Data driven scheduling approach for the multi-node multi-GPU Cholesky decomposition

Yuki Tsujita, Toshio Endo

Tokyo Institute of Technology

Abstract. Recently large scale scientific computation on heterogeneous supercomputers equipped with accelerators is receiving attraction. However, traditional static job execution methods and memory management methods are insufficient in order to harness heterogeneous computing resources including memory efficiently, since they introduce larger data movement costs and lower resource usage. This paper takes the Cholesky decomposition computation, which is an important linear algebra kernel, as the target for optimization. And we describe a scalable data-driven scheduling method and a heterogenous memory management method in order to improve resource utilization and reduce amount of data movement. Through the performance evaluation on TSUBAME2.5, which is a heterogenous supercomputer with NVIDIA GPUs, we demonstrate the efficiency of the proposed task scheduling method and data replacement strategies considering data reusability.

1 Introduction

Recently general purpose graphic processing unit (GPGPU) computing, technology that harnesses GPUs for generic computation including scientific computing, is gathering attraction in high performance computing area, for GPUs' high computation throughput and memory bandwidth. In the latest Top500 supercomputers ranking[1], the Titan supercomputer ranked as world No. 2 is equipped with 18,688 GPUs to accelerate its performance and improve the power performance ratio. Also the TSUBAME2.5 supercomputer[2] at Tokyo Institute of Technology embodies 4,224 NVIDIA K20X GPUs.

GPGPU has been used for applications from various areas, including numerical optimization applications. In this paper, we take SDPARA software [12], a high performance solver for semi-definite programs (SDP) as the target for optimization. SDPARA's important computation kernel is the Cholesky decomposition for a dense large matrix, which already harnesses multiple GPUs in the recent version. It has achieved peta-scale computation speed of 1.7PFlops by using 4,080 GPUs on TSUBAME2.5 [10, 11]. For this application, it is required to support larger scale problems, which produces the larger matrix to be decomposed. In order to support larger matrix than the aggregate capacity of device memory among GPUs, we put the matrix on host memory, which has larger capacity. On the other hand, a typical, synchronous implementation suffers from larger amount of data movement between GPUs and CPUs. Although this issue is partially mitigated by parameter tuning such as block sizes[9], we will require further optimization toward future supercomputer architectures on which data movement will be more expensive relatively.

In this paper, we introduce data driven scheduling approach for the optimization of the multi-node multi-GPU Cholesky decomposition. Unlike the synchronous approach, the algorithm is expressed as a task dependency graph, where a single task corresponds to an update kernel of a small block of the matrix, which takes approximately 1 to 10 milliseconds. The matrix is distributed among the multiple nodes, and the task graph includes dependencies between tasks on different nodes. Our distributed fine-grained task scheduling method has the following properties:

- The scheduling method is scalable in order to support more than O(1M) tasks.
- The scheduling method is aware of memory hierarchy that consists of GPU device memory, local host memory and remote host memory. It is designed to minimize the data movement between the hierarchy.
- Our implementation supports overlapping of computation and data movement to improve the overall performance.

Through the performance evaluation on TSUBAME2.5 supercomputer, we demonstrate that the amount of data movement between CPUs and GPUs are reduced largerly, and we have achieved 13.9 TFlops on 16 nodes.

2 Background

2.1 GPGPU and PCIe Communication

GPGPU(General Purpose Graphics Processing Unit) is a technique to use computing resources of GPU (Graphics Processing Unit) for a general-purpose calculation as well as image processing. GPUs are processors originally designed for image processing, and mainly equipped by video cards and connected to the host computer via the PCI Express (PCIe) bus. Current GPUs can not work by themselves but works under the control of the host CPUs. Compared with CPUs, GPUs are designed to make throughput of computation higher; thus, they have been successful in parallel computations with regular structures, including matrix operations. Programmers can use GPGPU with dedicated programming tools, such as CUDA, OpenCL and OpenACC. In this paper, we use CUDA programming environment designed for NVIDIA GPUs, however, the proposed techniques are applicable to other environments.

While GPUs have higher computation throughput and memory bandwidth, they have limitations on memory size. The memory region that is directly accessible from GPU cores is called device memory, which is attached on the graphic card. Currently the device memory size is limited to several gigabytes (6GB on NVIDIA K20X GPUs, used in our evaluation), while the host memory can be expanded more easily (54GB on the TSUBAME nodes).

