Scalable Multi-GPU 3-D FFT for TSUBAME 2.0 Supercomputer

Akira Nukada*†, Kento Sato* and Satoshi Matsuoka*‡†
*Tokyo Institute of Technology
†Japan Science and Technology Agency
‡National Institute of Informatics
Email: {nukada,kent}@matsulab.is.titech.ac.jp,matsu@is.titech.ac.jp

***Abstract*—For scalable 3-D FFT computation using multiple GPUs, efficient all-to-all communication between GPUs is the most important factor in good performance. Implementations with point-to-point MPI library functions and CUDA memory copy APIs typically exhibit very large overheads especially for small message sizes in all-to-all communications between many nodes. We propose several schemes to minimize the overheads, including employment of lower-level API of InfiniBand to effectively overlap intra- and inter-node communication, as well as auto-tuning strategies to control scheduling and determine rail assignments. As a result we achieve very good strong scalability as well as good performance, up to 4.8TFLOPS using 256 nodes of TSUBAME 2.0 Supercomputer (768 GPUs) in double precision.**

I. Introduction

Fast Fourier Transform (FFT) [1], [2] is one of the most important computational scheme as well as being commonly used as a powerful tool to reduce the amount of overall calculation by transforming operations into that in spectral space. FFT is used not only in many multimedia applications including signal processing, audio recognition, image processing, decoders/encoders, but also in large scale simulations in various fields such as weather/climate, molecular dynamics, and fusion. In particular, 3-D FFT is mainly used in high performance computing applications such as direct numerical simulations [3] and Protein docking simulations [4], [5], in which FFT becomes the dominant kernel. As a result, typically FFT is a key part of computational benchmarks; as such most of processor vendors, compiler vendors and OS vendors provide highly optimized FFT libraries for their products.

GPGPU [6], i.e. use of Graphics Processing Unit (GPU) for general-purpose computation, is now focused especially in high performance computing area due to its high floating-point performance, high memory bandwidth as well as high power efficiency. FFT was ported to GPUs very early on [7], [8]; initially FFT implementation was designed as a shader code executed in the graphics pipeline, therefore there were many limitations and overheads. Henceforth, the programmability of GPU was extended continuously so as to enable high quality graphics rendering using more complex shader code. The by-product of this was that general-purpose programming also became easier. Then NVIDIA introduced CUDA GPU architecture [9] and the CUDA language, both of which were optimized for GPGPU computations. By the use of CUDA, several implementations of high-performance FFT [10], [11], [12] were proposed, in addition to NVIDIA's CUFFT library. Also, there are several works on auto-tuning FFT implementations for CUDA GPUs [13], [14], achieving high performance across many kinds of GPU products and transform sizes.

CUDA applications can be extended into multiple GPUs using OpenMP, MPI, and other standard parallel programming framework. The use of multiple GPUs increases the performance as well as capacity of the device memory. Some CUDA applications are forced to use multiple GPUs due to the limitation of the capacity. However, a major issue arises here: we need to effectively manage the global data transfer between GPUs, and typically communication becomes dominant part of the multi-node FFT implementations.

In an earlier work we extended single GPU implementation of 3-D FFT into multiple GPUs using MPI+CUDA [15]. The implementation was fairly straightforward; however it was very difficult to achieve scalable performance, even for small number of GPUs. The primary reason for the poor scalability was that, when we increased the number of nodes, the message size in all-to-all communication became increasing smaller. Overall, overheads in data transfer operations were quite large in multi-node FFTs, especially with GPUs.

Today, numerous FFT libraries are available, but their interfaces of library functions are often similar. Typically, an FFT library would have two library functions; one function that performs initialization (or planning), followed by another which actually computes FFT. Some libraries also conduct auto-tuning during initialization. In general, an application repeatedly calls FFTs of same transform size. For this reason, many libraries provide separate initialization routines to minimize the overheads of FFT computation itself during the main iterations. We should minimize the overheads in the same way to improve the scalability of 3-D FFT when using multiple GPUs.

We propose to minimize the overhead by combination of several strategies. In order to minimize the overheads, we carefully optimize the scheduling of the data transfers using low-level IBverbs API and CUDA API, and effectively overlap intra- and inter-node communication in an aggressive fashion. Another problem is network congestion in large-scale InfiniBand network, and we propose to dynamically select from multiple rails automatically to avoid network congestions caused by busy traffic or faults in the InfiniBand itself. Finally,

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

we auto-tune the parameters to obtain the best results, both statically as well as dynamically. The result is dramatic—we achieve very good strong scalability as well as good performance, achieving up to 4.8TFLOPS performance using 256 nodes of TSUBAME 2.0 Supercomputer (768 GPUs), several times faster than reported in comparable work.

The rest of this paper is composed as follows. Section 2 summarizes the results of typical implementation of multi-GPU 3-D FFT using CUDA and MPI. In Section 3, we propose several optimizations to improve the scalability of all-to-all communication in multi-GPU 3-D FFT. In Section 4, we evaluated the proposed optimizations. Section 5 describes related works, and conclusions in Section 6.

II. Background

Many GPU environments nowadays are multi-GPUs. Workstations typically now accept multiple GPUs, and there are large-scale GPU cluster system with thousands of GPUs such as our Tokyo Tech.'s TSUBAME2.0. Across such environments, GPU applications can use algorithms that are tailored for theoretically unlimited number of GPUs, up to the point of their physically availability. Such multi-GPU environments can be fundamentally categorized as distributed memory parallel computer systems, whereupon numerous FFT implementations research had been conducted. [3], [16], [17], [18], [19], [20], [21].

3-D FFT computes 1-D FFTs for each dimension of 3-D array data. To compute the 1-D FFT efficiently, all the data along the dimension should be located on the same node. We need to exchange data between nodes during the computation of 3-D FFT on distributed memory systems, where data distribution depends on the application. In the case of a typical 1-D distribution, the computation of 3-D FFT can be done as follows, where we need only one all-to-all communication.

Step 0. Initial data as (NX, NY, NZ/P).
Step 1. Perform 2-D FFT for dimension X&Y.
Step 2. Shuffle data as (NX, NY/P, NZ).
Step 3. Perform 1-D FFT for dimension Z.

