CHAPTER

# Large-Scale Fast Fourier Transform

# 39

**Yifeng Chen, Xiang Cui, Hong Mei**

Bandwidth-intensive tasks such as large-scale fast Fourier transfers (FFTs) without data locality are hard to accelerate on GPU clusters because the bottleneck often lies with the PCI bus or the communication network. Optimizing FFT for a single-GPU device will not improve the overall performance. This chapter shows how to achieve substantial speedups for these tasks. Three GPU-related factors contribute to better performance: first, the use of GPU devices improves the sustained memory bandwidth for processing large-size data; second, GPU device memory allows larger subtasks to be processed in whole and hence reduces repeated data transfers between memory and processors; and finally some costly main-memory operations such as matrix transposition can be significantly sped up by GPUs if necessary data adjustment is performed during data transfers. The technique of manipulating array dimensions during data transfer is the main technical contribution. These factors (as well as the improved communication library in our implementation) attribute to 24.3x speedup with respect to FFTW and 7x speedup with respect to Intel MKL for 4096 3-D single-precision FFT on a 16-node cluster with 32 GPUs. Around 5x speedups with respect to both standard libraries are achieved for double precision.

## 39.1 INTRODUCTION

A GPU cluster is a network-connected workstation cluster with one or more GPU devices on each node. Several such systems are now ranked among the top 20 fastest supercomputers (as of June 2010).

Computation-intensive tasks such as dense matrix multiplication and Linpack are often easy to accelerate by GPUs [1–3]. Memory-bandwidth-intensive tasks with a high degree of data locality such as finite-difference simulation are also suitable for GPU clusters [4]: the source data in 2-D or 3-D can be decomposed spatially and distributed over the device memories ("dmem") of GPUs, which only communicate with each other about state updates at the boundaries of the spatial decomposition so that the bandwidth between GPU devices (through PCI bus and Infiniband) will not form the performance bottleneck. Applications can benefit from the high bandwidth of dmem and achieve significant speedups. Memory-bandwidth-intensive tasks without data locality such as large-scale FFT [5], on the other hand, are much harder to accelerate.

**GPU Computing Gems**

Most existing codes [6–9] assume FFTs to have a "small scale" so that the entire user data can be held in one GPU's dmem. It fits an application scenario in which FFT is performed repeatedly on these data. Then the overhead to transfer the source data from and to the host memory (i.e., the main memory or simply hmem) is overwhelmed by the computation time.

In this chapter, we consider a "large-scale" FFT whose dataset is too large to fit in one GPU's dmem and requires multiple GPU nodes. The performance bottleneck then lies with either PCI bandwidth between hmem and dmem or the network's communication bandwidth between GPU nodes. Some applications, such as turbulence simulation, use FFT as large as 4096 3-D (total 512 GB complex-to-complex data in single precision). Compared with a single node, a cluster will provide multiplied memory bandwidth and memory capacity. FFT on a cluster requires transposition of the entire array over the network — an operation usually requiring communication of the entire dataset and slow hmem transpositions on each individual node [10]. It is not obvious how GPU with high-compute capability and on-card memory bandwidth can help accelerate such tasks.

In this chapter, we take on this challenge and design and implement an FFT code that achieves 24.3x speedup (using GPUs) with respect to FFTW (not using GPUs) and 7x speedup with respect to Intel MKL (not using GPUs) on a 16-node cluster with two GPU devices and two QDR infiniband adapters on each node. Around 5x speedups with respect to both standard libraries are achieved for double-precision FFTs.

Figure 39.1 illustrates the architecture of our test cluster with 16 nodes. Each node is connected with an NVIDIA Tesla C1060 GPU, a GTX 285 GPU, and two Mellanox ConnectX QDR Infiniband adapters with a combined peak duplex bandwidth of 80 Gbps. Motherboards are Supermicro X9DAH+F, each providing 18 DIMM slots, two IOH controllers, and multiple PCI-e 2.0-gen. Each node has one Xeon Quad-Core E5520 2.26G CPU and 36 GB 1066 MHz memory. The main characteristic of this cluster is its balanced bandwidth for PCI and the network.

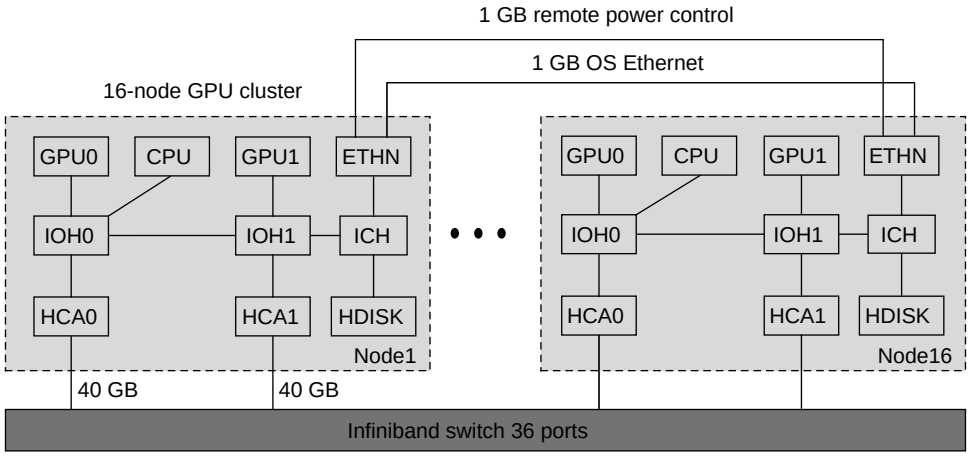


**FIGURE 39.1**

The architecture of PKU McClus cluster.