Therefore in order to support larger scale computation, we can harness the capacity of host memory in addition to device memory. However, we should consider the amount of data movement between CPUs and GPUs (hereafter we call it PCIe communication). Since the bandwidth of PCIe, 8GB/s in our case, is much smaller than device memory bandwidth (250GB/s on K20X), we have to reduce the amount of PCIe communication in order to achieve high performance. We take the Cholesky decomposition as the target computation, and introduce task scheduling methods that are aware of memory access locality, in order to reduce PCIe communication cost.

2.2 Cholesky Decomposition

The Cholesky decomposition takes a symmetric positive definite matrix A, whose size is $N \times N$. We assume A is a dense matrix. Then it decomposes A into the product of a lower triangle matrix L and its transposition, where $A = LL^T$.

Here we describe a typical parallel algorithm in the ScaLAPACK parallel linear algebra library[7]. The matrix A is divided into blocks with a uniform size $n_b \times n_b$, and the blocks are distributed among processes in two-dimensional block cyclic method. The algorithm consists of an outermost loop; at the k-th iteration of the loop, the sub matrix $A^{(k)}$ of size $n \times n$, where $n = N - k \times n_b$, is transformed into $L^{(k)}$ in place as follows.

$$\begin{aligned}\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix} \\ &= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}\end{aligned}$$

Here A_{11} is a single block of the $n_b \times n_b$ size, A_{21} is a $(n - n_b) \times n_b$ matrix and A_{22} is a $(n - n_b) \times (n - n_b)$ matrix as shown in Figure1. In a single iteration, we calculate the Cholesky decomposition of A_{11}, L_{11} first. Then the rest part of $L^{(k)}$ is obtained as follows.

$$\begin{aligned} L_{21} &\longleftarrow A_{21}(L_{11}^t)^{-1} \\ \tilde{A_{22}} &\longleftarrow A_{22} - L_{21}L_{21}^t = L_{22}L_{22}^t \end{aligned}$$

The ScaLapack routine executes this decomposition as follows.

1. PDPOTF2 : The process which has A_{11} performs the Cholesky decomposition.

$$A_{11} \longrightarrow L_{11}L_{11}^t$$

2. PDTRSM : L_{11} is send to all the processes which have A_{21} and they calculate L_{21}.

$$L_{21} \longleftarrow A_{21}(L_{11}^t)^{-1}$$

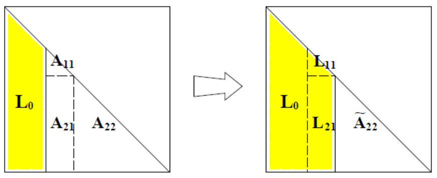

Fig. 1. The snapshot of the block Cholesky decomposition from figure 5 in J. Choi et al. [7]

3. PDSYRK : L_{21} is sent to all the processes and transposed. Then each process has L_{21} and L_{21}^t. They update a part of A_{22} using them.

$$\tilde{A_{22}} \longleftarrow A_{22} - L_{21} L_{21}^t$$

2.3 Simple Implementation on GPUs and Its Problem

The existing SDPARA GPU version uses a simple extension of this ScaLapack algorithm [11]. In order to support matrix larger than the aggregated capacity of device memory of GPUs, the matrix A is distributed among the compute nodes and allocated on host memory. In order to accelerate computation, each process executes the following work at each kernel routine. We copy the partial matrix, which is divided so that it fits in the device memory, from host to device via PCIe. And then we let the GPU compute for it by using high performance BLAS routines, and the copy back the result into the host memory. The computation on GPUs and PCIe communication are overlapped with each other.

This GPU implementation has the following problems:

Lower utilization of computing resources: The current ScaLapack based implementation is based on a synchronous style. For example, while a single process is calculating L_{11}, other processes tend to be idle, which degrades the total performance. We could improve the GPU utilization by introducing asynchronous execution, while we need to keep necessary data dependency.

More PCIe communication: The current implementation assumes that all the matrix data is available on host memory after each kernel finishes. With this method, we suffer from the cost of PCIe communication; the amount is $O(N^3/n_b)$. We could reduce it if the matrix data can reside in device memory over several iterations of the outer loop.

Larger memory consumption: ScaLapack uses the rectangular matrix data format, although only a triangular part is necessary for Cholesky decomposition. Thus memory consumption on host memory is twice than that is really required. We could support larger matrices by changing the data format.

