2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies

Implementation of 3D FFTs Across Multiple GPUs in Shared Memory Environments

Nimalan Nandapalan, Jiri Jaros, and Alistair P Rendell
Research School of Computer Science
ANU College of Engineering and Computer Science
Australian National University, ACT 0200, AUSTRALIA
{*Nimalan.Nandapalan, Jiri.Jaros, Alistair.Rendell*}*@anu.edu.au*

Bradley Treeby
Research School of Engineering
ANU College of Engineering and Computer Science
Australian National University, ACT 0200, AUSTRALIA
Bradley.Treeby@anu.edu.au

***Abstract*—In this paper, a novel implementation of the distributed 3D Fast Fourier Transform (FFT) on a multi-GPU platform using CUDA is presented. The 3D FFT is the core of many simulation methods, thus its fast calculation is critical. The main bottleneck of the distributed 3D FFT is the global data exchange which must be performed. The latest version of CUDA introduces direct GPU-to-GPU transfers using a Unified Virtual Address space (UVA) that provides new possibilities for optimising the communication part of the FFT. Here, we propose different implementations of the distributed 3D FFT, investigate their behaviour, and compare their performance with the single GPU CUFFT and CPU-based FFTW libraries. In particular, we demonstrate the advantage of direct GPU-to-GPU transfers over data exchanges via host main memory. Our preliminary results show that running the distributed 3D FFT with four GPUs can bring a 12% speedup over the single node (CUFFT) while also enabling the calculation of 3D FFTs of larger datasets. Replacing the global data exchange via shared memory with direct GPU-to-GPU transfers reduces the execution time by up to 49%. This clearly shows that direct GPU-to-GPU transfers are the key factor in obtaining good performance on multi-GPU systems.**

***Keywords*-GPU; UVA; unified-virtual-address; multi-GPU; FFT; distributed; shared-memory;**

I. Introduction

The use of graphics processing units (GPUs) as general-purpose massively-parallel processors is now common place in high performance computing systems. Initially this involved augmenting each node of a distributed memory system with a single GPU. Typical node hardware can, however, support multiple GPUs and, as node CPUs become increasingly multicore, the trend would suggest that each node will become populated with multiple GPUs.

To date, relatively little work has been reported on optimising algorithms to run on multiple GPUs attached to a single shared memory host. This paper considers this issue within the context of the Fast Fourier Transform (FFT) algorithm [1]. The FFT is a core component for many computational techniques, including signal processing, fluid dynamics, molecular dynamics, medical imaging, etc.

Our particular interest in the use of multiple GPU systems is driven by the desire to perform large-scale ultrasound simulations using the k-space pseudo-spectral method in time-frames that are clinically meaningful [2]. The k-space method makes extensive use of large 3D FFTs (dimensions of 1024^3 or greater), which constitute over half of the total computation time.

This paper is structured as follows. In section II we consider the hardware and software environment in detail. Section III outlines related work. Sections IV and V describe our algorithm and the use of direct device-to-device transfers respectively. Section VI presents our performance results, while section VII contains conclusions and discussion.

II. Multi-GPU Hardware and Software

The multi-GPU system used in this work is based on the Tyan barebone TYAN FT72B7015 [3]. The motherboard has two LGA 1366 sockets for Intel Core i7 processors in a NUMA configuration (see schematic in Figure 1). Each socket is populated with a six-core Intel Xeon X5650 processor giving a total of twelve physical cores. The server is equipped with twelve 4 GB memory modules (48 GB RAM) and has an aggregated memory bandwidth of 2×25 GB/s. Communication between the CPUs is supported by the Intel QuickPath Interconnection (QPI) with a theoretical bandwidth of 12 GB/s.

The QPI also connects each CPU with an Intel IOH chip that offers various I/O connections including a total of four PCIe x16 links. Each PCIe link is branched by a PLX PEX 8647 switch to give a total of eight PCIe slots. As this system is designed as a node in a cluster, one PCIe slot is reserved for a high-bandwidth interconnect (Infiniband). The remaining seven slots are populated with GPUs.

Each GPU is an NVIDIA GeForce GTX 580 with 512 CUDA cores and 1.5 GB of memory. Access to the CPU by the GPUs or vice versa is provided by the PEX bridge multiplexing the PCIe x16 links as required on demand. All GPUs use NVIDIA CUDA 4.1 [4].