Several factors have contributed to the surprising performance of our FFT on PKU McClus: first, data processing by GPUs via PCI bus is faster than sustained large memory accesses (exceeding the cache size) by CPUs; second, dmem is much larger than CPU cache and hence allows larger sub-tasks to be processed in whole and reduces repeated data transfers between memory and processors; finally, array transposition on the GPU is much faster than that in hmem by CPUs. As a major technical contribution, we discover that by applying subtle data adjustments during data transfers (between dmem and hmem and between hmem and infiniband buffers), arrays no longer need hmem transposition. GPU transposition is up to 20 times faster than CPU transposition. For example, when copying a 512 MB array block from hmem to dmem, the algorithm partitions the data block and transfers it in 8 MB units. The source and destination addresses of these units are carefully permuted so that the effect of hmem transposition can be achieved by a combination of the permutation and some fast GPU transposition. This method of dimensional manipulation is also readily applicable to other algorithms of GPU clustering.

## 39.2 MEMORY HIERARCHY OF GPU CLUSTERS

The most notable characteristic of a GPU-accelerated workstation cluster is its exposed memory hierarchy. Different memory media have different sizes and bandwidth. There are two types of hmem: "pageable memory" and "pinned memory." Both can be very large (36 GB on PKU McClus), but relatively slow compared with other types of memory. The device memory of GPUs is smaller (e.g., 1 GB for GTX285 and 4 GB for C1060), but much faster. The bandwidth for data transfers between pinned hmem and dmem can be twice as fast as that between pageable hmem and dmem. L1 and L2 caches on CPU normally have several MBs and are faster than dmem. Memory components such as shared memory (or smem) and registers on GPU offer a combined throughput much higher than CPU caches. Instructions entirely operating on registers are known to outperform instructions with smem operands [3].

Figure 39.2 provides a convenient starting point for algorithm design. It illustrates the peak bandwidth of one-way singlex and two-way duplex data transfers on PKU McClus. Two-way singlex arrows are used if both directions of singlex data transfers have the same bandwidth. The bandwidth figures are all tested using real codes. In our tests, bandwidths of 73 GB/s and 122 GB/s are obtained on C1060 and GTX285, respectively, for properly coalesced dmem accesses. A common bottleneck for GPU acceleration lies with the PCI bus. The peak bandwidth from pinned hmem to dmem is 5.7 GB/s for one GPU device (either C1060 or GTX285). Bandwidth from dmem to pinned hmem can reach 5.4 GB/s. Bandwidth from or to pageable hmem is lower at 3.3 GB/s or 2.7 GB/s. When one is transferring data from pinned hmem to two GPUs simultaneously, the bandwidth can reach a combined 9.4 GB/s, while the opposite direction reaches 6.2 GB/s. The PCI of a GT200-based GPU works only in singlex mode. Thus, by overlapping computation and data transfers in streamed data processing, two GPUs can average 9.4*6.2/(9.4+6.2) = 3.7 GB/s. Note that the memcpy bandwidth within hmem is tested on a single-CPU thread. Multiple threads can reach about twice the bandwidth on our cluster.

Singlex infiniband communication through one Quad Data Rate (QDR) host channel adapter (HCA) on each node delivers 3 GB/s bandwidth using OpenMPI, while duplex communication only reaches 2.2 GB/s for either direction. OpenMPI is intended to be a transparent layer that handles the infiniband topology automatically. Ideally, the bandwidth using two HCA cards should be doubled, but our tests

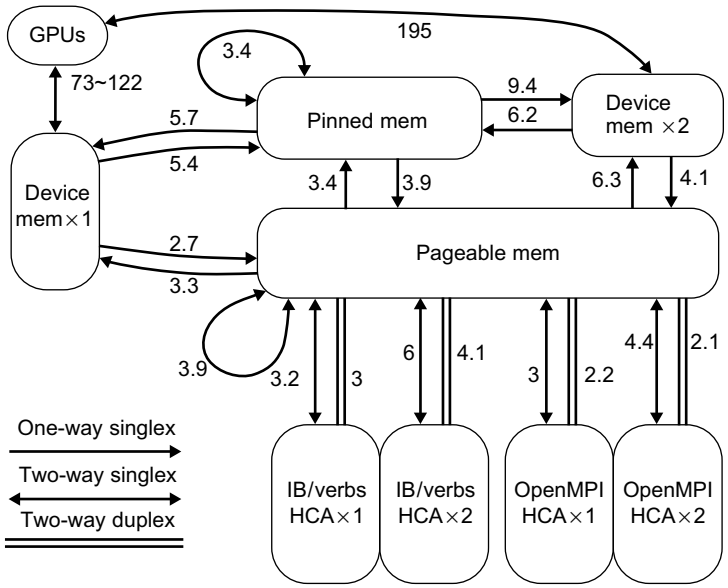


**FIGURE 39.2**

Bandwidth relations in GB/s between different memory media.

show that it only reaches 2.1 GB/s for either direction (under both OpenMPI 1.2.8 and OSU Mvapich2 1.2) in a so-called trunking, or link aggregation, mode. To maximize the bandwidth, we have developed an interface directly invoking IB/verbs's RDMA communication, achieving duplex 4.1 GB/s (for either direction), and singlex communication is improved to 6 GB/s as well. All tests are performed on direct HCA connections without switching. On our cluster, PCIs (with duplex 3.7 GB/s in combination) now reach a similar bandwidth as the infiniband's 4.1 GB/s. This is ideal for bandwidth-intensive tasks without data locality so that neither the PCI nor the network forms an obvious bottleneck.

