels*.

3.3 CPU–GPU Parallel *dgemm*

The basic concept of CPU–GPU parallel matrix–matrix multiplication is very simple. The original matrix A has $K \times (Mc + Mg)$ elements, and matrix B has $N \times K$ elements. After the *dgemm* operation, there should therefore be $N \times (Mc + Mg)$ elements in matrix C. Mc and Mg are scalars that depend on the capabilities of a CPU and a GPU. Matrices A and C can be viewed as the union of two submatrices $A = Ag \cup Ac$ and $C = Cg \cup Cc$, where Ag and Cg denote the parts of A and C allocated to a GPU, and Ac and Cc denote the parts of A and C allocated to a CPU.

In other words, we can divide the *dgemm* operation into two completely independent parts (a CPU side and a GPU side), and execute it in parallel (Fig. 3).

Figures 4 and 5 show the flow and pseudocode of the local parallel *dgemm*. In this implementation, we first send the matrices needed from the GPU to the device memory. Once the data have been transferred, we call the CUBLAS function and the GotoBLAS2 function in this order. Because the CUBLAS call returns immediately without blocking (the following cublasGetMatrix(), instead of this non-blocking call, does not return until the *dgemm* computation on the GPU is completed), this simple implementation enables the overlap.

Figure 6 shows the performance of the CPU–GPU par-

allel *dgemm* operation and the optimum experimental values of the computation ratio $R = Ac/A = Mc/(Mc+Mg)$ for matrix sizes 4096. For this matrix size, the optimal R value is around 0.34. We conducted similar experiments for various matrix sizes, e.g., 2048 and 8192, but the optimal R value is fixed. The hardware and software experimental environments are summarized in Table 1 and Table 2, respectively.

Note: It is known that the *dgemm* function call in CUBLAS maps to several different kernels, depending on the size of the matrices, and the best performance is achieved when M is a multiple of 64. So the experiments use computation ratios that satisfy this condition.

The calculation used to obtain the theoretical value of R is as follows.

P_{CPU} performance of CPU (GFlops)
P_{GPU} performance of GPU (GFlops)
T_{CPU} *dgemm* execution time on CPU
T_{GPU} *dgemm* execution time on GPU

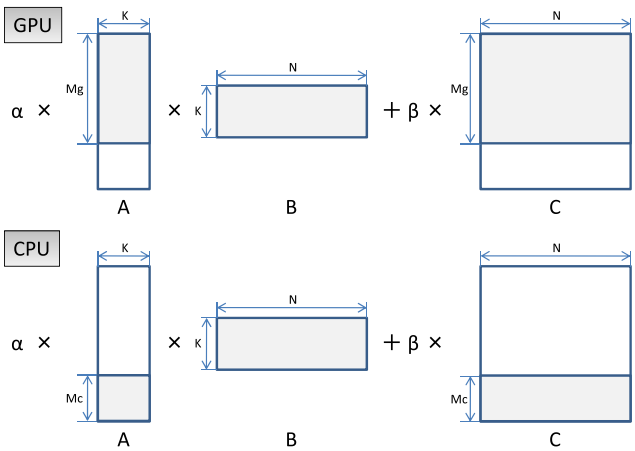


Fig. 3 A model of CPU–GPU parallel *dgemm*.

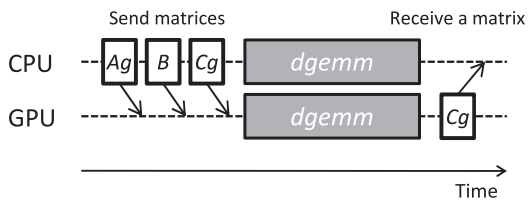


Fig. 4 A flow of the local parallel *dgemm*.

$$T_{CPU}(M,K,N) = \frac{2MKN}{P_{CPU}}$$
$$T_{GPU}(M,K,N) = \frac{2MKN}{P_{GPU}}$$

Note: A *dgemm* call results in $2MKN$ operations.

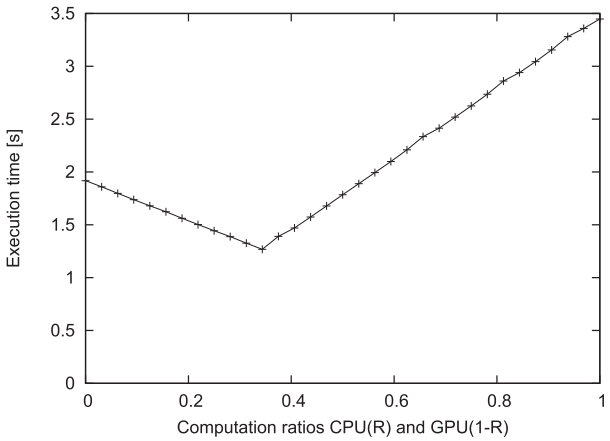


Fig. 6 CPU–GPU parallel *dgemm* performance for a matrix size of 4096.