A typical CUDA workflow involves creating a task on the host (CPU) side, allocating memory for task data on the device (GPU), copying that data to the device, and then executing the task "kernel" on the device. When the kernel completes, data is retrieved from the device before the device memory is freed.

When exploiting multiple devices, multiple tasks are executed simultaneously. If the tasks are mutually independent,

978-0-7695-4879-1/12 $26.00 © 2012 Crown Copyright
DOI 10.1109/PDCAT.2012.79

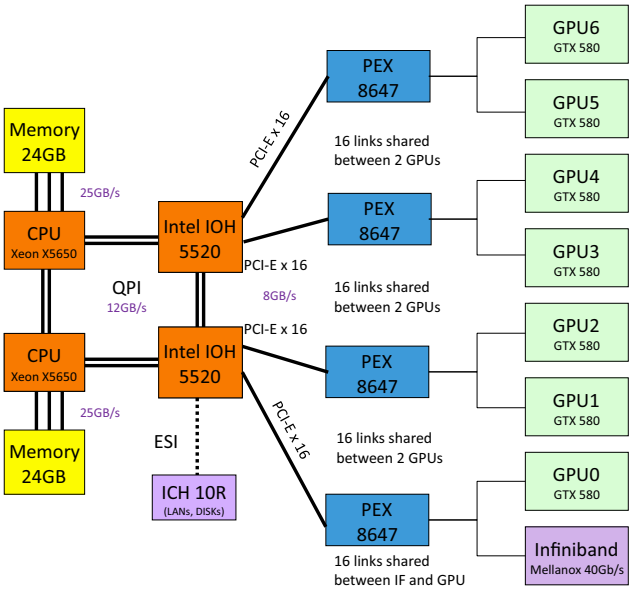


Figure 1. Schematic of the core components on the motherboard of a multi-GPU shared-memory system.

no communication among devices is necessary – if dependent, communication is required. Historically, inter-device communication in CUDA was performed by the host. That is, the host would collect pieces of data from each device into its memory space, and then send the relevant data to the relevant destination device memory.

The latest versions of CUDA introduce the Unified Virtual Address space (UVA). This enables direct access to remote device memory by CUDA kernels and CUDA memory copy routines. However, on the platform used here CUDA device-to-device communication is limited to devices that are under the same Intel IOH bridge (e.g. GPU0, GPU1 and GPU2 in Figure 1). This is due to Intel's implementation of the PCIe 2.0 protocol in the 5520 chipset and its incompatibility with the Intel QPI [5].

Recognizing the time required to transfer data to and from the GPU devices, NVIDIA included in CUDA the concept of streams. A CUDA stream represents a queue of GPU operations (kernels, memory transfers) that get executed in a specific order. Effectively the GPU devices can be thought of as being made of two parts: a copy engine (the device DMA controller); and a compute engine (the CUDA cores). With multiple streams the copy engine can execute a memory transfer from one stream while the CUDA cores are busy processing a kernel from another stream.

III. Related Work

The de facto standard for CPU FFT implementations is FFTW [1], [6], now in version 3.3. It is available for shared and distributed memory systems. For NVIDIA GPUs there is CUFFT [7] (we use CUFFT version 4.1), however, this provides no interface for utilizing multiple GPU devices.

P3DFFT is an open-source off-the-shelf framework for distributing 3D FFT [8]. It does not compute the transform itself, but handles all of the decomposition and communication tasks required for performing a distributed 3D FFT. It makes use of a 2D (pencil) decomposition, using a localized library, such as FFTW, for the component transform.

Csechowski et al. [9] extended P3DFFT to create DiGPUFFT for use on their GPU cluster. They observed significant performance bottlenecks which they attribute to the cost of communication over the PCIe bus between CPU and GPU. This was estimated to represent approximately 27% of the total time taken by the 3D FFT.

PKUFFT [10] is similar to DiGPUFFT in that it uses a pencil data decomposition with GPUs to perform the actual computation. Whereas P3DFFT appears to be limited to real-to-complex (R2C, forward) and complex-to-real (C2R, backward) transforms, PKUFFT includes complex-to-complex (C2C) transforms. It differs from P3DFFT in the data manipulation it performs at the various stages, and the factoring of architectural elements of GPU clusters into the decomposition and underlying computations.