On the other hand, after those three steps (1∼3), data distribution is different from the initial one, but data distribution change during the transform is usually accommodated for in the applications themselves to obtain maximum performance. For example, several applications call inverse transform after forward transform and they have no limitation in the data distribution between the transforms, such as three-dimensional reference interaction side model (3D-RISM) [22], [23], [24] and many convolution-based applications including docking simulations [4], [5].

In most FFT implementations, it is most important to optimize the all-to-all communication between nodes, even sacrificing the data distribution uniformity as above, as the data transfer between the compute nodes are the bottleneck in parallel computations of FFT. However, matters become more complicated for coprocessor implementations such as with GPUs: In addition to the host CPU memory, GPU devices have local device memory which allows fast access from GPU. In CUDA, we use CUDA memory copy API for data transfer between host memory and GPU devices whereas MPI library for data transfer between nodes. As a result, data transfer between GPU devices requires three steps: CUDA device-to-host transfer, MPI transfer, and CUDA host-to-device transfer. Since multi-GPU FFT would require all-to-all communication between GPU devices, the simplest implementation would be to use MPI_Alltoall() routine in combination with CUDA memory copy operations as pre- and post-ambles. NVIDIA Fermi GPUs support efficient memory copy operation of block-stride patterns using CUDA 2-D memory copy API functions.

Such an implementation is simple, but not optimal because DMA controllers of GPU devices are idle during the all-to-all communication between nodes. Furthermore, CUDA memory copy operations of two directions, host-to-device and device-to-host, are not overlapped, and wastes bi-directional PCI-Express as well as inter-processor link bandwidth such as QPI or HyperTransport. In order to maximize bandwidth usage in the system, we propose to employ software-pipelining for the three steps to perform all the transfers simultaneously, but this would result in splitting the MPI all-to-all communication into many point-to-point communications, which would not be very efficient for small payloads.

Figure 1 shows the strong scalability of a simple implementation of N^3 3-D FFT on TSUBAME 2.0 Thin nodes using CUDA 4.1 and OpenMPI 1.4.2. Table I and Figure 2 show the specifications and the block diagram of the TSUBAME 2.0 Thin node respectively. On-board 1-D and 2-D FFT computations use corresponding functions of the NukadaFFT library [25], which enables 1-D FFT computations for dimension Z

TABLE I
Specifications of TSUBAME 2.0 Thin node (HP SL390G7, 1408 units)

Component	Model
CPU	Intel Xeon X5670 (6-core, 2.93GHz)×2
Memory	54GB, DDR3-1600 Triple-Channel×2 sockets
GPU	NVIDIA Tesla M2050 (3GB)×3
Network	Mellanox QDR x4 InfiniBand ConnectX-2 HCA×2
OS	SUSE Linux Enterprise Server 11 SP1
MPI	OpenMPI 1.4.2
CUDA	Tool Version 4.1, Driver Version 285.05.09
IBverbs	Version 1.1.4

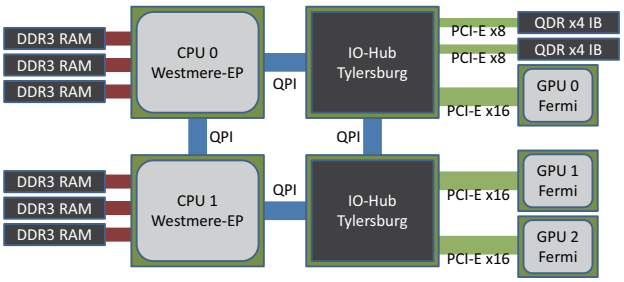

Fig. 2. Block diagram of TSUBAME 2.0 Thin node.

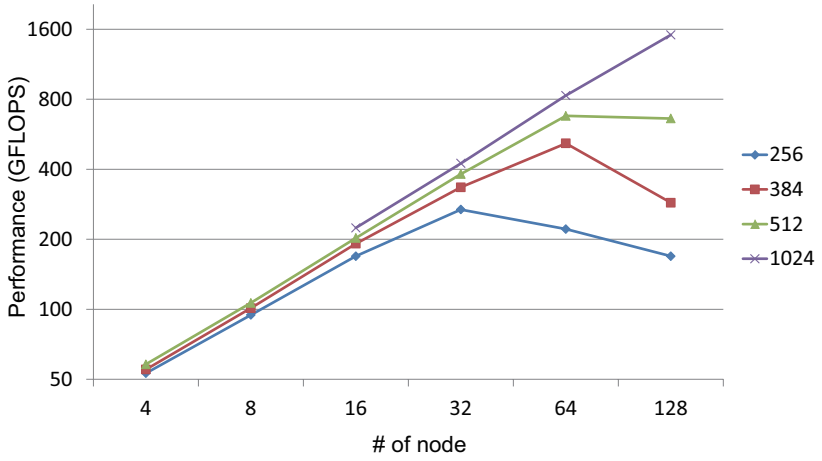

Fig. 1. Strong scalability of a simple N^3 3-D FFT implementation using MPI library and CUDA APIs straightforwardly. Only one GPU is used on each node to maximize bandwidth. CUDA memory copy operations and MPI data transfers are performed concurrently.

[1] without transpose operation [10], and optimizes for varying dimensional sizes using auto-tuning [13].

Since the aggregate bandwidth between compute nodes is theoretically proportional to the number of compute nodes in full fat-tree network topology as TSUBAME 2.0, the increase in the number of compute nodes should proportionally speed-up not only the on-board FFT part but also all-to-all communication between GPUs in FFT computation. For smaller numbers of nodes, performance is well scalable as shown in the graph. For larger numbers of nodes, however, the performance quickly saturates except for the $N = 1024$ case. That is because the message size in the all-to-all communication becomes too small to achieve high data transfer rate. When computing 256^3 3-D FFT using 128 nodes, the message size is only 16KBytes.

III. Improving Scalability of Multi-GPU 3-D FFT

We observed that the performance of 3-D FFT is not very scalable for large number of nodes. This is in contrast to on-board FFT computation where NukadaFFT exhibited strong scalability up to the limit of GPU memory. This is largely due to all-to-all communication not being scalable, because of the large overhead in transferring very small messages. Furthermore, all-to-all communication between large number of nodes easily suffers from network congestion, despite efforts by the modern MPI implementations to avoid them. For scalable all-to-all communication between GPUs, we need to solve these two major problems.