Table 1 Hardware experimental environment (single node).

CPU	Intel Xeon W3520
CPU clock rate	2.67 GHz
CPU number of cores	4
Memory	6 GB
L2 cache	1 MB
L3 cache	8 MB
CPU IPC	4 (double precision)
CPU peak performance	42.72 GFlops
GPU	Tesla C1060
GPU clock rate	1.30 GHz
GPU number of cores	240
GPU memory	4 GB
GPU IPC	30 (double precision)
GPU peak performance	78 GFlops
Graphics Bus	PCI-Express 2.0
Quick Path Interconnect	25.6 GB/s

Table 2 Software experimental environment (single node).

OS	CentOS release 5.3
Compiler	Intel C compiler 11.1
CPU BLAS	GotoBLAS2 1.13 (USE_OPENMP=1)
GPU BLAS	CUBLAS 2.3
MPI library	Open MPI 1.4.1 (–enable-mpi-threads)

```
cublasSetMatrix(Mg, K, A, devptrA)  // sends matrix Ag to the device memory
cublasSetMatrix(K , N, B, devptrB)  // sends matrix B  to the device memory
cublasSetMatrix(Mg, K, C, devptrC)  // sends matrix Cg to the device memory

cublasDgemm(Mg, N, K, devptrA, devptrB, devptrC)  // dgemm on the GPU (non-blocking)
dgemm      (Mc, N, K, A+Mg   , B      , C+Mg)     // dgemm on the CPU

cublasGetMatrix(Mg, K, devptrC, C)  // receives updated matrix Cg from the device memory
```

Fig. 5 A pseudocode of the local parallel *dgemm*.

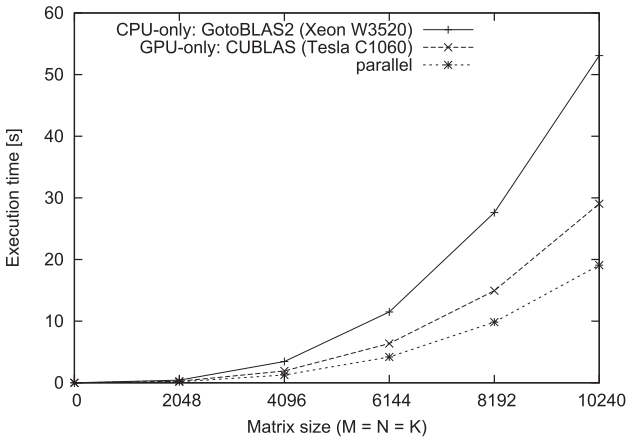


Fig. 7 CPU–GPU parallel *dgemm* performance versus that of CPU-only and GPU-only *dgemm* on a single node.

As Fig. 4 shows, the highest performance is achieved when the time for the CPU side is the same as that for the GPU side *dgemm*:

$$\begin{cases} T_{CPU}(Mc, K, N) = T_{GPU}(Mg, K, N) \\ M = Mc + Mg \end{cases}$$

Therefore,

$$\frac{2McKN}{P_{CPU}} = \frac{2MgKN}{P_{GPU}}$$
$$R = \frac{Mc}{M} = \frac{P_{CPU}}{P_{CPU} + P_{GPU}}$$

In our experimental environment,

$$R = \frac{P_{CPU}}{P_{CPU} + P_{GPU}} = \frac{42.72}{42.72 + 78} \approx 0.35$$

The experimental value of R is nearly the same as the theoretical value; we can therefore predict the optimal division point using the above equation.

The performance of the CPU–GPU parallel *dgemm* operation is compared with that of the CPU-only and GPU-only *dgemm* operations; this comparison is shown in Fig. 7. The parallel *dgemm* operation is more effective than both CPU-only and GPU-only operations. In the case of the parallel *dgemm* operation, the execution time of matrix–matrix multiplication in a single node is reduced by 34% compared with the GPU-only case, and by 64% compared with the CPU-only case.