ULSFFT [11] look at a recursive composition, essentially a restructuring of the butterfly graph to allow a scatter-gather processing model. Gu et al. [12] propose a number of techniques for performing large FFTs when the data is maintained out of a single devices memory space.

None of the above explicitly consider the case of multiple GPUs hosted by a single node; so direct communication between devices was not considered. Also aspects of all these systems are now obsolete, e.g. PKUFFT is presented for version 2.3 of CUDA, and the PCIe controller versions used in the various systems are not immediately apparent.

IV. 3D FFT Implementation

Our implementation for processing the 3D FFT on multiple GPUs can be described in three phases as illustrated in Figure 2. In the 3D domain we refer to X as the depth, Y as the width, and Z as the height with the upper-left-forward corner as the origin or zeroth element. Consecutive elements in X are stored in consecutive memory locations (i.e. the matrix is stored in row-major order).

The first phase begins by partitioning the matrix in the Z dimension into *#batches* contiguous batches ($\#batches = \frac{Z}{batch\_size}$) where the dimension of each batch is: $batch\_size \times Y \times X$. Each batch is also divided in the Y dimension to give $batch\_size \times batch\_size \times X$ pencils, where the number of pencils, $\#pencils = \frac{Y}{batch\_size}$. This is shown as Phase 1 in Figure 2, where the cubic matrix is divided into three batches, each with three pencils (left panel, Phase 1, Figure 2). These batches are distributed among the GPUs (center panel, Phase 1, Figure 2), and *batch_size* 2D FFTs of size XY are performed (right panel, Phase 1, Figure 2)

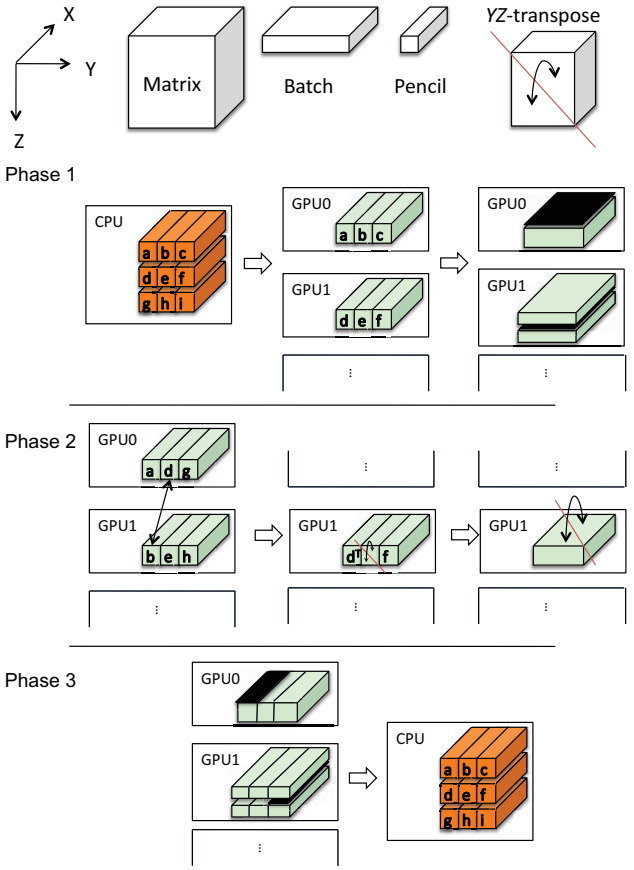


Figure 2. Decomposition and Movements of Pencils: Phase 1) matrix distributed in batches, 2D FFTs performed; Phase 2) communication between devices (D2D) with pencils and memory manipulations (transposes); Phase 3) 1D FFTs and transformed data returned in original format.

for each batch using the CUDA provided CUFFT library. At the end of Phase 1 the device memory contains the XY component of the result.

To complete the 3D FFT each Z component, which is presently distributed across multiple devices, is rearranged to be contiguous within a device. This is referred to as Phase 2 in Figure 2 and requires an all-to-all communication. This communication is managed by traversing the upper-right triangular set of pencils, calculating the destination buffer for the pencil, and performing a data swap (left panel, Phase 2, Figure 2). The destination is uniquely identified by the ID number of the device, a buffer identified by the ID of the batch stored in it, and an offset into the buffer for the number of pencils before it. These values are derived as follows:

- *device ID* = $\dfrac{pencilId}{\#pencils/\#GPUs}$ (where *pencilId* is the unique number of the pencil within the batch with range $\big[0$–*#pencils*$\big)$).
- *batch ID* = pencilId $(\mathrm{mod}\ \dfrac{\#pencils}{\#GPUs})$.
- *pencil offset* = *batchId* (the unique number of the batch in the range $\big[0$–*#batches*$\big)$).