Hardware and low-level software often require data to be transferred in certain restricted patterns. Failure to observe these patterns can yield very poor performances. For example, the GPU must access dmem in coalesced patterns. Noncoalesced dmem accesses can be exceedingly slow. In some algorithms it is advantageous to perform certain data adjustment during data transfers, which inevitably incur overheads. Our experiment shows that to reach near-peak bandwidth, data should be transferred in no-less-than 2 MB units between hmem and dmem. This provides a lower limit to the granularity of data transfers. Similarly, network bandwidth is also affected by message lengths (though much finer granularity is tolerated). Tests also show that duplex bandwidth about 4 GB/s seems to be as much as we can squeeze out of a typical 5500-series motherboard's PCI bus with 1066 MHz memory. That means when PCI bus and infiniband HCAs operate in parallel, their combined bandwidth will be capped. Streaming operations can overlap computation and communication, but the room for improvement is limited.

The software environment may also impose platform-related restrictions that further complicate the design and implementation of algorithms. Our FFT solution assumes CUDA 2.3 or higher because greater than 4 GB contiguous pinned hmem is needed. Another software restriction requires pinned memory not to be directly used as infiniband buffer. Our FFT solution works under this restriction.

## 39.3 LARGE-SCALE FAST FOURIER TRANSFORM

The process of data partition and distribution for traditional clusters is flat: the source data are distributed over all processes. GPU clusters, however, require data to be further decomposed according to the memory hierarchy.

### 39.3.1 A Naive Algorithm

To illustrate the design of our algorithm, we consider the largest FFT that the cluster can accommodate in the main memory. The source array is of three-dimensional $4096^3$ complex-to-complex in single precision with total 512 GB data (32 GB on each of the 16 nodes), which is close to the total hmem size 576 GB. Note that storing data in GPU dmem will not improve performance because any communication with other GPUs will inevitably go through hmem and the network. The bottleneck still lies with PCI or the network.

Let us first consider a naive algorithm illustrated in Figure 39.3(a). At first glance, what we need to do is to decompose the 3-D array along one of the dimensions so that each node holds a $4096\times4096\times256$ subarray of complex numbers. In order to use both GPUs (which have separate pinned hmem), we further halve the subarray into two $4096\times4096\times128$ subarrays of 16 GB each. In the first phase of 3-D-FFT, each GPU performs 128 2-D (for the the first two dimensions) FFTs, which are grouped into a stream of 32 batches, each as $4096\times4096\times4$ of 512 MB (fitting in device memory). As the computation time is overwhelmed by data transfers in the streaming operation, we simply use CUFFT. The results from the first phase are swapped with other nodes via AllToAll infiniband communications. Data are copied to the outgoing infiniband buffer before communication and from the

| | |
|---|---|
| for loop | for loop |
| pinned2d – 2D CUFFT – d2pinned | pinned2d – 2D CUFFT – d2IBbuf |
| memcpy pinned to IB buffer | AllToAll |
| AllToAll | ‖ memcpy IB buffer to pinned (with adjustment) |
| memcpy IB buffer to pinned | for All |
| **CPU hmem transposition** | pinned2d (with adjustment) |
| for loop | **GPU dmem transposition** |
| pinned2d – 1D CUFFT – d2pinned | 1D FFT kernel |
| **CPU hmem transposition** | **GPU dmem transposition** |
| for loop | d2pinned (with adjustment) |
| memcpy pinned to IB buffer | for loop |
| AllToAll | memcpy pinned to IB buffer (with adjustment) |
| memcpy IB buffer to pinned | ‖ AllToAll |
| | ‖ memcpy IB buffer to pinned |
| (a) Naive FFT algorithm | (b) PKUFFT algorithm |

**FIGURE 39.3**

3D-FFT algorithms.

incoming buffer after communication. The purpose is to merge the results from different nodes so that the third dimension becomes continuous on each node, and then we can apply CUFFT again to perform batched 1-D-FFT for the third dimension. Note that AllToAll communications alone cannot completely transpose the first two dimensions with the third dimension. We need the CPU's local transposition in every individual node's hmem. For 32 GB of data, this is an expensive operation. After transposition, the data can then be grouped into $4096 \times 4 \times 4096$ subarrays so that we can use GPUs to compute their FFTs for the third dimension in a streaming operation. Another round of CPU-hmem transposition is needed before restoring the original element positions for the result array.

### 39.3.2 Optimized Algorithm

Our more optimized algorithm is illustrated in Figure 39.3(b). Several minor aspects of the naive algorithm are improved. Dmem data after the 2-D FFTs can be directly downloaded to the outgoing IB buffer. That saves one round of copying from hmem to the outgoing buffer. The data from the incoming IB buffer are transferred to the appropriate destinations in the hmem. The three steps of batched 2-D FFTs, AllToAll communications and transfers from buffer to hmem can be pipelined. Similarly after 1-D-FFT for the third dimension, transferring the batch to the outgoing buffer, AllToAll communications and transferring from the incoming buffer can be pipelined and parallelized. Our experiments show that parallelizing the PCI transfers with infiniband communication does not offer any performance gain, but parallelizing transfers from/to buffers with infiniband communication does. Experiments also show little performance advantage to perform FFTs on CPU and GPU at the same time owing to the capped overall bandwidth.