Our contributions described in this section are as follows:

- We employ low-level InfiniBand API to minimize the overhead of small message transfers, and to control the scheduling of the data transfers.
- The low-level API also allows selection of the InfiniBand rails for each data transfer. Using the multiple InfiniBand rails, we resolve the problem of congestion in InfiniBand networks.

Our contribution is not InfiniBand-specific, or limited to FFT alone. Rather, our technique is applicable to important class of algorithms such as FFT that require high performance all-to-all communications between GPUs. What we point out is that, by facilitating lower-level APIs than what is available with MPI, that (1) allow to overlap communications effectively along the communication path, and (2) facilitate path selection when multiple network routing paths are available, algorithms that require high-bandwidth could greatly benefit.

A. Reducing Overheads

Today, MPI is de facto standard programming model for distributed memory systems. Using the MPI library, we do not have to consider underlying hardware interconnect such as InfiniBand, Ethernet, or even proprietary ones such as those on Cray XT/XE, IBM BlueGene, Fujitsu K/FX-10, even if the nodes have multiple network rails or allow for multiple routing paths. As a sacrifice of such simplicity, however, we are not able to control the details of the behaviors of some MPI library functions, and would result in large overhead especially in the case of small message sizes.

As discussed above, achieving high-performance on bandwidth intensive algorithms such as FFTs require much finer and intricate control of the network, especially for GPUs. In this particular work, we utilize use IBverbs API for all-to-all communication instead of point-to-point MPI library routines. IBverbs is a portable, lower-level API for InfiniBand networks; to perform an RDMA transfer between nodes using InfiniBand, we must (1) establish connection between the nodes, (2) register the memory buffer used for RDMA, (3) exchange access keys for the buffer, (4) post request for RDMA transfer, and (5) wait for completion using IBverbs API functions. In practice, we need to execute the steps (1)$\sim$(3) only once if we maintain the established connections and re-use the same buffers and other resources, an optimization not explicitly

[1] nufftPlan0d() library function creates a plan which performs 1-D FFTs only for dimension Y of a 3-D array. We can use this for dimension Z by assuming ($N_x N_y$, N_z, 1) 3-D array.

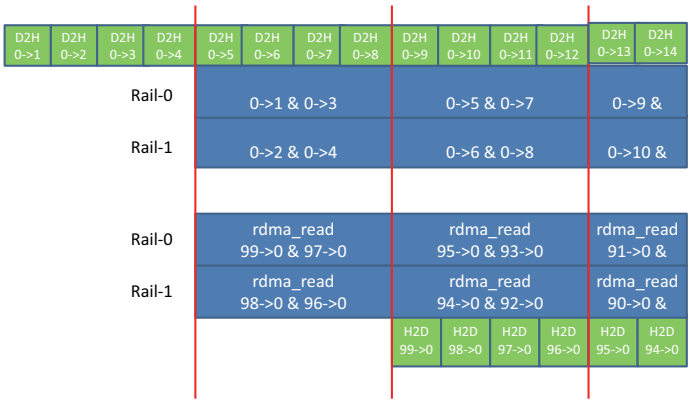

Fig. 3. An example of the data transfer scheduling using CUDA and IBverbs for dual-rail InfiniBand (Rank #0).

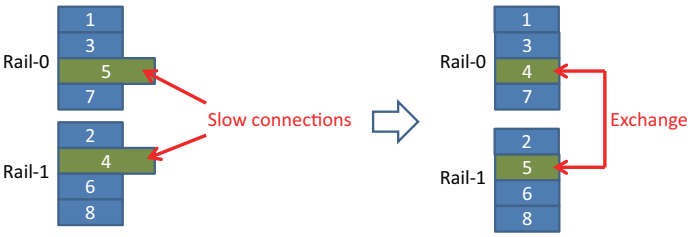

Fig. 4. Exchanging rails assigned for slow connections.

controllable with MPI. If the compute nodes have multiple InfiniBand HCAs or ports, the API allows us to manually choose the InfiniBand rails used for each peer node. This is useful in all-to-all communications to prevent the small message to each peer node from being split into smaller messages for multiple rails, which might cause performance degradation seen in some MPIs that implement all-to-all on top of its peer-to-peer API.

For our work, for portability we use the IBverbs APIs only for actual data transfers in all-to-all communications, where performance is critical. MPI library functions are used for other communication that are less performance critical, such as synchronization, reduction and gather operations, and our 3-D FFT routine works correctly when linked with existing MPI-based application codes.

Figure 3 shows an example of data transfer scheduling using CUDA and IBverbs for dual-rail InfiniBand network such as TSUBAME2.0. We use simple ring scheduling in the all-to-all communication. When a node with rank P receives data from rank $P-k$, the rank P also sends data to rank $P+k$ simultaneously, effectively pipelining data transfer. On every node, each direction of each InfiniBand rail communicates with a different node. Moreover, in each pipelined stage, multiple RDMA transfers are performed simultaneously, because this reduces the overheads of RDMA transfer in total. In the previous stage, the data to be sent to other nodes are transferred from device to host as shown in top of the Figure 3. In the next stage, the data received from other nodes are transferred from host to device as shown in bottom of the Figure 3, maximally utilizing the PCIe uplink and downlink simultaneously.

In the figure, the red vertical lines between stages indicate synchronizations between all nodes, ensuring that all the data transfers in the previous stage have completed, and of course data for the next stage is ready. This synchronization is also important to control scheduling. Assume that some nodes takes relatively longer time in previous stage. Without synchronization, earlier nodes start data transfers of the next stage and since those nodes will access the same peer nodes, this further slows down the slow nodes.

As an optimization parameter, we can choose the number of RDMA transfers executed simultaneously in each stage. Here, a large number reduces the overheads of RDMA transfers and number of synchronizations. On the other hand, the first and last stages take longer time which is not overlapped with RDMA transfers. Considering these kinds of trade-offs, we need to carefully choose the best parameter, and we resolve this through auto-tuning as we will discuss later.