After the pencil movement in Phase 2, all Z data is on the correct device for the final 1D FFT, although it is not contiguous. This is addressed by performing a YZ transposition on each pencil (center panel, Phase 2, Figure 2), followed by an XY transposition on the entire batch (right panel, Phase 2, Figure 2). The net effect of Phase 2 is a global YZ transpose plus a global XY transpose. With this complete the final stage consists of *batch_size*$\times Y$ 1D FFTs each of size Z (left panel, Phase 3, Figure 2).

The 3D FFT of the data now exists in the device memory space, albeit in a slight permutation. Performing the inverse of the memory operations/communications will place the transformed data in the same orientation as the original data back into main memory (right panel, Phase 3, Figure 2).

V. Device-To-Device Communication

The challenging phase of the distributed 3D FFT is the distributed 3D matrix transposition in the second phase. All the elements of the 3D matrix have to be rearranged and redistributed which leads to at least *#GPUs* $\times$ (*#GPUs* -1) device-to-device (D2D) transfers and at most $Z \times (Z-1)$. These two cases occur as the number of transfers (*#transfers*) and the size of each transfer (*transfer_size*) is dependent on the *batch_size*. In a simplified analysis, *#transfers* $= \frac{Z}{batch\_size} - \frac{\#batches}{\#GPUs}$ and *transfer_size* $=$ *batch_size* $\times Y \times X$. Poorer performance is expected in the second case, when the *batch_size* is small, as there is a greater number of smaller transfers which are less likely to saturate the PCIe bus.

To understand the communication characteristics and bottlenecks of the given server architecture, we designed programs that perform data exchanges (swaps) between pairs of GPUs under different scenarios. These programs take a list of device IDs as pairs and swap the data, i.e. given a list (1, 2, 3, 4), two swaps will be performed: GPU1 with GPU2, and GPU3 with GPU4. Four different swap methods were designed: 1) a swap via host memory; 2) a staged CPU swap, where one transfer is performed via CPU and the other by direct D2D copy; 3) a direct D2D memory copy; and 4) a swap via kernel, where a GPU kernel uses registers to carry out the swap. In order to compare the methods we measured the effective bandwidth over the PCIe, defined as the rate at which the net data movement is achieved (in this case $\frac{\#GPUs \times transfer\_size}{\text{time}} \frac{\text{GB}}{\text{s}}$, where *#GPUs* is the number of devices in the list).

In method 1 the communication is performed via the host's memory (main memory) in two steps. First the data packages from the devices are gathered and stored in main memory in an asynchronous manner. After synchronising, the data packages are scattered to the particular devices.

Method 2 uses the CPU to stage part of the swap which takes three steps. In the first step the data of the first device

Table I
EFFECTIVE BANDWIDTH FOR INTER-DEVICE DATA SWAP (GB/S)

Method	2 GPUS		4 GPUS		6 GPUS	
	Min.	Max.	Min.	Max.	Min.	Max.
1) via CPU	2.95	5.38	3.83	7.55	4.54	5.71
2) staged CPU	2.82	4.71	2.83	9.41	4.18	9.88
3) ptr swap	2.91	8.11	2.97	16.20	4.39	24.29
4) kernel	5.94	8.24	5.96	16.46	-	-

is copied to host main memory. In the second step, a D2D memory transfer is used to move data from the second device to the first one overriding memory freed in the previous phase. Finally, the host transfers the first device's data to the second one from the temporary host buffer.

Method 3 achieves the communication using built-in CUDA functions to perform two asynchronous D2D transfers followed by pointer swapping. For this method, two distinct buffers have to be allocated on each device (the source and destination). One buffer acts as an input/output buffer, while the other contains the current data. The pointer swapping between the buffers marks the new transferred data as current, and the old current buffer as reusable.

Method 4 uses a kernel to handle the communication in place. The kernel is executed only on one device in the pair. As the UVA enables direct accesses to the remote memory, the kernel first loads the data on the executing device into local registers. Then it accesses the remote memory to perform the transfer from the other device to this. The swap is finalised by storing the value in the local registers to the remote memory.