4. Computation-Communication Linpack Overlap

4.1 Equation System

The benchmark used in Linpack is solution of a dense system of linear equations. One of the Linpack benchmark implementations is the HPL. The system of linear algebra equations has the following form:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases} \quad (1)$$

where the vectors $x_1 \ldots x_n$ are unknown values that need to be determined. The coefficients a_{ij} and the vectors b_i on the right-hand side are all generated randomly. The accuracy of the solution is double precision for 64-bits of double-data type.

4.2 LU Decomposition Algorithm

Several types of algorithm are used to solve linear problems. Of these, the LU decomposition algorithm is considered to be the most suitable. This algorithm is based on a simple principle, and it is also effective in terms of computing performance. Both the original HPL benchmark and our approach for GPU acceleration of the Linpack solutions employ the LU decomposition algorithm.

The system can be expressed in matrix form as follows:

$$A \times x = b \quad (2)$$

where A is the matrix of the coefficients and b is a column vector. We can consider matrix A as two submatrices:

$$LU = A \quad (3)$$

where L is a lower triangular matrix and U is an upper triangular matrix. We can use this decomposition to solve the set:

$$L \times (U \times x) = b \quad (4)$$

First, we determine the vector y such that

$$L \times y = b \quad (5)$$

Next, we solve

$$U \times x = y \quad (6)$$

to determine the unknowns x.

The two equations above (5 and 6) can be solved by forward and backward functions. When the problem size N is sufficiently large, these equations take relatively less computation time than that for LU decomposition. We therefore focus solely on the LU decomposition algorithm.

4.3 Block Right-Looking LU Decomposition Algorithm

A square matrix A can be decomposed into lower and upper components using various methods. The block right-looking LU decomposition algorithm is one of the most effective methods for parallelization. We decompose the matrix A stepwise, using this algorithm. At each step, we set *nb*, the block size at which we perform the decomposition, the *nb* columns of the lower triangular matrix, and the *nb* rows of the upper triangular matrix.

A undergoes partial factorization at a certain point,

Table 3 Nonoverlap task list.

No.	Description	Cur. dep.	Prev. dep.
1+	Bcast(D)	-	7
2	Decom(D)	1	-
3*	*dtrsm*(L)	2	-
4*	*dtrsm*(U)	2	-
5+	Bcast(L)	3	-
6+	Bcast(U)	4	-
7*	*dgemm*(T)	5,6	-

*: GPU-accelerated tasks.
+: Communication tasks.

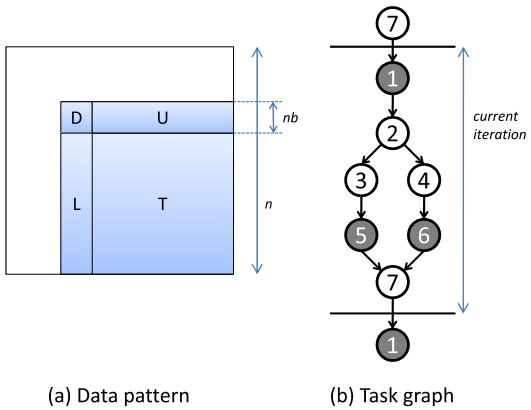


Fig. 8 A nonoverlap task flow of an iteration in the block right-looking LU decomposition algorithm.

from which the first nb columns of L and the first nb rows of U are evaluated. A is then expressed as follows:

$$A = \begin{bmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & & I \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ & A_{22} & A_{23} \\ & A_{32} & A_{33} \end{bmatrix} \tag{7}$$

where L_{11} and U_{11} are $nb \times nb$ matrices; the blocks A_{ij} are matrices that should be factorized from this point onward in the experiment.

We then factorize the next nb columns of L and nb rows of U in advance.

$$A = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & I \end{bmatrix} \times \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & A'_{33} \end{bmatrix} \tag{8}$$

Our aim is to determine L_{22}, L_{32}, U_{22}, and L_{23}, and submatrix A_{33}. From the two equations above (7 and 8), this can be done by solving the following equations using BLAS:

$$L_{22}U_{22} = A_{22} \tag{9}$$
$$L_{32}U_{22} = A_{32} \tag{10}$$
$$L_{22}U_{23} = A_{23} \tag{11}$$
$$A'_{33} = A_{33} - L_{32}U_{23} \tag{12}$$

4.4 Nonoverlap Task Flow and GPU Acceleration

The tasks of an iteration, and the data dependence of these tasks on the iteration, are listed in Table 3, data related to the iteration (submatrices D, L, U, and T) are outlined in Fig. 8 (a), and the task graph is depicted in Fig. 8 (b). In the table, "No." refers to the task number, "Cur. dep." refers to the operation in this iteration on which the current operation depends, and "Prev. dep." refers to the operation in the last iteration on which the current operation depends.