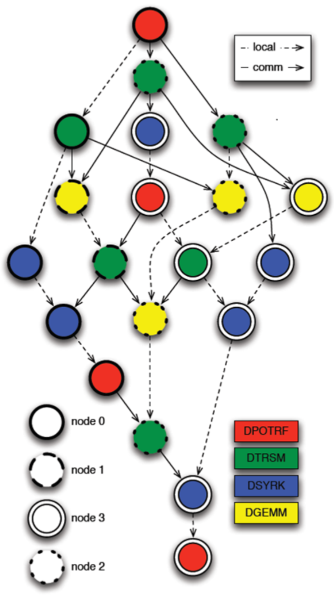

Fig. 2. Direct Acyclic Graph(DAG) of the Cholesky decomposition from figure 2 in G. Bosilca et al. [6]

2.4 Motivation for Data Driven Execution

The above discussion motivates us to adopt the data driven implementation similar to DAGuE [5] or StarPU [3]. Here we describe the basic execution method. First, we change the data format for the matrix. Instead of the rectangular format, we let each process maintain several blocks, each of which is an array of $n_b \times n_b$ size. Thus we can reduce memory consumption.

Next, when we consider the data dependency in the block level, we could harness more parallelism than in the synchnorous style. Therefore we consider the computation of a single block as a task, and we consider the dependency among the tasks. Also we introduce task scheduling methods of these fine grained tasks, while conforming the dependency, as described later. If a task execution requires input blocks that resides on remote processes, MPI communication is involved. Also we have to consider the memory hierarchy of device memory and host memory. If the input block is not available on the device memory, PCIe communication is involved. After all the input blocks are available on the device memory, we can execute the task. If the device memory is already full, some blocks are swapped out to the host memory. Also our scheduler is designed so that computation, MPI communication and PCIe communication are overlapped with each other.

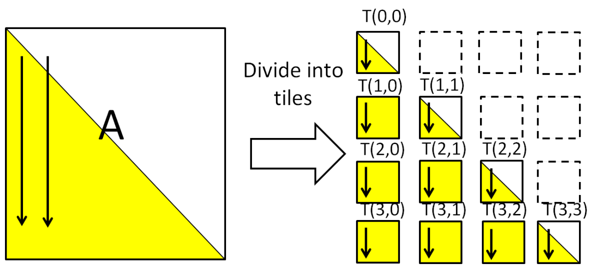

Fig. 3. Tile division

By using this method, the utilization of computing resources is expected to be improved, and the PCIe communication amount is reduced.

3 Our Scheduling Method

This section explains the implementation of the block Cholesky decomposition. After describing the basic data driven scheduling method, we discuss strategies for selecting runnable tasks and for GPU memory sweeping.

First, we divide the input matrix data A into the units called "tiles", each of which has $n_b \times n_b$ size. The tiles are distributed among MPI processes (we do not distinguish MPI processes and computing nodes in this discussion) in a two-dimensinal block cyclic style. Instead of holding all the tiles included in A, we hold only tiles for the lower triangular part of A as shown in figure3, because Cholesky decomposition assumes A as a symmetric matrix. In the initial state, the tile data is put on the host process.

We regard each computation kernel for a tile as a task, which is executed by the owner process of the target tile (owner computing rule). Each task can be executed if all the precedent tasks in the task graph has been finished. Unlike the synchronous execution method, tiles may be updated step by step independently. Thus each tile maintains a variable to express its current running step.

Also each tile is in one of the following three states:

RUNNABLE: The next task for this tile is runnable, since all the precedent tasks are finished.
SLEEP: In order to proceed the next task for this tile, we have to wait for precedent tasks.
FINISHED: All tasks for this tile have been finished. No more update is required.

In our implementation, an MPI process consists of several (two or three typically) worker threads and a ignition thread. We introduce multiple calculation threads in order to achieve overlapping of calculation, PCIe communication and MPI communication in a simple implementation. Each process has its task queue, shared by all its threads, in order to manage the runnable tasks on the process.

Each worker thread performs the following steps, task select, localize, execute, and finalize, continuously.

Task select It takes out a runnable task from the task queue if exists; we let T be the target tile of the task. If the task queue is empty, the caluculation is blocked.