B. Rail Selection

Many supercomputers employ InfiniBand network for interconnects of the compute nodes. To provide higher bandwidth, some including TSUBAME 2.0 or Kyoto-U/U-Tsukuba T2K have multiple InfiniBand HCAs or HCA with multiple ports. The usual tradeoff is that, multi-rail InfiniBand network increases complexities because we must load-balance between rails. On the other hand, multi-rail also gives us another optimization opportunity. If we observe a network contention between two nodes on an InfiniBand rail, we might be able avoid it by using the other rails. This is because each InfiniBand rail has independent routing tables. In the TSUBAME 2.0 system, all of the Thin nodes have two InfiniBand HCAs and those are connected to primary and secondary InfiniBand networks, respectively. Since several nodes such as Medium/Fat nodes with large memory capacity and storage systems are connected only to the primary InfiniBand network, those two networks are not identical. As a result, the routing between two nodes is not always the same in both rails.

Rail selection also allows us to avoid using bad/unstable InfiniBand links or ports by changing rails. In large-scale InfiniBand network with fat-tree topology, it is not easy to detect bad links, especially between edge switches and top-level core switches. Although bad links can be avoided by using performance counters, this is difficult to do in a production machine running numerous jobs by many users. Moreover, links can have slow-down problems without increasing error counts, and such issues occur only when the link is overloaded, making detection very difficult.

Figure 4 illustrates the strategy of exchanging rail assignments. As mentioned in the previous subsection, multiple RDMA transfers are performed simultaneously in our implementation. The number k ($k = 1, 2, \ldots, 8$ in Figure 4) indicates the ring transfer to k-th neighbor rank. The RDMA requests are posted almost at same time by all the nodes; however the exact time at which they complete are not all the

same. Here, we measure the elapsed time for each RDMA transfer, and find the slowest one for each rail on all the nodes. Those slowest ones become the candidates of the rail exchange, although the phenomenon might be one-time only or minor in that we should exchange the rail assignment only if the candidates continue to be extremely slow. Current implementation decides to conduct rail exchange only if the slowest one takes more than 1.1x of the fastest one of each rail. We note that such hysteresis threshold is very important: assume that 2.x slower rail is exchanged with another rail. Without the threshold, it may return to the original rail if it was selected as the slowest in the newly assigned rail, even if the slow-down ratio is only 1.01x. Although exact determination could have minor effect, in practice we have found threshold of 1.1 works well, and that is the number used in our evaluations in the next section.

In our implementation, multiple ring transfers are performed concurrently. Each ring transfer must be assigned to one of the InfiniBand network to avoid load imbalance. In this case, if network congestion is observed, the congestion occurs (1) between multiple ring transfers, or (2) within a single ring transfer. In both cases, we can move only the slowest ring transfer to the other rail.

The exchange of rail assignment is conducted during the last phase of each 3-D FFT computation. In this phase, we first gather the timing data on each node. Then, for each execution, we exchange the rail assignment at most once for each stage until it converges. One caveat is that, the solution may be local minima, but since the number of combination of the rail assignments is too large, it would not be practical to find the theoretically optimal combination, especially during runtime. Instead, we focus on eliminating critical congestions with this simple method, and experiences have shown that this step-by-step method convergences to the near-optimal fairly quickly.

C. Auto-tuning

In general, applications using FFT computations repeatedly perform transforms of same size. This makes it easy to use an online auto-tuning approach [13].

Of the two proposed optimizations for all-to-all communications, we need to determine the best parameters corresponding to those optimizations. The first parameter is the chunk size, and for this both offline and online tuning would be feasible. Since the number of candidates for this parameter is fairly small, our implementation uses online tuning based on the first several executions. The second parameter is the rail assignment. Since in a large production machine the overall condition of InfiniBand network will continue to fluctuate on continuous basis with execution of new jobs, fault occurring in various parts of the machine, etc., tuning of the rail assignment is performed constantly even if convergence is reached to cope with changes in the network.

IV. Performance Evaluation

We proposed several optimizations to improve the scalability of 3-D FFT using multiple nodes and multiple GPUs. In this section, we evaluate the effects of each using the TSUBAME 2.0 Thin compute nodes as shown in Table I. When calculating performance in GFLOPS, we assume N-point 1-D FFT requires $5N \log_2 N$ floating-point operations, known as pseudo GFLOPS in FFT. N^3 3-D FFT requires $15N^3 \log_2 N$ floating-point operations. Due to the lack of space, we compute complex-to-complex 3-D FFT in double precision in all of the performance evaluations as this is a typical application requirement, although effects in single precision and/or real would be similar.

A. NUMA support

The TSUBAME 2.0 Thin nodes have two CPU sockets, and each has an embedded memory controller and local memory. In 3-D FFT computation, most of the memory accesses are done by (multiple) DMA controllers of InfiniBand HCAs and GPU devices. In standard scenarios, DMA access can efficiently transfer the data between host memory and devices; however, a memory controller might become a bottleneck when several DMA controllers are accessing the same memory controller. This could seriously affect performance especially after minimizing the overheads using low-level APIs. For this reason, we allocate memory buffers on all CPU sockets beforehand to avoid such a situation.

Figure 5 shows the performance comparison with and without NUMA support. TSUBAME 2.0 Thin nodes have two InfiniBand HCAs and two CPU sockets. Although using memory controllers of both CPUs might seem advantageous, CUDA memory copy between CPU socket 1 and GPU 0 increase the data transfer of QPI bus, since both HCAs are attached to the PCIe lanes of the chipset on Socket 1. To determine the optimal configuration, we assigned separate buffers for each HCA rail and conducted comparison. Figure 5 compares two memory locations of the buffers: (1) both buffers being on CPU socket 0, and (2) each HCA buffer different CPU sockets. As shown in the graph the latter is advantageous by about 3–11%, confirming that avoiding bottlenecking memory controller with DMA engines being more significant. We note that the improvement ratio is somewhat less compared with simple InfiniBand throughput tests without GPUs or computation thereof, where the effect of individual buffers allows us to attain much greater performance differences (4.4GB/s $\rightarrow$ 6.1GB/s, without GPUs).