There are several broadcast tasks (1, 5, and 6) because the matrix A is divided into several $nb \times nb$ blocks distributed by a block cyclic scheme on the process grid. Using this approach, we can minimize the cost of inter-node communication. Submatrix D is the i_{th} block of the main diagonal. Submatrices L, U, and T represent the current parts of the lower, upper, and trailing matrices, respectively. Tasks 2 (Decom(D)), 3 (*dtrsm*(L)), 4 (*dtrsm*(U)), and 7 (*dgemm*(T)) are for Eqs. (9), (10), (11), and (12), respectively.

Of these tasks, all matrix–matrix multiplication computation tasks (two *dtrsm* tasks and one *dgemm* task) can be accelerated with the GPU by employing local parallel *dtrsm*/*dgemm* methods; the *dgemm* case is mentioned in Sect. 3. This acceleration is achieved by a simple replacement of the *dtrsm* and *dgemm* calls. Fatica exemined the performance of an HPL accelerated with GPUs by this replacement method [5].

As mentioned above, we also split each *dtrsm* operation into CPU-side and GPU-side computations in a similar way to the *dgemm* case. However, the optimal value of R which we obtained experimentally is 0.31; very slight improvement (less than 0.2% compared to the result for 0.35) of the overall performance is achieved. Implementation details of the GotoBLAS2 and/or the CUBLAS seems to result in this.

4.5 Computation and Inter-Node Communication Overlap

There are two overlap techniques employed for our acceleration. In this subsection, we describe one of them: an overlap between the computation and the inter-node communication.

Among all the tasks in a certain iteration, tasks 5 and 6 consume more than 90% of the overall communication cost, and task 7 consumes more than 90% of the overall computation cost. We use a small cluster for computation, and, hence, the matrix cannot be very large. Thus, the cost to *dtrsm(L)* and *dtrsm(U)* is very small. We can simply focus on tasks 5, 6, and 7, as they are also our main target for overlap between computation and inter-node communication. In order to enable the overlap, we first need to eliminate the dependence of the operation on these tasks.

As Fig. 8 (b) shows, all tasks of the current iteration in the nonoverlap case depend on task 7 of the previous iteration that updates the "previous" trailing matrix, including the "current" D, U, L, and T. In order to break this dependence into several smaller ones, we split task 7 into 7_1, 7_3, 7_4, and 7_7, which update D, U, L, and T, respectively. The new task list is shown in Table 4, the task dependence graph is shown in Fig. 9 (b), and the data pattern is shown in

Table 4 Computation-communication overlapped task list.

No.	Description	Cur. dep.	Prev. dep.
1+	Bcast(D)	-	7_1
2	Decom(D)	1	-
3*	*dtrsm*(L)	2	7_3
4*	*dtrsm*(U)	2	7_4
5+	Bcast(L)	3	-
6+	Bcast(U)	4	-
7_1 *	*dgemm*(D_1)	5,6	7_7
7_3 *	*dgemm*(L_1)	5,6	7_7
7_4 *	*dgemm*(U_1)	5,6	7_7
7_7 *	*dgemm*(T_1)	5,6	7_7

*: GPU-accelerated tasks.
+: Communication tasks.

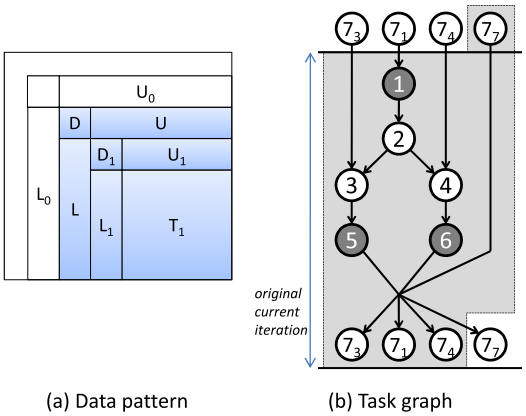


Fig. 9 A computation-communication overlapped task flow of an iteration in the block right-looking LU decomposition algorithm.