VI. DISTRIBUTED 3D FFT PERFORMANCE RESULTS

The first set of results we present are for the effective bandwidths of the four methods of swapping data between devices. A range of bandwidths were recorded by varying the device pairs used by the methods and are displayed in Table I. Both bandwidths are important as an all-to-all communication is required.

When any communication involving the CPU occurs (methods 1 and 2), the observed effective bandwidth is significantly limited. These methods have the smallest bandwidth when devices on the same IOH chip or PEX switch are used, which limits the bandwidth in and out of the CPU. Conversely, methods 3 and 4 achieved their maximum bandwidths in these conditions due to the aggregated bandwidth of the PEX switches. These two methods, not involving the CPU, only move data when necessary. In contrast, method 1 has to move twice the minimum amount of data and method 2 one-and-a-half times the minimum amount over the PCIe network which lowers the effective bandwidth. All methods except method 4 require some amount of additional memory on the host or device. However, method 4 is limited to using devices on the same IOH chip. Although method 3 works with devices on different IOH chips, it requires data to be staged by the CPU, which is managed differently to method 1 as the effective bandwidths vary significantly depending on device selection.

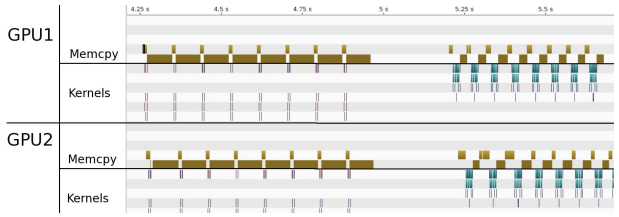
Table II
PERFORMANCE FOR ASSORTED *batch_sizes* AND SIMPLE DISTRIBUTED 1024^3 COMPLEX SINGLE-PRECISION FFT ACROSS 6 GPUS

batch_size	Time (seconds)		Size of Pencils
	1 Stream	2 Streams	
1	8.96	8.72	8 KB
2	7.27	7.30	32 KB
4	6.35	6.42	128 KB
8	6.33	6.33	512 KB
16	6.17	6.15	2 MB
32	6.77	6.57	8 MB

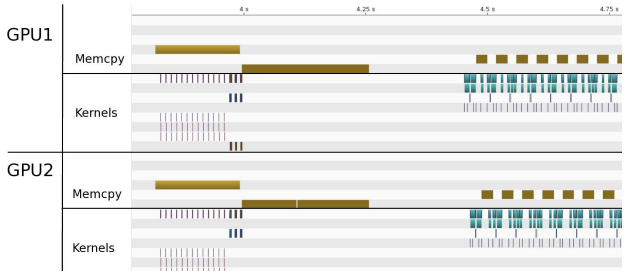
Considering these communication characteristics, we created three implementations for the distributed 3D FFT. The first implementation, simple, is a point of reference differing to the details in Section IV only in that the matrix is maintained in main memory, and batches and pencils are transferred on demand via the CPU to the GPUs for processing. The second implementation, D2D via ptr. swap (pointer swap), and third implementation, D2D via kernel, follow the technique in Section IV and perform the D2D communication directly between devices managing them via pointer swapping and a kernel respectively.

For all three of these implementations the *batch_size* is one parameter which can affect the performance in two ways. Table II demonstrates the effect of varying the *batch_size* for the simple implementation of the distributed 3D FFT. The first way this impacts performance is because this value directly changes the size of the largest contiguous set of data that is accessed and transferred, i.e. the size of the transfers on the PCIe. In order to saturate the PCIe bus and operate at maximum bandwidth, sufficiently large messages are required. This can be seen in the speed up as the *batch_size* increases, and the 25% improvement from a size of 1 to 32.

The second way in which the *batch_size* can affect performance is in the application of streams. This factor defines the size and consequently the time of the computations. A shorter time results in a higher granularity of tasks (kernels, or memory transfers) to be scheduled between the copy and compute engines of the device. If the computational component is significant, then the *batch_size* should be balanced according to the bandwidth to efficiently overlap computation and communication. Table II suggests that this component is insignificant, accounting for approximately 4% of the total time. We also observe some loss of performance



(a) Simple Profile



(b) D2D via Pointer Swapping Profile

Figure 3. Timeline of GPU activity showing memory transfers (brown), in-device transposes (green), and FFT kernels (blue).

for select *batch_sizes* when streamed, which we attribute to the dynamic nature of the device scheduler and the alignment of data in intermediary caches.