B. Chunk Size Selection

Figure 6 shows the performance of 256^3 3-D FFT with different chunk sizes. The number of nodes is 64, and three GPUs are used on each node. In this evaluation, the dynamic rail selection is disabled.

As mentioned in Section 3.1, small chunk size is not preferable because it requires significant synchronizations, and small numbers of RDMA transfers are performed simultaneously which leads to relatively large overhead. On the other hand, large chunk size decreases the percentage of overlapped stages. In the case of 64, there are no overlap between RDMA

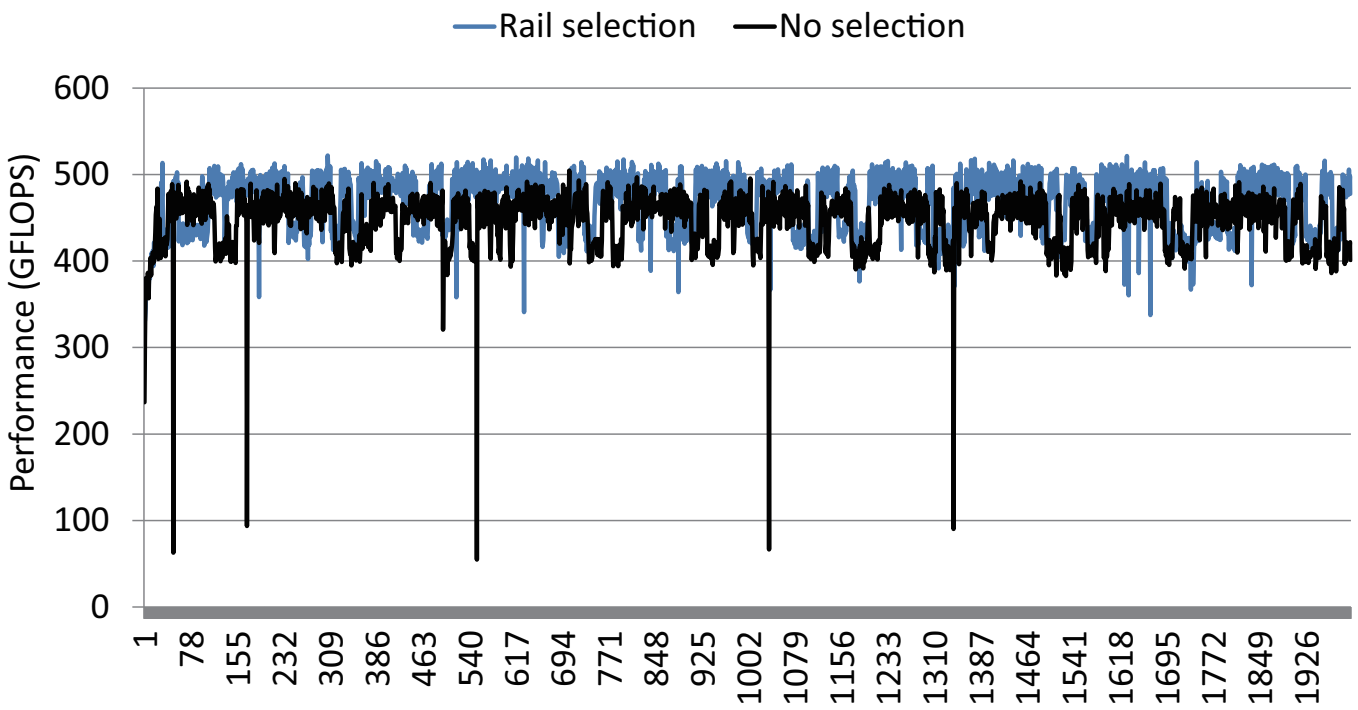

Fig. 7. Performance of 2,000 executions of 3-D FFT with and without the dynamic rail selection.

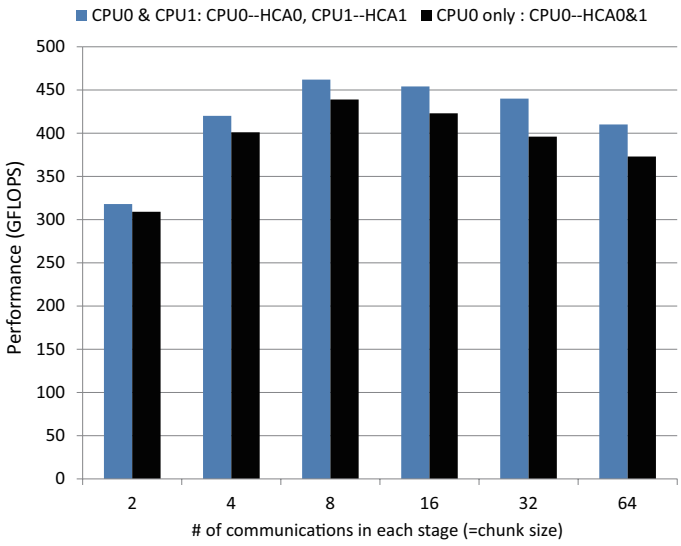

Fig. 5. Performance of 256^3 3-D FFT with and without NUMA support. The number of nodes is 64, and three GPUs are used on each node.

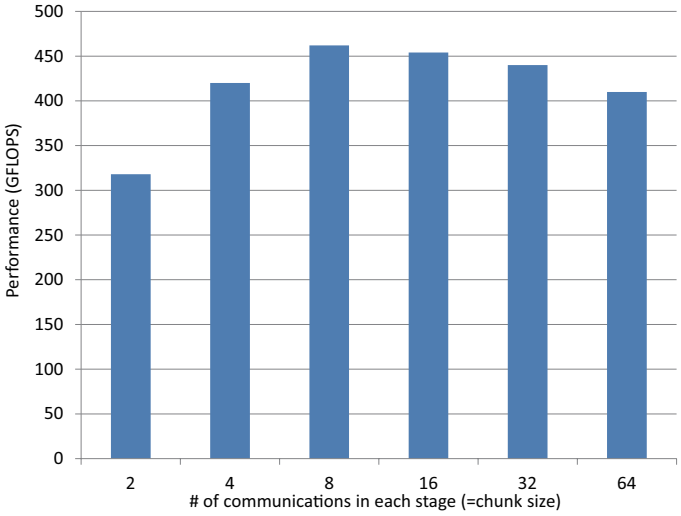

Fig. 6. Performance of 256^3 3-D FFT with different chunk sizes. The number of nodes is 64, and three GPUs are used on each node.

transfers, CUDA host-to-device transfers, and CUDA device-to-host transfers. Here, performance degradation is not as bad as one would expect, as RDMA transfers of both directions are still performed simultaneously even without the overlap.