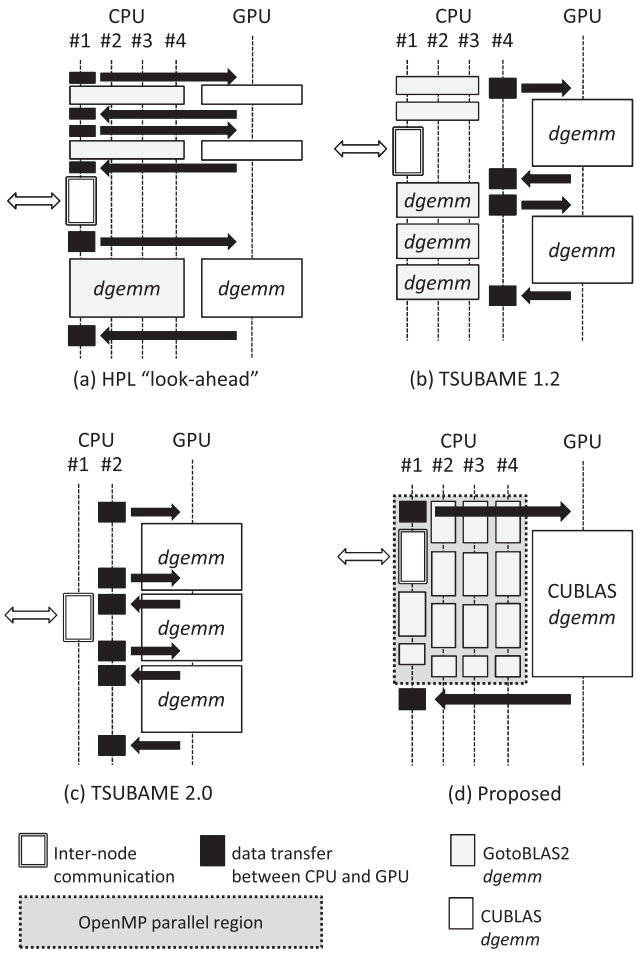


Fig. 10 Detailed task assignments for each implementation.

Fig. 9 (a).

Since tasks 5 and 6 (major communication) and the previous task 7_7 (major computation) are data independent, we can reconstruct the loop such that they are in the same iteration (shaded polygon in Fig. 9 (b)) and make them occur simultaneously.

4.6 Computation and CPU-to-GPU Data Transfer Overlap

In this subsection, we describe the other overlap between computation on the CPU and data transfer from the main memory to the device memory on the GPU.

As we showed in Sect. 3, in order to use the GPU for the computation, we first need to transfer the data of matrices Ag, B, and Cg. During this transmission, only one CPU core is working and the other CPU cores are not working. So we can enhance the usage rate of the CPU computational power by assigning these CPU cores to perform the computation during this period.

This technique can produce an advanced performance only when we use a hardware environment which ensures that data transfer to a GPU via the PCI-e, and the main memory accesses required by the computations, are done with no bandwidth constraints. In our cluster processors, the cores are able to access main memory and transfer data to the device through a quick path interconnect (QPI) at the same time. There are separate buses connected to the main memory and PCI-e slots. We can therefore obtain a performance improvement.

4.7 Intra-Node Task Assignment

We can use multiple CPU cores for computations by using the latest BLAS libraries, such as GotoBLAS2, as shown in Fig. 10 (a), (b) and (c). However, this method does not execute any other tasks on the CPU core instead of the computation task. We therefore use OpenMP #pragma directives to make our own computation threads, and let the inter-node communication and data transfer to the GPU device memory be executed on the master thread before it is used for the *dgemm* computations. CPU side *dgemm* operations are divided into smaller GotoBLAS2 tasks each of which is executed on either one CPU thread, and in order to make a lower overhead and a balanced workload distribution, last several tasks are relatively small.

In order to use GotoBLAS2 and OpenMPI together with OpenMP, we use the GotoBLAS2 library built with the setting USE_OPENMP=1, and the OpenMPI library configured with the –enable-mpi-threads option. These supports have an insignificant impact on the performance.

Moreover, we have a only one large GPU-side task for each iteration instead of dividing it into smaller ones, in order to minimize the number of the overheads of GPU-side task executions. Unlike Fig. 10 (b) and (c), it does not requires CPU cores dedicated for data transfers between CPU and GPU.

These our approaches result in exploiting more computation power of not only GPUs but also CPUs without any complex implementations. Finally, our detailed task assignment for tasks 5, 6, and 7_7 has become like depicted in Fig. 10 (d).