Notice that the naive algorithm performs two CPU transpositions in hmem. These operations make sure that data of the third dimension are aggregated in addresses close to each other. On modern clusters, the network bandwidth is in the same range as the hmem bandwidth. Hmem transpositions thus become comparably expensive.

Can better results be achieved, given that GPUs can perform very fast dmem transpositions owing to the high dmem bandwidth? It turns out that this is possible if certain data adjustments are performed during data transfers between hmem and dmem and between hmem and infiniband buffers. The idea is that when, for example, transferring 512 MB data from hmem to dmem just before the third dimension's FFT, we partition the data and transfer them in 8 MB units. The source and destination offsets are carefully permuted so that the data are moved to the right places convenient for the following operations. Such data adjustment, as well as various array partitioning for GPUs and infiniband buffers, requires sophisticated manipulation of array dimensions, which will be the topic of the next section. Note that GPU transpositions are still needed for data manipulation at smaller scales because data transfers between different memory media require a certain level of granularity. Thus, the transposition of an entire array over the network requires a combination of data adjustment and GPU transpositions.

Figure 39.4 shows the threads on a single node and the data flow. Two GPUs deliver a combined PCI bandwidth. As any pinned hmem is owned by the thread that allocates it, two separate areas of pinned hmem and two infiniband communication buffers are allocated and associated to the GPUs. Eight CPU threads run on each node: two for controlling the GPUs, two for infiniband communications, and another four for managing the infiniband buffers and moving data between pageable buffers and pinned memory.

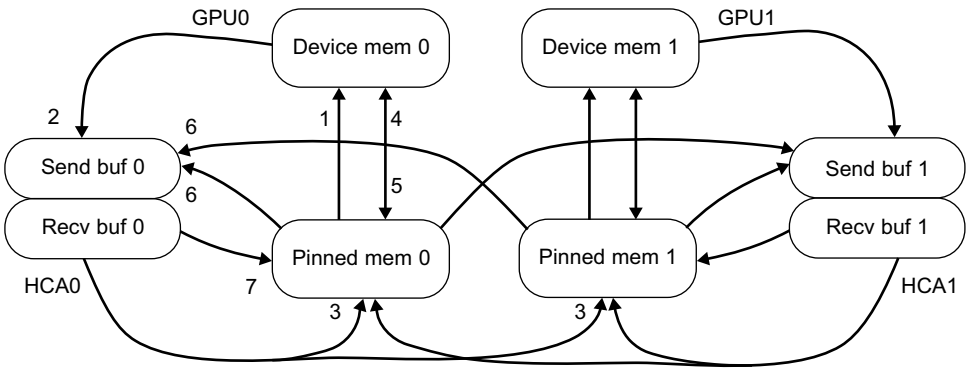


**FIGURE 39.4**

Data flow of 3-D-FFT on a single node.

## 39.4 ALGEBRAIC MANIPULATION OF ARRAY DIMENSIONS

We now study an algebraic method that can help algorithm designers determine the layouts of the array dimensions in intermediate steps of an algorithm. We first introduce the notation for simple FFT operations on traditional clusters (c.f. [10]).

### 39.4.1 Basics

An $8 \times 8$ array is shown in Figure 39.5(a) and represented as $\frac{y}{[8]}\frac{x}{[8]}$. The dimension of rows is named as $x$ and that of columns as $y$. The array is assumed to be row major (which means the dimension $x$ is stored continuously in memory). After a local transposition, it becomes like Figure 39.5(b) and represented as $\frac{x}{[8]}\frac{y}{[8]}$ in column-major order. In these expressions, each number in brackets under a line describes the size of the dimension, and the symbol above the line denotes the dimension's name. The example array can be partitioned along dimension $y$ and distributed over four processes, as shown in Figure 39.5(c) and represented as $\frac{y_1}{[4]}\frac{y_0}{[2]}\frac{x}{[8]}$. Partition along dimension $x$ for the transposed array is illustrated in Figure 39.5(d) and represented as $\frac{x_1}{[4]}\frac{x_0}{[2]}\frac{y}{[8]}$. It is also possible to partition the original array along both $x$ and $y$ dimensions. The result is a $4 \times 4$ array of $2 \times 2$ blocks: $\frac{y_1}{[4]}\frac{x_1}{[4]}\frac{y_0}{[2]}\frac{x_0}{[2]}$.