Localize Generally, execution of a task requires the result data of the precedent tasks as inputs. We let T_{i1}, T_{i2} be the result tiles of the precedent tasks [1]. Then the worker thread checks the state of tiles T, T_{i1}, T_{i2} and executes the corresponding operations as follows.

1. if the tile data is on device memory, nothing is required.
2. if the tile data is not on device memory, but on the local host memory, the tile data is copied to device memory via PCIe bus. This may involve *swapping out* operation, as described below.
3. if the tile data is neither on device memory nor on local host memory, the thread issues `MPI_Recv` in order to receive the tile data from its owner process. After the data arrival, we execute as in Case 2.

Execute Now all the requred tile data are available on GPU; thus we execute the calculation task, which is typically invocation of a BLAS function on GPU.

Finalize When a task is finished, the worker thread performs operations for the following tasks, which need the result of this task. Thie operations involve inter-process messages of two types as shown in Figure 4. First, the calculation thread sends *notice messages* to processes that have following tasks, which may eventually make the following tasks runnable. In the current implementation, we send the data of tile T to the receiver immediately. To avoid blocking the worker thread long, we use non-blocking communication, `MPI_Isend`, for sending notice messages and tile data.

The ignition thread continuously checks arrivals of notice messages that notify information of finishing tasks. If the ignition thread finds the notice message makes a local task runnable, it adds the task into the process's task queue [2].

With this described method, we can execute the whole computation in a data driven style. This method reuse data of tile on the GPU device memory if possible; this can reduce PCIe communication between CPUs and GPUs.

3.1 Memory Management

In our implementation, each process put data of all the tiles owned by the process on the host memory. On the other hand, the smaller GPU device memory is used like a "cache" of the host memory.

[1] In Cholesky decomposition, each task depends on two tasks or less.

[2] Note that the input tile data is received by a work thread, not by the ignition thread

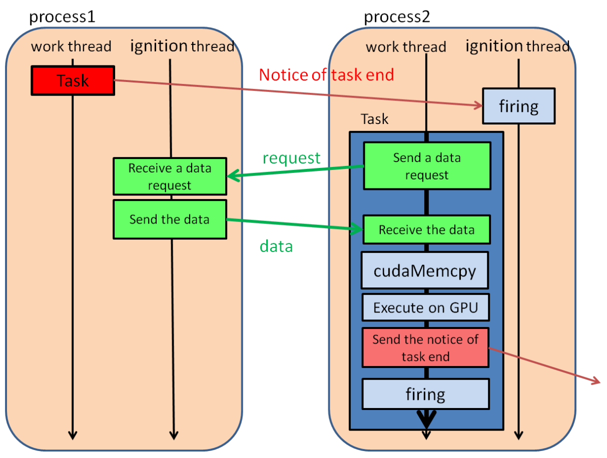

Fig. 4. MPI Communication pattern when a task is finalized

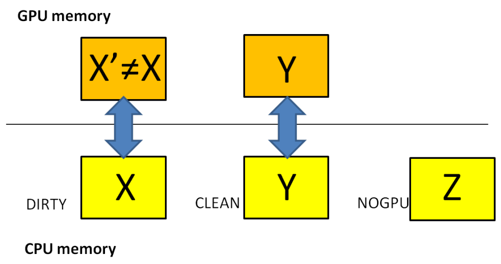

Fig. 5. The association state of CPU memory and GPU memory

When a process copies a tile data to GPU, it needs to evict other tile if the capacity is full. In this time, the data on GPUs has to be copied back if it is newer than data on host memory (the data is dirty). In order to maintain consistency, each tile has an additional variable expressing the state of the cached data on GPU as shown in Figure5. Each tile is in one of the following three states.

DIRTY: The tile has a copy on GPU memory, which may be different from data on host memory.

CLEAN: The tile has a copy on GPU memory, and consistent with host memory.

NOGPU: There is no copy on GPU memory.

When the DIRTY tile is swapped out from the GPU, the process copies back the tile to host memory, copies the data of the new tile from host memory to device memory. When the CLEAN tile is swapped out, copying back can be omitted.

When the process replaces the data of the device memory, it chooses a tile to be swapped out as the victim. The strategy for selecting the victim can affect the performance of following processes. In the evaluation of this paper, we use LRU strategy. In LRU strategy, we select a tile that has not been used for a long time. To implement this, we maintain a list of tiles and move tiles that are used for task execution into the tail of the list.