Figure 8 shows the performance comparison between two implementations with IBverbs and with MPI_Isend&Irecv. In both cases, the chunk selection is enabled, and dynamic rail selection is disabled. The implementation with IBverbs achieve 1.1x–1.6x higher performance than MPI. The difference comes from the overheads of MPI library functions. We observed large performance difference in case of large number of nodes, that is, when the message size is small.

C. Dynamic Rail Selection

Figure 7 shows the performance of continuous 2,000 executions of 3-D FFTs, with and without the dynamic rail selection. Although we observe some performance degradations in both

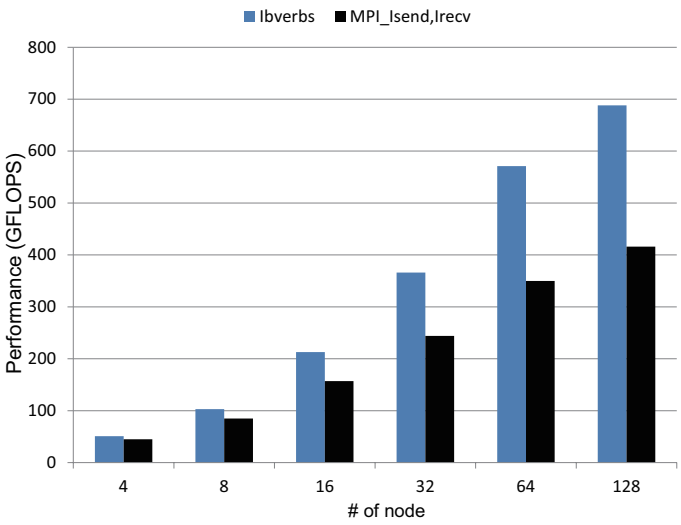

Fig. 8. The performance comparison between implementations with IBverbs and with MPI_Isend,Irecv. 256^3 3-D FFT is computed. Chunk size selection is enabled, and dyamic rail selection is disabled. Only one GPU is used on each node.

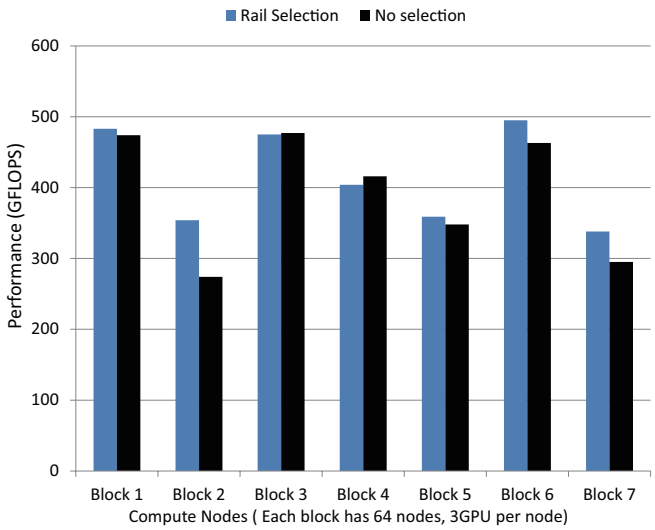

Fig. 9. Performance of 3-D FFT with and without the dynamic rail selection. Seven jobs are executed simultaneously on the system, and each job compute 256^3 3-D FFT repeatedly using 64 nodes and 192 GPUs.

cases, tremendous performance degradations are avoided by the use of dynamic rail selection.

Figure 9 shows result of another experiment. Assume a job executes 256^3 3-D FFTs repeatedly using 64 nodes and 192 GPUs. We launched seven such jobs simultaneously as a block in this experiment to simulate a real-life scenario when one would be performing parallel parameter sweep, across those seven jobs. Each block of nodes consists of 64 nodes, and the same set of nodes was used for each block, both with and without dynamic rail selection. In this case, $64 \times 7 = 448$ nodes perform heavyweight all-to-all data transfers, possibly causing considerable network congestion. Furthermore, since TSUBAME2.0 is a highly-utilized production supercomputer, the remaining 900 or so nodes were being used for jobs of other 50-100 or so users totally oblivious to the experiment but still affecting the network. Since many of the Tsubame2.0 jobs are MPI jobs or conduct heavy I/O, the InfiniBand network was considerably congested. Here, we see that although block 3 and 4 exhibits a small amount of performance degradation, we succeeded in performance improvement in most of the blocks, by up to 29%.

Needless to say, dynamic rail selection works well only if there are network contentions. Otherwise, for example when executing with 64 nodes and other nodes being practically idle, we might not have to exchange the rail assignments; in such a case allgather operation to collect the transfer speeds at each node simply becomes the overhead. One small improvement would be to turn on the rail selection when the system is busy, and turn off or reduce frequency of the adjustment otherwise. Another option is to merge the current all-to-all communication data and transfer speed information as being carried over to the next stage. This might be advantageous with small increase in the data size sent to each node, an issue we will experiment in our future work.

D. Scalability

Finally, we demonstrate scalability, especially focusing on strong scaling. Figure 10 shows the strong scalability results using one GPU on each node. The performance is considerably improved from the prior implementation using MPI interface shown in Figure 1. Our implementation still increases the performance even with 128 nodes, while the pure MPI implementation saturates with only 32 nodes. Table II shows the breakdown of the execution time spent in each phase. The on-board 1-D and 2-D computations are well scalable, although they finally saturate due to insufficient parallelism for GPUs. The all-to-all time also reduces as the number of nodes increases, as a result of minimizing the overheads in all-to-all communications, although still on a small increase trajectory. In the case of 4 nodes, all-to-all communication occupies 76.0% of total execution time, while it occupies 87.7% in case of 128 nodes.