Three-dimensional FFT for traditional clusters can now be represented algebraically. Figure 39.6(b) shows the original 3-D array

$$\frac{x_1}{[4]}\frac{x_0}{[2]}\frac{y}{[8]}\frac{z}{[8]}$$

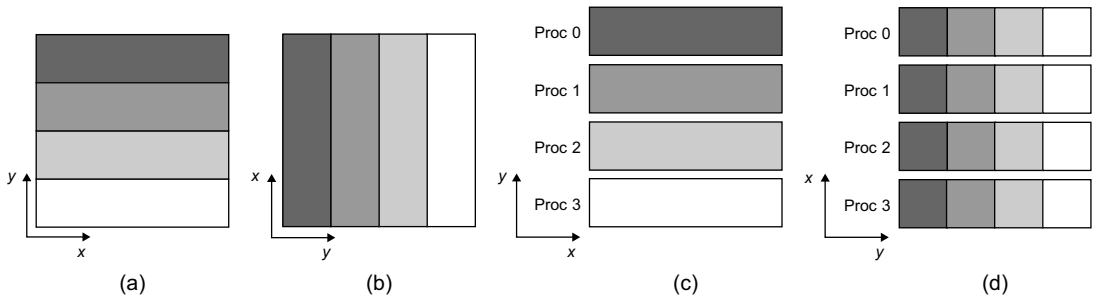


**FIGURE 39.5**

2-D arrays, transposition, and partition.

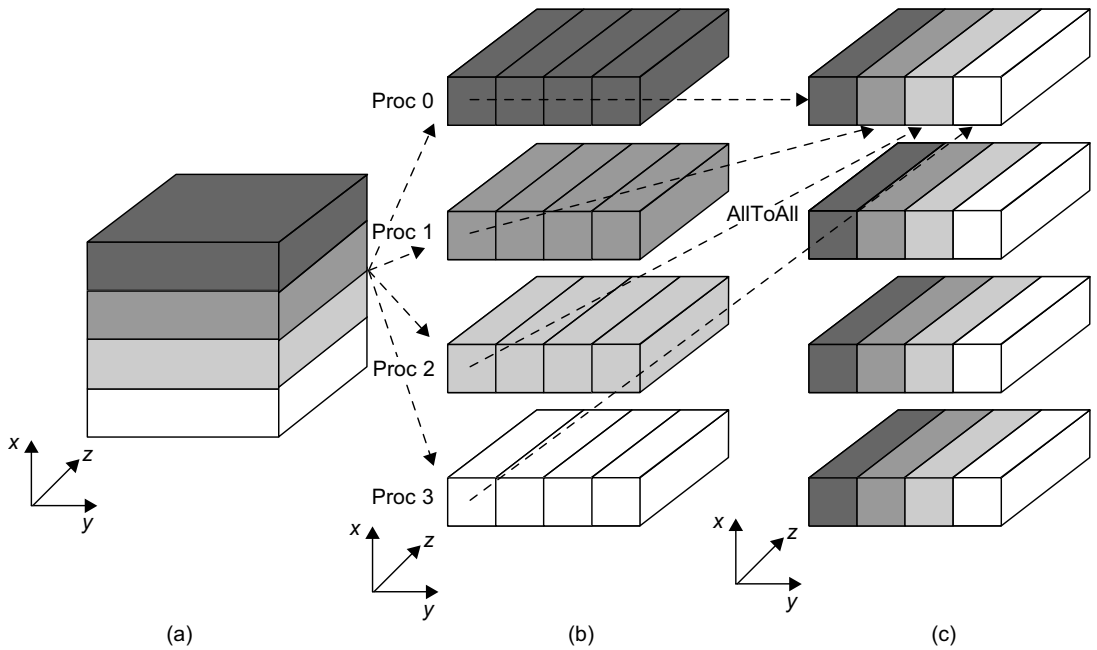


**FIGURE 39.6**

A three-dimensional FFT.

in which dimension $x$ is partitioned and distributed over four processes. Batched 2-D FFTs are performed for the slice of the original array on each process. Before AllToAll communication, the results of 2-D FFTs are partitioned along dimension $y$ for the four destination processes as

$$\frac{x_1}{[4]}\,\frac{x_0}{[2]}\,\frac{y_1}{[4]}\,\frac{y_0}{[2]}\,\frac{z}{[8]}.$$

After AllToAll communication, the array becomes

$$\frac{y_1}{[4]}\,\frac{x_0}{[2]}\,\frac{x_1}{[4]}\,\frac{y_0}{[2]}\,\frac{z}{[8]}.$$

The data are distributed along dimension $y_1$.

### 39.4.2 Avoiding Main-Memory Transpositions by Permuting Array Dimensions

We now introduce a method to adjust dimensions during data transfers (between different memory media) so that the combined effect of the adjustments together with GPU-based transpositions renders the expensive main-memory transpositions unnecessary (see Figure 39.7).

To support sophisticated parallelization, the source data are decomposed into a nine-dimensional array of complex numbers:

$$\frac{z_3}{[16]}\,\frac{z_2}{[2]}\;\;\frac{z_1}{[32]}\,\frac{z_0}{[4]}\,\frac{y_3}{[16]}\,\frac{y_2}{[2]}\,\frac{y_1}{[32]}\,\frac{y_0}{[4]}\,\frac{x}{[4096]}$$

where dimension $z_3$ indicates that the array is partitioned along dimension $z$ and distributed over 16 nodes. Dimension $z_2$ indicates that two separate pinned hmem areas are allocated for the two GPU devices (a restriction by CUDA for high-speed PCI data transfers).