3.2 Task Selection Strategies

As previously described, we manage the runnable tasks by using the task queue per process. Since a task queue may contain several runnable tasks, we need to make strategies to select a task to be executed. We compare the following four strategies.

FIFO strategy:
We take the oldest task from the queue.

Random strategy:
We take one of runnable tasks randomly.

Greed strategy
We prefer a task that can be executed with less data movement. When a worker thread is going to take a task, we traverse tasks in the task queue to see how much data movement will be required in the "Localized" step described above. If we find a task whose required tiles data are already available on the device memory, we take the task for execution immediately. This strategy is expected to improve the access locality and reduce total PCIe communication costs.

ByIJ strategy
This strategy is also expected to improve the access locality, and takes the property of the Cholesky decomposition into account. In this computation, the tiles that reside in the same row or the same column tend to have strong relations with each other. In this strategy, we track a last task that has been finished. When the task of $T(i, j)$, a tile in the i-th row and j-th column, finishes, the next task is chosen as follows.

1. If the queue has a task of the tile in the same column $T(\cdot, j)$, it is taken out. If not, we proceed the next step.
2. If the queue has a task of the tile in the same row $T(i, \cdot)$, it is taken out. If not, we proceed the next step.
3. A task of the top of the queue is taken out.

4 Performance Evaluation

To evaluate the performance of our implementation (called "NEW") with data driven scheduling, we have conducted the performance measurement. The NEW

Table 1. Hardware specification of TSUBAME 2.5 node

CPU	Intel Xeon X5670 2.93 GHz (6 cores) x 2
CPU Memory	54GiB
GPU	NVIDIA Tesla K20X × 3
GPU Peak Performance	1.31TFlops per GPU
GPU Memory	6GiB per GPU

implementation includes several strategies for task selection in memory swapping as described in the previous section. Our implementation is compared with a synchronous implementation used in the existing SDPARA version[10, 11], which is called "OLD" in this section. OLD has recorded 1.7PFlops on the TSUBAME 2.5 supercomputer with 4080 GPU.

4.1 Experimental Condition

For the experiment we have used the TSUBAME 2.5 supercomputer at the Global Scientific Information and Computing Center at Tokyo Institute of Technology. TSUBAME 2.5 is a GPU-accelerated supercomputer, and its total peak performance reaches 5.7PFlops. Table1 represents the hardware specification of the node used in the evaluation.

We have used a GPU per node and conducted all the performance evaluation with four work threads. We have fixed the tile size at 1024×1024. For GPU management, we used NVIDIA CUDA 6.0, and used CUBLAS 6.0 as basic linear algebra library on each GPU.

4.2 PCIe Communication Amount

First we have measured the amount of data movement between CPUs and GPUs. We have performed it with the OLD implementation and several versions of NEW implementation. For NEW implementation, we measured with each task selection strategy. The measurement is done with varying matrix sizes; with the smallest case, the matrix data can be fully allocated on (aggregated) GPU memory, and the other matrix sizes are larger than the GPU memory capacity. The results are shown in Figure 6 (one node cases) and Figure 7 (four nodes cases). The PCIe communication amounts are normalized to one with OLD implementation.

The figures exhibits the communication amount between CPU and GPU greatly decreased in NEW implementations, for all the combinations of strategies. If the matrices are small, we have reduced it to about 17% on one node and 15% on four nodes. Here the strategies do not affect the performance, since data reuse is fully successful if the matrix data fits the GPU memory. PCIe communication is limited to the beginning and the end of the Cholesky decomposition.