Figure 11 shows the strong scalability results using three GPUs on each node. Only one process is launched on each node. Our implementation is hybrid, that is, the data from three GPUs are packed and then unpacked on the target node so that the size of the message transferred between nodes is same as the 1 GPU per node experiment. In most cases, the performance is about 1.5x higher than 1 GPU per node case whereas the on-board computation improvement is of course much greater. As such, if total performance is of premium importance, then one would still obtain performance gains with 3 GPUs. Here, the performance difference is attributed to the fact that the all-to-all communication part is faster, because the CUDA data transfer using six DMA controllers completes earlier, and for the remaining time InfiniBand HCAs are able to work more efficiently. On the other hand for very small transform sizes, the use of three GPU on each node decreases performance. Although message size transferred

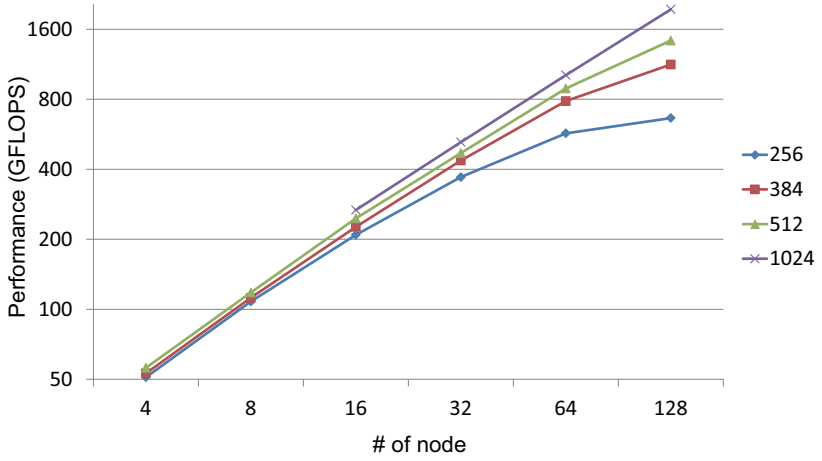

Fig. 10. Strong scalability of N^3 3-D FFT performance. Only one GPU is used on each node.

TABLE II
BREAKDOWN OF THE EXECUTION TIME OF COMPUTING 256^3 3-D FFT. ONLY ONE GPU IS USED ON EACH NODE. TIME IS IN MSEC.

# of nodes	4	8	16	32	64	128
2-D FFT(dim. X&Y)	4.780	2.378	1.229	0.650	0.333	0.182
All-to-all	29.961	13.864	7.168	4.137	2.853	2.657
1-D FFT(dim. Z)	4.658	2.368	1.196	0.650	0.338	0.19
% of all-to-all	76.0%	74.5%	74.7%	76.0%	80.9%	87.7%
Total	39.399	18.610	9.593	5.437	3.524	3.029

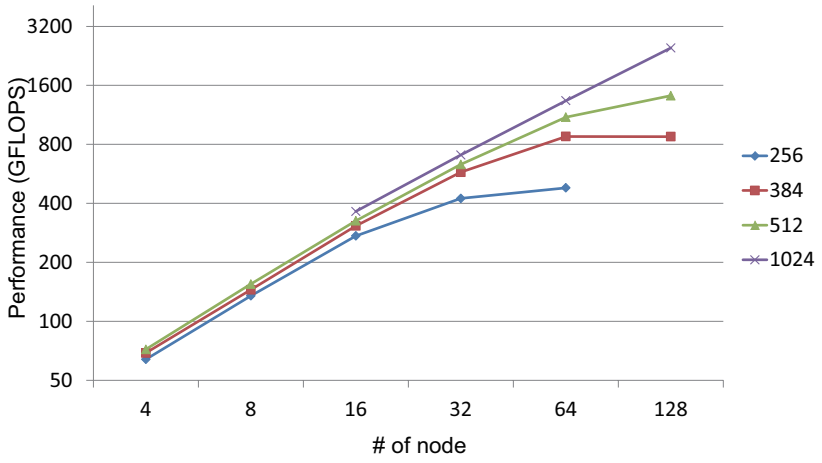

Fig. 11. Strong scalability of N^3 3-D FFT performance. All three GPUs are used on each node.

between nodes is same, the message size of CUDA memory copy becomes smaller, and the advantages of having DMA controllers becomes diminishes.

Although it was easy to achieve scalable performance using MPI for large transform sizes, we can still improve performance. In case of 1024^3 transform using 128 nodes, MPI implementation achieves only 1.5TFLOPS with one GPU per node, but our optimized implementation achieves 1.9TFLOPS with one GPU per node and 2.5TLOPS with three GPUs per node. Using 256 nodes and 786 GPUs, achieve more than 4.8TFLOPS in 2048^3 3-D FFT computation, several times faster than multi-GPU FFTs reported in the past, or CPU-only implementations with very similar network bandwidth and (full fat-tree) topology.

In-core performance of 3-D FFT using single GPU of 256^3, 384^3 and 512^3 are 68.0GFLOPS, 48.5GFLOPS and 50.7GFLOPS respectively. We could achieve up to 9.7x, 23.3x and 28.2x speed-ups using multiple GPUs. This is a very

favorable result for many applications, as this indicates that FFT-based solvers can obtain good speedups (i.e., strong scaling) even for smaller problem sizes, such as being the case for molecular dynamics.

V. Related Work

Chen et al. work on 3-D FFT using GPU cluster [26]. They also use low-level InfiniBand API to achieve higher bandwidth using dual-rail than MPI library. However, their target is very large-scale and using small GPU cluster consists of 16 nodes and one InfiniBand switch. We focus on not only large-scale but also small sizes which can be executed on single GPU, and we could achieve much higher performance in double precision than [26] in single precision.

Czechowski, et al. work on 3-D FFT using same compute nodes with similar GPUs but only one InfiniBand HCA [27]. Their achieved performance is less than 1/3 of ours for large size (1,024), and less than 1/6 for smallest size (256). Our implementation is much better even in consideration of the difference in number of InfiniBand rails. Their FFT algorithm is also different from ours. Their implementation uses local transposes which is one of the most time consuming part. On the other hand we are using NukadaFFT library routines that enable transpose-free multi-dimensional FFT proposed in [10].