In order to confirm this, the implementations were profiled using the NVIDIA Visual Profiler [4]. Figure 3 provides a timeline of GPU activity for the simple and D2D via kernel implementations. This confirms that the computational component is not significant, and is hidden by communication.

In Table III we present results for the three implementations on multiple GPUs alongside results for FFTW on 6 and 12 CPU cores and CUFFT (where applicable). Each FFT was applied to complex 3D single-precision cubic matrices with dimensions 512^3, 768^3, and 1024^3. The performance was measured as the average of the times taken to perform the FFT. Times were recorded for *batch_sizes* 1, 2, 4, 8, 16, and 32, and for one and two streams per device. The fastest average times were used. The results for the simple implementation provides a baseline for the D2D versions. The performance of this version is slower in all cases to the reference CPU (FFTW) and GPU (CUFFT) versions.

The results for the D2D via ptr. swap implementation are not available for 1024^3, and 768^3 for two and four GPUs, as the total device memory is not enough for the swap method which requires 7 GB for 768^3 and 16 GB for 1024^3. Similarly, for the D2D via kernel implementation for 1024^3 and 768^3 with two GPUS. When using six GPUs, the D2D via ptr swap implementation stages some communications through main memory, and the D2D via kernel implementation can not be run due to the Intel IOH chips preventing the UVA accessing more than four devices.

Table III
TIME (SECONDS) TO COMPUTE 3D COMPLEX FFT

Implementation	Cores/GPUs	Size 512^3	768^3	1024^3
simple	1	1.74	6.32	14.05
	2	1.21	4.26	9.81
	4	0.81	2.78	6.52
	6	0.77	2.64	6.16
D2D via ptr. swap	2	0.61	-	-
	4	0.50	-	-
	6	0.64	2.07	-
D2D via kernel	2	0.80	-	-
	4	0.74	2.46	-
CUFFT	1	0.56	-	-
FFTW	6	0.61	2.01	5.64
	12	0.30	1.07	2.82

- missing value, not enough memory on GPUs.

In all cases the implementations using direct D2D transfers were faster than the simple version communicating via host memory. The greatest speed up was observed with the D2D via ptr. swap implementation on two GPUs over the 512^3 matrix where the performance improved by 49%.

Despite the kernel swapping approach performing better in the effective bandwidth tests, the performance of the D2D via kernel implementation was in general worse than the D2D via ptr. swap version. However, for smaller *batch_sizes* this method was faster. An explanation for this performance for large *batch_sizes* and matrices is the limit in the resource configuration of kernels. Another, is that the kernel both involves transfers and computation and consequently does not schedule well. It should be noted that the D2D via kernel implementation requires half the memory of the other implementation which allows it to compute the 768^3 matrix with only four GPUs. Although using fewer GPUs, this was slower than the pointer swapping method with six GPUs by 16%. This suggests that the slower transfers staged through main memory with pointer swapping may have been masked by the other D2D communications.

Compared to CUFFT and FFTW, all three implementations were generally slower. However, at 512^3 the D2D via ptr. swap implementation over four GPUs was found to outperform CUFFT (a single GPU) by 12%, and FFTW when six cores on a single CPU were used by 18%. FFTW over 12 cores (both CPUs) was still faster in these conditions by at least 40%.

VII. CONCLUSIONS AND FUTURE WORK

The goal of this paper was to investigate the acceleration of the 3D FFT using multiple GPUs within a shared memory environment. There are two reasons to do this: to extend the

available memory beyond the limits of a single device, and to reduce computation time. The distributed 3D FFT consists of three phases. First, the data is distributed in batches, and the 2D FFT on each slice is performed. Second, the data is redistributed using pencils amongst the devices to make it contiguous in the third dimension. Third, the 1D FFTs are performed over the remaining dimension, and the data gathered back to the host. The most time consuming component is the data redistribution between devices. In this work we propose several different methods for this, including via host memory, and directly between devices. The most efficient means of data exchange is by swapping pointers between multiple buffers using asynchronous D2D CUDA UVA memory copies. Compared to global data exchange via shared memory, this reduces the execution time of the 3D FFT by up to 49%. This clearly shows that direct D2D are a key factor in obtaining high performance on multi-GPU systems.