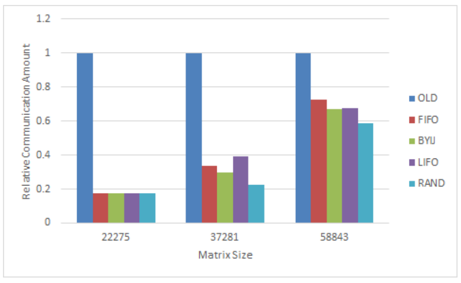

Fig. 6. Relative communication amount of PCIe on one node

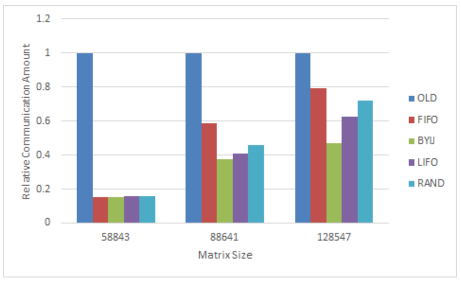

Fig. 7. Relative communication amount of PCIe on four nodes

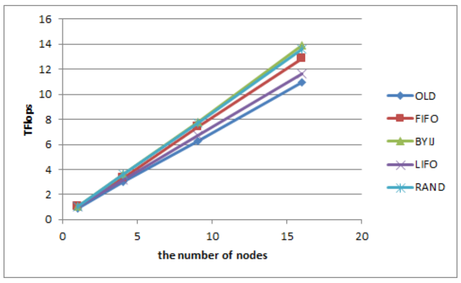

Fig. 8. Weak Scalability

On the other hand, OLD invokes PCIe communication on every kernel routine that causes redundant communication.

With larger matrices than GPU memory, we also observe the reduction of PCIe communication, though the reduction is mitigated as the matrix gets larger. On four nodes, NEW with BYIJ strategy reduces it to 37% with matrix size of 88,641. Among the task selection strategies, the BYIJ strategy greatly reduces the communication amount by using the reusability of data, as expected.

4.3 Weak Scalability

Figure 8 represents the weak scalability study. This matrix size is scaled up accordingly to the number of nodes to keep the data size per a node constant. The minimum matrix size is 58,843 with one node and the maximum one is 247,131 with 16 nodes. We observe that all the implementations exhibits good scalability. Among them, BYIJ achieves the best performance, 13.92TFlops with 16 nodes (16 GPUs). We observe the every NEW versions shows better performance than OLD version. We consider the reason is as follow; the reduction of the PCIe communication amount improved performance as planned. But, with LIFO strategy, we can not get much performance for PCIe communication amount. We will further investigate this point in future.

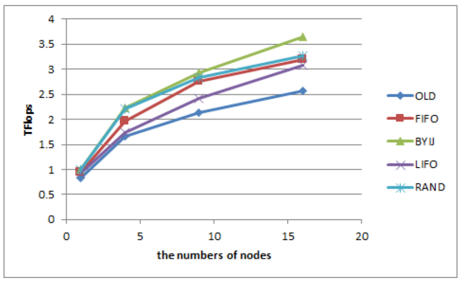

Fig. 9. Strong Scalability

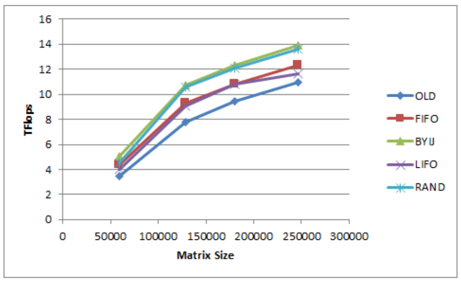

Fig. 10. Performance with varying matrix sizes on 16 nodes

4.4 Strong Scalability

Figure 9 represents the strong scalability study. The matrix size is fixed at 47142, and the number of nodes varies from one to 16. Compared with weak scalability case, the scalability is milder, especially with FIFO and RAND strategy. But BYIJ and LIFO strategy are a little better scalability than other strategies. As the number of nodes increases, the NEW strategies give much better performance than the OLD strategy. This seems due to the followings; increase of the number of the nodes cause the matrix data assigned to the node to be smaller than the GPU memory capacity.

4.5 Varying Matrix Sizes

We have conducted the performance evaluation with varying matrix sizes from 58,843 to 247,131. The number of nodes is fixed at 16, and the results are shown in Figure 10. When we compare the various strategies, we see similar results to Figure 8; the ByIJ strategies show the best performance and the RAND version comes next. The BYIJ strategy achieves 13.92TFlops with the 247131×247131 matrix.

5 Related Work

Our data driven scheduling method is strongly influenced by DAGuE/PaRSEC by Bosilca et al. [5, 4]. They have presented a direct acyclic graph (DAG) scheduler for distributed environments with GPUs, and demonstrated that the scheduler can execute applications including the Cholesky decomposition efficiently. We also use their methodology of tiling algorithm. On the other hand, to our knowledge, it is not clear how DAGuE/PaRSEC treats memory objects when GPU memory is full. Our focus is to reduce the amount of data movement between host and GPUs, by introducing task selection strategy that is aware of data locality and memory swapping strategy. In future, we are planning to compare our methods and DAGuE/PaRSEC in detail. Also we could embed our strategies in their implementation.