5. Differences between Implementations

Figure 10 depicts differences in overlapping schemes among four implementations of the Linpack benchmark. In this figure, each white arrow represents a data flow between nodes, and each black arrow a data flow between CPU and GPU in a single node.

Figure 10 (a) is the case of the original HPL optimization called "look-ahead." With look-ahead, the *dgemm* calls are fragmented into small pieces to check incoming messages periodically. When a message comes, the process receives an entire message (during this time, neither CPUs nor GPUs perform any computation tasks) and then performs the remaining computation task at a time. However, fragmented *dgemm* calls to GPUs cause the performance degradation.

In the TSUBAME 1.2 and 2.0 implementations, Endo et al. avoid this disadvantage by creating a separate thread per process that makes *dgemm* calls for coarse grain submatrix portions (Fig. 10 (b) and (c) figure out examples when using them in our experimental environment). However, there exist idle times on CPU cores especially which are not responsible for communication. Moreover, TSUBAME 2.0 implementation does not use CPUs for *dgemm* due to the much smaller computation power of CPUs, which is 8% of the entire computation power, compared to that of GPUs.

In our experimental environment, CPUs contribute 35%, which is much greater than that in TSUBAME 2.0, and the inter-connects are relatively slow; therefore this types of idle time on the CPU cores are not negligible. So, our implementation depicted by Fig. 10 (d) carefully exploits these computation powers.

6. Results

An image of the GPU-accelerated PC cluster is shown in Fig. 11. In the figure, all the nodes denoted by rectangles are connected by two gigabit switches. The four small circles within the dark rectangle represent four cores of the processor. The hardware and software experimental environments are the same as in the case of the local parallel *dgemm* (Table 1 and Table 2).

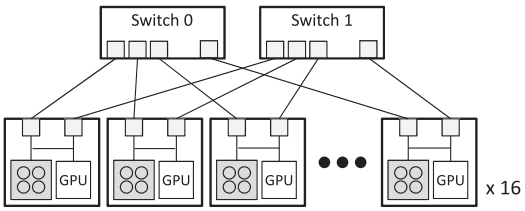


Fig. 11 An experimental GPU-accelerated PC cluster.

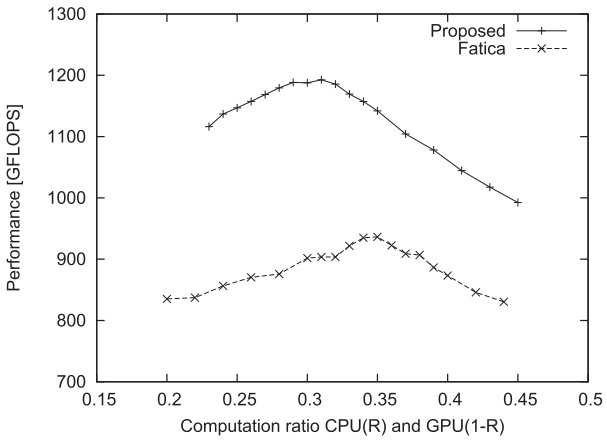


Fig. 12 Linpack performance with varying computation ratios *R* for the overlapping part.

Table 5 Optimal computation ratios for each task.

No.	Description	Ratio
3	*dtrsm*(L)	0.31 (see Sect. 4.4)
4	*dtrsm*(U)	0.31 (see Sect. 4.4)
7_1	*dgemm*(D_1)	0.35 (see Sect. 3)
7_3	*dgemm*(L_1)	0.35 (see Sect. 3)
7_4	*dgemm*(U_1)	0.35 (see Sect. 3)
7_7	*dgemm*(T_1)	0.31 (see Fig. 12)

6.1 Optimal Computation Ratios

Because part of the CPU time is assigned to tasks other than the *dgemm* operation in the overlapping part (see Fig. 10 (d), this part corresponds to the inter-node communication task and the CPU-to-GPU data transfer task in the OpenMP parallel region), we must reconsider the optimal computation ratio *R* for the overlapping part. We take different computation ratios, depending on the type of matrix–matrix multiplication and whether or not computation is overlapped with communications. We conducted experiments with varying computation ratios for the overlapping part and a fixed problem size $N = 90{,}000$. The experimental results are shown in Fig. 12, and the experimentally obtained optimal computation ratios are listed in Table 5.