In the first phase, each GPU computes 32 batches of four two-dimensional FFTs over dimensions $x$ and $y$ (which includes $y_3, y_2, y_1, y_0$). Each batch with four $4096^2$ complex numbers of 512 MB data is computed in one round of hmem-to-dmem transfer, CUFFT and dmem-to-IBbuffer transfer. Thus, the data in all outgoing infiniband buffers (of $16 \times 2 = 32$ HCA adapters in total) have a layout:

$$\frac{z_3}{[16]}\,\frac{z_2}{[2]}\;\;\frac{z_0}{[4]}\,\frac{y_3}{[16]}\,\frac{y_2}{[2]}\,\frac{y_1}{[32]}\,\frac{y_0}{[4]}\,\frac{x}{[4096]}.$$

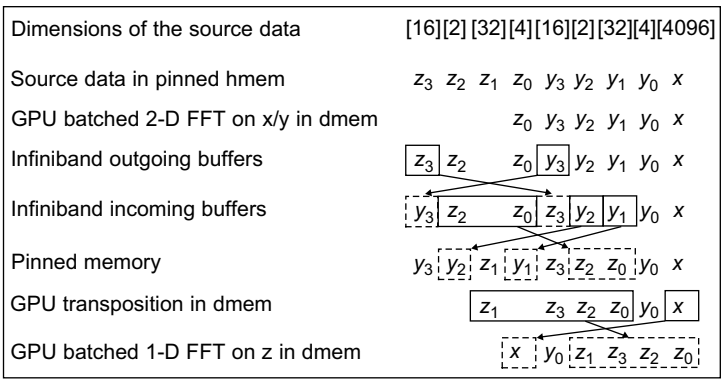


**FIGURE 39.7**

Permuted dimensions in the forward propagation.

In each of the 32 rounds, the AllToAll communication distributes data according to dimension $y_3$ over 16 nodes. Note that all HCA0 adapters communicate with each other, and so do HCA1 adapters. The layout in all incoming buffers then becomes:

$$\frac{y_3}{[16]}\frac{z_2}{[2]}\ \frac{z_0}{[4]}\frac{z_3}{[16]}\frac{y_2}{[2]}\frac{y_1}{[32]}\frac{y_0}{[4]}\frac{x}{[4096]}.$$

As our FFT is performed in place (512 GB data in 576 GB memory), incoming data can be stored only in the memory just released after GPU computation in the first phase. When copying the data in the incoming buffers back to the pinned hmem, we perform some subtle data adjustment. The purpose is to move the subdimensions of $z$ to the right so that the data of dimension $z$ become more aggregated in hmem. We also try to sort the subdimensions of $z$ as much as possible so that it is easier to coalesce dmem access for the later GPU computation. After 32 rounds, we will have a layout of the entire data as follows:

$$\frac{y_3}{[16]}\frac{y_2}{[2]}\ \frac{z_1}{[32]}\frac{y_1}{[32]}\frac{z_3}{[16]}\frac{z_2}{[2]}\frac{z_0}{[4]}\frac{y_0}{[4]}\frac{x}{[4096]}.$$

The permutation is achieved by copying the data in 128 KB units of $4 \times 4096$ complex numbers. This granularity is determined by the rightmost unmoved dimensions $y_0$ and $x$ in combination.

Adjustment is also needed when transferring data from pinned hmem to dmem for batched 1-D FFTs of dimension $z$, which also consists of 32 rounds according to dimension $y_1$. Thus, in each round, the data on each GPU have a layout:

$$\frac{z_1}{[32]}\frac{z_3}{[16]}\frac{z_2}{[2]}\frac{z_0}{[4]}\frac{y_0}{[4]}\frac{x}{[4096]}.$$

A slightly unconventional GPU transposition is then performed to swap the subdimensions of $z$ and dimension $x$ and reach a desirable layout for FFT:

$$\frac{x}{[4096]}\frac{y_0}{[4]}\frac{z_1}{[32]}\frac{z_3}{[16]}\frac{z_2}{[2]}\frac{z_0}{[4]}$$

from which $4096 \times 4$ 1D-FFT over dimension $z$ can be efficiently computed using our own single-GPU FFT code that handles the irregular order of the subdimensions of $z$. That code works in a semicoalesced manner that coalesces every 1/4 warp, and the distances between the subwarps are small. The GPU computation for FFT is well overlapped by PCI data transfers in our streaming-mode tests.

After FFT for $z$, the results need to be transferred back to the pinned hmem and communicated with other nodes to return to the original positions. The backward propagation of these intermediate steps is exactly the reversed forward propagation.

Thus, the permutations performed during data transfers have made it possible to aggregate elements of the third dimension $z$ on individual GPUs so that GPU-based transposition can move the elements and achieve locality for them. CPU-based transpositions are then no longer necessary!

## 39.5 PERFORMANCE RESULTS

Our FFT code (using GPUs) is compared with FFTW version 2.1.5 and Intel MKL version 10.2.1.017 (not using GPUs) on PKU McClus. The performance is calculated by:

$$\frac{\sum_{d=1}^{D} M_d(5N_d log_2 N_d)}{\text{execution time}}$$

where $D$ is the total number of dimensions, $M_d = E/N_d$ is the number of FFTs along each dimension, and $E$ is the total number of data elements. The execution time is obtained by taking the minimum time over multiple runs.