StarPU[3] is a DAG scheduling framework for heterogeneous environments. It allows for each task to run either on CPUs or GPUs according to the resource utilization, in order to improve the performance of execution of the whole task graph. It also maintains data consistency, while mitigating data movement between CPUs and GPUs. However, StarPU has been basically designed for a single node, while our target is distributed environments. Although the recent version is integrated to MPI communication, the programming model for distributed task dependency is different from local dependency.

In order to harness memory hierarchy of GPU memory and CPU memory in a transparent style, authors have proposed a runtime library called hybrid hierarchical runtime (HHRT)[8]. HHRT uses an oversubscription model; each GPU is shared by multiple processes, and when GPU memory is full, data of some processes are automatically swapped out. This methodology is successful for stencil based applications, however, we did not adopt it for the Cholecky decomposition. One of the reasons is that using MPI communication between processes on the same node degrades the overall performance for this computation. Also the memory consumption would be increased because of the lack of the mechanism for sharing memory objects among processes. After these problems are solved, we could integrate HHRT and the scheduling methods in this paper.

6 Conclusion and Future Work

We have described data driven scheduling approach for the optimization of the multi-node multi-GPU Cholesky decomposition. With our implementation, the communication amount between CPU and GPU is reduced by scheduling tasks appropriately and replacing the data considering its reusability. Compared with the synchronous implementation, the amount of PCIe communication is reduced by more than 80% with smaller matrices than GPU memory size, and for larger matrices, it is reduced by 40 to 60% with the best strategies. The implementation is scalable and achieved 13.9TFlops with 16 GPUs on 16 nodes.

Among the described strategies, the "BYIJ" task selection strategy shows the best performance both for communication reduction and the speed performance. It shows we can get better performance by adopting the property of the computation for scheduling.

Also we are going to measure the performance with O(1000) nodes of TSUBAME, in order to evaluate the scalability in peta-scale environments in order to accelerate the solution of large scale SDP problems with more than 2,000,000 constraints.

ACKNOWLEDGMENT

This research was supported by the Japan Science and Technology Agency (JST), the Core Research of Evolutionary Science and Technology(CREST) research project.

References

1. Top500. `http://www.top500.org/`.
2. Tsubame2.5. `http://tsubame.gsic.titech.ac.jp/`.
3. Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Concurrency and Computation: Practice and Experience*, pages 187–198, Februaly 23, 2011.
4. G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *IEEE Computing in Science and Engineering*, 15(6):36–45, 2013.
5. George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. In *Parallel Computing*, volume 38, pages 27–51,, 2012.
6. George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. Technical Report ICL-UT-10-01, Innovative Computing Laboratory, April 11, 2010.
7. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the scalapack lu, qr, and cholesky factorization routines. In *Technial Report UT CS-94-246, LAPACK Working Note 80*, September 1994.
8. Toshio Endo and Guanghao Jin. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. In *Proceedings of IEEE Cluster Computing (CLUSTER2014)*, pages 132–139, 2014.
9. Toshio Endo, Akira Nukada, Satoshi Matsuoka, and Naoya Maruyama. Linpack evaluation on a supercomputer with heterogeneous accelerators. In *Proceedings of IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2010)*, pages 1–8, 2010.
10. Katsuki Fujisawa, Toshio Endo, Hitoshi Sato, Makoto Yamashita, Satoshi Matsuoka, and Maho Nakata. High-performancd general solver for extremely largescale semidefinite programming problems. In *Proceedings of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC12)*, pages 1–11, 2012.

11. Katsuki Fujisawa, Toshio Endo, Yuichiro Yasui, Hitoshi Sato, Naoki Matsuzawa, Satoshi Matsuoka, and Hayato Waki. Peta-scale general solver for semidefinite programming problems with over two million constraints. In *In Proceedings of the International Conference on Parallel and Distributed Processing Symposium 2014 (IPDPS2014)*, page 10pages, 2014.
12. M. Yamashita, K. Fujisawa, and M. Kojima. Sdpara : Semidefinite programming algorithm parallel version. In *Parallel Computing*, volume 29, pages 1053–1067, 2003.