6.2 Overall Performance

According to our experiments, the highest performance for our overlap GPU-accelerated Linpack benchmark can be

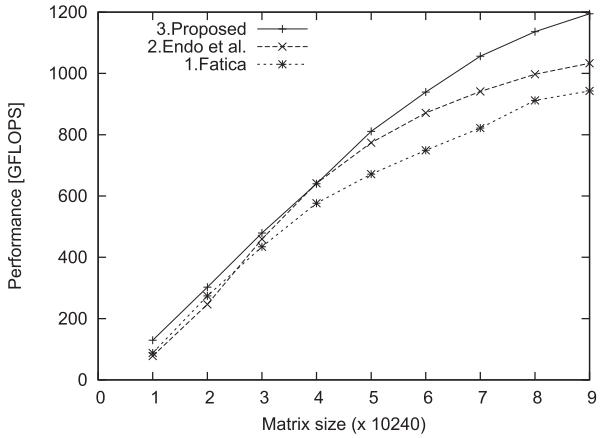


Fig. 13 Linpack performance with varying N.

achieved when the block size $NB = 576$.

The experimental results for different problem sizes are shown in Fig. 13. Our implementation (3 in Fig. 13) outperforms both implementations introduced by Fatica (1 in Fig. 13) and Endo et al. (2 in Fig. 13) at all problem sizes. The implementation of Endo et al. is one for the TSUBAME 2.0 but modified to use 3 CPU cores not involving a core for communication with GPU.

We perform about 69% of the computation on the GPU side. We use a 4×4 process grid, where each process has four threads; our implementation achieve 62% of the theoretical peak performance, while those of Fatica and Endo et al. are 48% and 54%, respectively. Our implementation outperforms that of Fatica by about 28% and that of Endo et al. about 16% thanks to the increase in the rate of the utilization of CPUs by an overlap between computation and inter-node communication and an overlap between computation and data transfer to the GPU device memory. However, the effect of the latter ovarlap is relatively small: 3% improvement in performance.

7. Discussion and Conclusion

We show that an implementation using the thread-level overlapping scheme can exploit the higher rate of the utilization of the entire computation power. However, when using the GPUs, it is also required to use efficient task breakdown and assignment to avoid performance degradation. We introduced our breakdown and assignment method and showed its efficiency.

We believe the proposed parallelizing scheme is also effective for the other data-parallel applications. Following is a procedure to apply the proposed scheme;

1. Select blocks for parallelization which should include both communication and computation and occupy a noticeable percentage of the execution time.
2. For each block, build a task-dependency graph which should also include the dependencies concerning the previous and next iteration.
3. Enlarge the available overlapping part by trying one or more techniques mentioned in [9], such as splitting tasks into small ones and so on.
4. Rebuild the task-dependency graph with the modifications caused by above steps, and build a new task-schedule using the hybrid MPI-OpenMP thread-level overlapping.
5. Divide computation tasks into CPU and GPU parts according to the capabilities of them. To avoid degradation of GPU calls and complex implementations, make sure that each overlapping part has a only one large GPU task instead of ones divided, and the CPU-to-GPU data transfer added to the thread-level overlapping.
6. Due to the communication tasks performed on a CPU core, available CPU computation power may be smaller than the theoretical values. Make an adjustment of the computation ratio of GPUs experimentally.

Our future work includes applying the proposed scheme for other applications based on this procedure.

Acknowledgements

This research is supported in part by a JSPS Grant-in-Aid for Scientific Research (C) No.22500042. We would like to thank Fatica and Dr. Endo since they allow us to use their Linpack source codes for our experiments.

References

[1] Q. Wang, J. Ohmura, S. Axida, T. Miyoshi, H. Irie, and T. Yoshinaga, "Parallel matrix–matrix multiplication based on HPL with a GPU-accelerated PC cluster," Proc. 2nd International Workshop on Parallel and Distributed Algorithms and Applications, PDAA'10, pp.243–248, 2010.

[2] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, "Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment," Proc. 7th International Conference on High Performance Computing for Computational Science, VECPAR'06, pp.305–318, Berlin, Heidelberg, Springer-Verlag, 2007.

[3] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," Concurrency and Computation: Practice & Experience, Euro-Par 2009, vol.23, no.2, pp.187–198, Feb. 2011.

[4] MAGMA, http://http://icl.cs.utk.edu/magma/, accessed March 11, 2011.

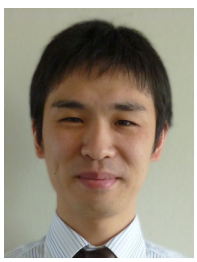
[5] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, pp.46–51, New York, NY, USA, ACM, 2009.