Figure 39.8(a) shows the performance of our FFT, FFTW, and MKL when performing single-precision C2C 3-D-FFT on 4 or 16 nodes with different data sizes. The experiments for CPUs use all cores (e.g., 16 CPUs with 64 cores in total), which perform the best for FFTW and Intel MKL when running eight threads per node. GPU experiments do not use CPUs for computation. Instead, eight CPU threads are used for controlling GPUs, Infiniband buffers, and data transfers. On 16 nodes, our FFT reaches 209 GFLOPS and completes 4096 3D-FFT in 54 seconds; MKL does not process more than 256 GB data as out-place workspace of the same size is used; FFTW does not process more than 32 GB (for OS-related reasons). Figure 39.8(b) shows the performance of our FFT, FFTW, and MKL when performing 3-D-FFT of 32 GB data on different numbers of nodes. PKUFFT demonstrates a speedup 24.3× over FFTW and 7× over MKL. Figure 39.9 shows the corresponding performance results for double-precision complex-to-complex 3-D-FFT. PKUFFT achieves 5.4× speedup over FFTW and MKL. The dramatically improved performances are attributed to the following factors:

1. Data transfers through two PCIs are faster than sustained large-scale hmem accesses (exceeding the size of cache) by CPU.
2. The dmem is much larger than the CPU cache, and that means larger subtasks can be processed in whole, and this reduces the need for reloading data.
3. Algorithmic improvement that uses fast GPU transpositions and does not need expensive CPU transpositions in hmem.
4. An improved communication interface based on IB/verbs, which accounts for roughly 38.8% of the total time.

It is notable that three out of the four factors are related to GPU.

## 39.6 CONCLUSION AND FUTURE WORK

On GPU clusters, it is easy to achieve high speedups for computation-intensive tasks and bandwidth-intensive tasks with good data locality. Large-scale FFT is a bandwidth-intensive task without locality. Such a task initially appears to be hard for GPUs to accelerate. The surprising high performances reveal an interesting picture. Several GPU-related factors have helped us squeeze more memory

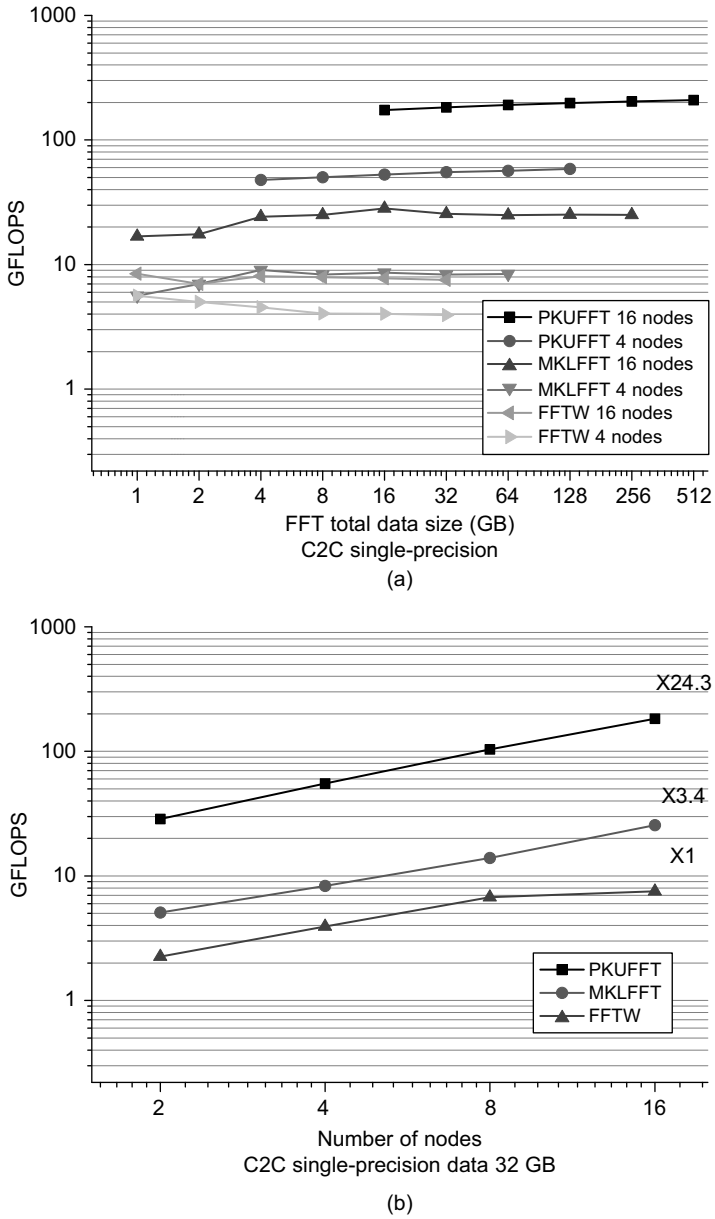


**FIGURE 39.8**

Single-precision 3D-FFT performance results of PKUFFT, FFTW 2.1.5, and Intel MKL 10.2.1.017.