Takahashi [20] shows the performance of 3-D FFT using T2K Tsukuba system. The compute nodes are CPU based, and have quad-rail DDR InfiniBand network. The theoretical peak bandwidth of InfiniBand network is 8GB/s, which is same as dual-rail QDR InfiniBand of TSUBAME 2.0 Thin node. He achieved up to 401GFLOPS using 256 nodes (4,096 cores). To achieve the same performance, we need only 32 nodes. Although the algorithm is different, we demonstrate that 3-D FFTs are not simply limited to network injection or bisection bandwidth, but could benefit from much faster many-core processors by allowing strong scaling with proper data transfer optimizations.

VI. Conclusion

Several multi-GPU applications require scalable implementation of 3-D FFT. Since typical implementations using MPI and CUDA have limitation in scalability due to the large overheads in data transfer for small messages. We have presented highly software-pipelined all-to-all communication between GPUs using low-level IBverbs API and CUDA API to minimize the overheads and directly manage the resources and rail assignment. In large scale InfiniBand network of supercomputers, network contention at some points may become the bottle-neck. Our implementation employs dynamic rail selection to avoid those contentions. As a result of these optimizations, we achieved much higher scalability for small transform size such as 256^3. Performance for large transform sizes also improved, and finally we achieved up to 4.8TFLOPS using 256 nodes, even if many other jobs are running on the other compute nodes of the system. Our results would likely be applicable to other many-core systems such as Intel MIC, or other interconnects such as proprietary ones with multiple routing strategies, calling for standardized lower-level APIs compared to the current MPI standard to cope with strong scaling for current as well as future Exascale systems.

Acknowledgment

This research is partially supported by Core Research of Evolutional Science and Technology (CREST) project "ULPHPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies" of Japan Science and Technology Agency (JST), MEXT Grant-in-Aid for Young Scientists (A) 22680002, Grant-in-Aid for Challenging Exploratory Research 23650012, and NVIDIA CUDA COE Program.

References

[1] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Comput.*, vol. Vol. 19, pp. 297–301, 1965.

[2] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. SIAM Press, Philadelphia, PA, 1992.

[3] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, and Y. Kaneda, "16.4-Tflops direct numerical simulation of turbulence by a Fourier spectral method on the Earth Simulator," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762808

[4] R. Chen, L. Li, and Z. Weng, "ZDOCK: an initial-stage protein-docking algorithm." *Proteins*, vol. 52, no. 1, pp. 80–87, 2003. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/12784371

[5] D. Kozakov, R. Brenke, S. R. Comeau, and S. Vajda, "PIPER: An FFT-based protein docking program with pairwise potentials," *Proteins: Structure, Function, and Bioinformatics*, vol. 65, no. 2, pp. 392–406, 2006.

[6] General-Purpose Computation Using Graphics Hardware, "http://www.gpgpu.org/."

[7] J. Spitzer, "Implementing a GPU-efficient FFT," in *SIGGRAPH Course on Interactive Geometric and Scientific Computations with Graphics Hardware*, 2003.

[8] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003*, 2003, pp. 112–119.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[10] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[11] V. Volkov and B. Kazian, "Fitting FFT onto the G80 architecture," 2008, http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf.

[12] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *the 2008 ACM/IEEE conference on supercomputing*, 2008.

[13] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 30:1–30:10. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654090

[14] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju, "Auto-tuning of fast Fourier transform on graphics processors," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 257–266. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941589

[15] A. Nukada, Y. Maruyama, and S. Matsuoka, "High performance 3-D FFT using multiple CUDA GPUs," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 57–63. [Online]. Available: http://doi.acm.org/10.1145/2159430.2159437

[16] C. Calvin, "Implementation of parallel FFT algorithms on distributed memory machines with a minimum overhead of communication," *Parallel Comput.*, vol. 22, no. 9, pp. 1255–1279, Nov. 1996. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(96)00039-7

[17] C. E. Cramer and J. A. Board, "The development and integration of a distributed 3D FFT for a cluster of workstations," in *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, ser. ALS'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 26–26. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268379.1268405

[18] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain, "Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: implementation and early performance measurements," *IBM J. Res. Dev.*, vol. 49, no. 2, pp. 457–464, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1147/rd.492.0457

[19] J. Doi and Y. Negishi, "Overlapping Methods of All-to-All Communication and FFT Algorithms for Torus-Connected Massively Parallel Supercomputers," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.38

[20] D. Takahashi, "An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors," in *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 606–614. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882864

[21] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 84–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1899016

[22] F. Hirata, *Molecular theory of solvation*. Dordrecht, The Netherlands, 2003.

[23] T. Imai, A. Kovalenko, F. Hirata, and A. Kidera, "A New Approach for Investigating the Molecular Recognition of Protein: Toward Structure-Based Drug Design Based on the 3D-RISM Theory," *J. Am. Chem. Soc.*, vol. 131, pp. 12 430–12 440, 2009.

[24] Y. Kiyota, N. Yoshida, and F. Hirata, "A New Approach for Investigating the Molecular Recognition of Protein: Toward Structure-Based Drug Design Based on the 3D-RISM Theory," *J. Comp. Theo. Chem.*, vol. 7, pp. 3803–3815, 2011.

[25] A. Nukada and S. Matsuoka, "NukadaFFT : An auto-tuning FFT library for CUDA GPUs," in *NVIDIA GPU Technology Conference 2010 (Research Summit Poster)*, 2010. [Online]. Available: http://www.nvidia.com/content/GTC/posters/2010/U04-NukadaFFT-An-Auto-Tuning-FFT-Library-for-CUDA-GPUs.pdf

[26] Y. Chen, X. Cui, and H. Mei, "Large-scale FFT on GPU clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810128

[27] K. Czechowski, C. McClanahan, C. Battaglino, K. Iyer, P.-K. Yeung, and R. Vuduc, "Prospects for scalable 3D FFTs on heterogeneous exascale systems," in *Proc. of 2011 ACM/IEEE Conf. Supercomputing (SC)*, Nov. 2011.