Even using the most efficient distributed 3D FFT implementation (D2D via ptr. swap), the performance is strongly limited by the PCIe throughput. Compared to the bandwidth between a single GPU's compute cores and memory (200 GB/s), or the CPU cores and host memory (25 GB/s), the limit of 8 GB/s using PCIe 2.0 is a significant bottleneck. For example, using two GPUs to compute a 512^3 3D FFT is 12% slower than using a single GPU. However, using four GPUs allows access to a greater aggregate PCIe bandwidth resulting in a 12% speedup. This has significant implications for both the hardware and software development for multi-GPU systems. It is possible that the hardware constraints may be partially alleviated by the release of systems based on PCIe 3.0 which doubles the throughput of PCIe 2.0. An additional limitation is that the GTX 580 used in our multi-GPU system has only a single copy engine (unlike the Tesla series of NVIDIA hardware [13]), and thus can not exploit the duplex capability of PCIe.

Compared to using FFTW on a modern twin CPU socket setup, the performance of the distributed 3D FFT is worse, even when using six GPUs. This is again due to the bandwidth limitations imposed by PCIe. However, there are several scenarios where using GPUs may be advantageous. For example, if the data is already distributed in GPU memory space (as part of a larger GPU based simulation), or if the availability of data is bottlenecked by another factor such as an external network (Infiniband).

Due to the hardware limitations of the Intel IOH chips, the D2D implementations are limited to four GPUs. This places a ceiling on the aggregate GPU memory (size of the UVA space) and size of the largest FFT that can be performed. To use more devices we are forced to stage data in host memory, and transfer data on demand to GPUs for processing. This is less efficient but allows for larger problem sizes.

A possible extension of our implementations would be to incorporate some of the memory transfer operations into a custom FFT kernel in each phase. The effect of this would be to perform these operations as soon as possible, within the same kernel launch, without the need for additional synchronisations. This would improve the balance between computational and communication tasks. Moreover, the CUFFT library requires additional memory for every batch of 2D and 1D FFTs processed. This could be avoided by implementing a custom FFT kernel.

ACKNOWLEDGMENT

This work was supported in part by the Australian Research Council/Microsoft Linkage Project LP100100588.

REFERENCES

[1] M. Frigo and S. Johnson, "FFTW: An adaptive software architecture for the FFT," in *ASSP*, vol. 3. IEEE, 1998, pp. 1381–1384.

[2] B. E. Treeby, J. Jaros, A. P. Rendell, and B. T. Cox, "Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method," *J. Acoust. Soc. Am.*, vol. 131, no. 6, pp. 4324–4336, 2012.

[3] MiTAC International Corporation. (2011, Sep.) Tyan ft72b7015 server barebone.

[4] NVIDIA Corp., "NVIDIA CUDA Programming Guide Version 4.1," NVIDIA, Tech. Rep., Nov. 2011.

[5] P. Micikevicius. (2011, Nov.) M07: High performance computing with cuda. SC11 Tutorial. [Online]. Available: http://sc11.supercomputing.org

[6] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[7] NVIDIA Corp., "CUDA Toolkit 4.1 CUFFT Library," NVIDIA, Tech. Rep., Jan. 2012.

[8] D. Pekurovsky and J. Goebbert, "P3dfft-highly scalable parallel 3d fast fourier transforms library," University of California, Tech. Rep., Nov. 2011. [Online]. Available: http://code.google.com/p/p3dfft/

[9] K. Czechowski, C. McClanahan, C. Battaglino, K. Iyer, P.-K. Yeung, and R. Vuduc, "On the communication complexity of 3D FFTs and its implications for exascale," in *ICS*, San Servolo Island, Venice, Italy, Jun. 2012.

[10] Y. Chen, X. Cui, and H. Mei, "Large-scale fft on gpu clusters," *ICS*, pp. 315–324, 2010.

[11] J. Glenn-Anderson, "Ultra large-scale fft processing on graphics processor arrays," *enparallelcom*, 2009. [Online]. Available: http://enparallel.com/ULSFFT.pdf

[12] L. Gu, J. Siegel, and X. Li, *Using GPUs to Compute Large Out-of-card FFTs*, 2011, pp. 255–264.

[13] NVIDIA Corp. (2012, Jul.) Tesla. [Online]. Available: http://www.nvidia.com/object/tesla-supercomputing-solutions.html