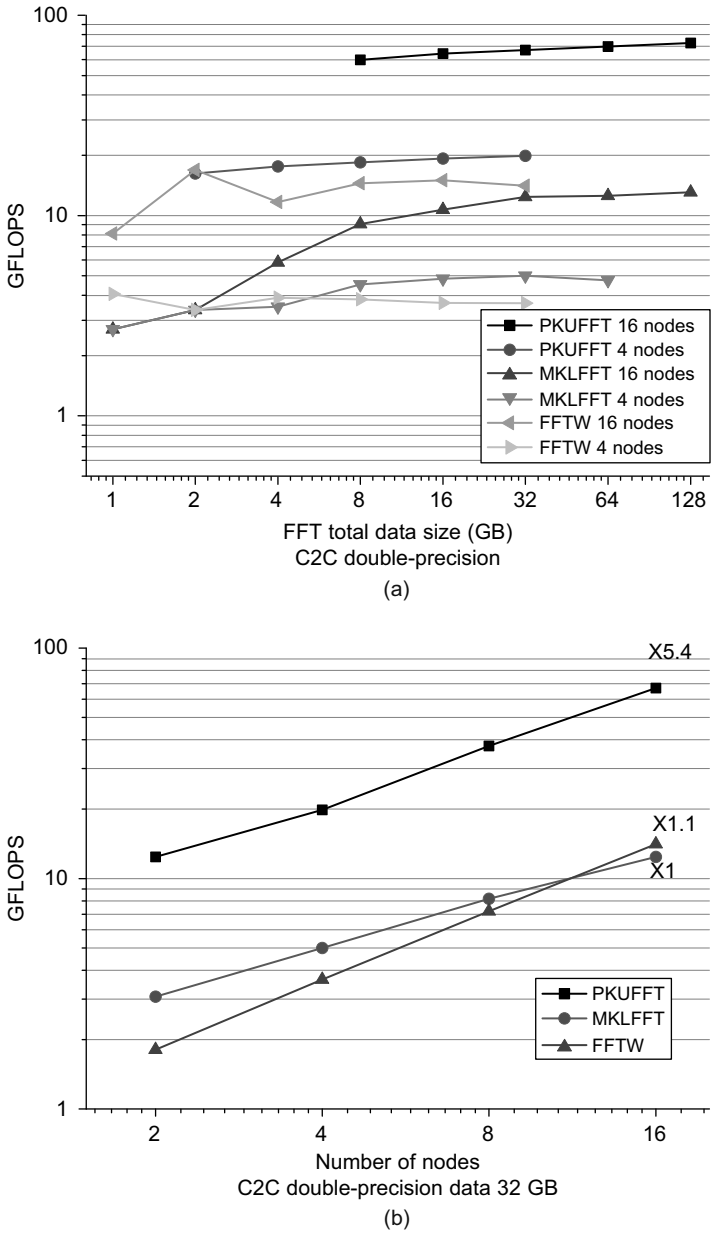


**FIGURE 39.9**

Double-precision 3-D-FFT performance results of PKUFFT, FFTW 2.1.5, and Intel MKL 10.2.1.017.

bandwidth out of each individual node. The main technical contribution is the algebraic manipulation of array dimensions. Without the method, it will be hard for a programmer to correctly determine various sophisticated index expressions. The exposed memory hierarchy of GPU clusters, on one hand, complicates programming, but on the other hand, presents an opportunity for optimization of data locality. The technique is readily applicable to a wide range of tasks.

Our design and implementation are based on the current restrictions of the hardware and software systems. If IB/verbs could use CUDA's pinned hmem directly as buffers, then data transfers from/to the buffers would be saved. Some progress (known as "GPU Direct") has been made to allow CUDA pinned hmem to become IB buffers directly (with certain size limits). The technique of dimension manipulation is still applicable to saving hmem transpositions. In that case, data adjustment in shorter messages can be performed during AllToAll communications.

A version of this chapter has been published at ACM Proceedings of ICS 2010 [11]. The authors would like to thank all anonymous referees for their helpful comments from which the preparation for this version has benefited.

## References

[1] X. Cui, Y. Chen, H. Mei, Improving performance of matrix multiplication and FFT on GPU, in: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, 8–11 December 2009, Shengzhen, China, 2009, pp. 42–48.

[2] M. Fatica, Accelerating linpack with CUDA on heterogenous clusters, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 8–8 March 2009, Washington, DC, 2009, pp. 46–51.

[3] V. Volkov, J.W. Demmel, Benchmarking GPUs to tune dense linear algebra, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 15–21 November 2008, Austin, Texas, 2008.

[4] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 8–8 March 2009, Washington, DC, 2009, pp. 79–84.

[5] E. Chu, A. George, Inside the FFT black box — serial and parallel fast Fourier transform algorithms, CRC Press, Boca Raton, Fla., 2000.

[6] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, High performance discrete Fourier transforms on graphics processors, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 15–21 November 2008, Austin, Texas, 2008.

[7] A. Nukada, S. Matsuoka, Auto-tuning 3-D FFT library for CUDA GPUs, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 14–20 November 2009, Portland, Oregon, 2009, pp. 1–10.

[8] A. Nukada, Y. Ogata, T. Endo, S. Matsuoka, Bandwidth intensive 3-D FFT kernel for GPUs using CUDA, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 15–21 November 2008, Austin, Texas, 2008, pp. 1–11.

[9] V. Volkov, B. Kazian, FFT prototype. http://www.cs.berkeley.edu/~volkov/, 2009.

[10] R.C. Agarwal, F.G. Gustavson, M. Zubair, A high performance parallel algorithm for 1-D FFT, in: Proceedings of the 1994 conference on Supercomputing, December 1994, Washington, DC, United States, 1994, pp. 34–40.

[11] Y. Chen, X. Cui, H. Mei, Improving performance of matrix multiplication and FFT on GPU, in: Proceedings of the 24th ACM International Conference on Supercomputing, 2–4 June 2010, Tsukuba, Ibaraki, Japan, 2010, pp. 315–324.