[6] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack tuning method on a heterogeneous supercomputer with hybrid accelerators," Proc. Summer United Workshops on Parallel, Distributed and Cooperative Processing, SWoPP2009, vol.2009-HPC-121, no.24, pp.1–8, Sendai, Aug. 2009.

[7] T. Endo, A. Nukada, S. Matsuoka, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," Proc. IEEE International Parallel & Distributed Processing Symposium, IPDPS 2010, pp.1–8, Atlanta, April 2010.

[8] T. Endo, A. Nukada, and S. Matsuoka, "Performance Evaluation of TSUBAME 2.0 Heterogeneous Supercomputer with Linpack Benchmark," Proc. 18th Hokkaido "High Performance Computing and Architecture Evaluation" Workshop, HOKKE 2010, Sapporo,

vol.2010-ARC-192/HPC-128, no.11, pp.1-6, Dec. 2010.
[9] T.Q. Viet, T. Yoshinaga, B.A. Abderazek, and M. Sowa, "Construction of hybrid MPI-OpenMP solutions for SMP clusters," IPSJ Transactions on Advanced Computing Systems, vol.46, no.3, pp.25–37, Jan. 2005.
[10] T.Q. Viet and T. Yoshinaga, "Improving linpack performance on SMP clusters with asynchronous MPI programming," IPSJ Trans. Advanced Computing Systems, vol.47, no.12, pp.340–348, Sept. 2006.
[11] J.J. Dongarra, J. Du Croz, S. Hammarling, and I.S. Duff, "A set of level 3 basic linear algebra subprograms," ACM Trans. Math. Softw., vol.16, pp.1–17, March 1990.
[12] ATLAS, http://math-atlas.sourceforge.net/, accessed Nov. 2010.
[13] Intel Math Kernel Library, http://software.intel.com/en-us/articles/intel-mkl/, accessed Nov. 27, 2010.
[14] AMD Core Math Library (ACML), http://www.amd.com/acml/, accessed Nov. 29, 2010.
[15] Texas Advanced Computing Center: GotoBLAS2, http://www.tacc.utexas.edu/tacc-projects/gotoblas2/, accessed Nov. 27, 2010.
[16] NVIDIA Corporation, NVIDIA CUDA Programming Guide Version 2.3.1, Aug. 26, 2009.
[17] A. Petitet, R.C. Whaley, J. Dongarra, and A. Cleary, "HPL - A portable implementation of the high-performance linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl/, accessed Nov. 27, 2010.



Junichi Ohmura is currently a master's student of the Graduate School of Information Systems, University of Electro-Communications (UEC). He is interested in high performance computing, cluster computing, and heterogeneous CPU/GPU computing. He is a student member of IPSJ.

Takefumi Miyoshi received his B.E., M.E., and D.E. from Tokyo Institute of Technology in 2003, 2005, and 2007, respectively. From Oct. 2007 to Nov. 2007, he was a JSPS post-doctoral researcher. From Dec. 2007 to Mar. 2009, he was an assistant professor in Graduate School of Creative Informatics, the University of Tokyo. From Apr. 2009 to Mar. 2010, he was post-doctoral researcher in Graduate School of Information Science and Engineering, Tokyo Institute of Technology. Since Apr. 2010, he has been with the Graduate School of Information Systems, UEC, where he is an assistant professor. His research interests are compiler techniques, many-core processor architecture, and co-design of hardware and software. He is also a member of ACM, IEEE, and IPSJ.

Hidetsugu Irie is an associate professor of network computing at Graduate School of Information Systems, the University of Electro-Communications. His research interests include microarchitectures, and network systems. He has a PhD in Information Science and Technology from the University of Tokyo. He was a researcher of Japan Science and Technology Agency from 2004 to 2008, and was an assistant professor of the University of Tokyo from 2008 to 2010. He is a member of IEICE, IPSJ, ACM, IEEE.

Tsutomu Yoshinaga received his B.E., M.E., and D.E. degrees from Utsunomiya University in 1986, 1988, and 1997, respectively. From 1988 to July 2000, he was a research associate of Faculty of Engineering, Utsunomiya University. He was also a visiting researcher at Electro-Technical Laboratory from 1997 to 1998. Since August 2000, he has been with the Graduate School of Information Systems, UEC, where he is now a professor. His research interests include computer architecture, interconnection networks, and network computing. He is a member of IEEE and IPSJ.