Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA

Akira Nukada∗†, Yasuhiko Ogata∗†, Toshio Endo∗†, Satoshi Matsuoka∗‡†

∗Tokyo Institute of Technology, Ookayama 2-12-1, Meguro-ku, Tokyo 1528552, Japan
†Japan Science and Technology Agency, 4-1-8 Hon-chou, Kawaguchi, Saitama 3320012, Japan
‡National Institute of Informatics, Hitotsubashi 4-5-6, Chiyoda-ku, Tokyo 1018430, Japan
{nukada,ogata}@matsulab.is.titech.ac.jp, {endo,matsu}@is.titech.ac.jp

Abstract

Most GPU performance “hypes” have focused around tightly-coupled applications with small memory bandwidth requirements e.g., N-body, but GPUs are also commodity vector machines sporting substantial memory bandwidth; however, effective programming methodologies thereof have been poorly studied. Our new 3-D FFT kernel, written in NVIDIA CUDA, achieves nearly 80 GFLOPS on a top-end GPU, being more than three times faster than any existing FFT implementations on GPUs including CUFFT. Careful programming techniques are employed to fully exploit modern GPU hardware characteristics while overcoming their limitations, including on-chip shared memory utilization, optimizing the number of threads and registers through appropriate localization, and avoiding low-speed stride memory accesses. Our kernel applied to real applications achieves orders of magnitude boost in power&cost vs. performance metrics. The off-card bandwidth limitation is still an issue, which could be alleviated somewhat with application kernels confinement within the card, while ideal solution being facilitation of faster GPU interfaces.

1. Introduction

Graphics Processing Units (GPUs) are increasingly starting to be used as commodity accelerators. The computation cores of GPUs are optimized for repeating simple graphical operations, resulting in sporting much higher memory bandwidth as well as floating-point performance. Although their current power consumption as a unit is fairly high (often 100 Watts or more), because of their massive compute power their flops-per-watt figure is much lower than that of conventional CPUs.

The difficulty, of course, is that GPUs are much less general purpose than conventional CPUs, so their applicability in a wide-ranging set of HPC applications must be carefully studied, especially to identify their pros and cons as well as devising effective algorithms, programming models, and programming techniques & methodologies, so as to attain optimal performance, thereby eliminating the “hype” factor and becoming truly General-Purpose Graphics Processing Units (GPGPUs) [1]. However, usages of GPUs for scientific computing have been mostly dominated by those with needs for a large number of tightly-coupled floating-point operations such as N-body problem [2], [3], or those based on kernel matrix multiplication [4] accelerations. As the nature of these applications is largely synonymous to graphics processing, i.e., abundance of independent parallelism, the ratio of floating point operations to memory access being large, as well as memory being accessed in a successive, “stream” fashion, they (obviously) can be programmed and accelerated fairly easily, using conventional shader graphics languages such as NVIDIA Cg [5] or Microsoft High Level Shader Language (HLSL), and using stream programming abstractions. BrookGPU [6] and Microsoft Accelerator [7] are extensions of C, and further develop these concepts by hiding away the complexities of underlying shader programming and allowing the programmer to focusing on stream programming. However, when the kernel algorithm(s) for an application requires descriptions beyond stream programming. GPU applicability is not well investigated.

Fast Fourier Transform (FFT) [8] plays an important role in numerous applications today. In particular FFT requires $O(N)$ memory access versus only $O(N \log N)$ floating point operations, requiring not only high computation throughput but also high memory bandwidth. Moreover, FFT requires exten-
sive stride memory access, so simple mapping to
stream programming could result in significant loss
in performance. NVIDIA’s CUDA (Compute Unified
Device Architecture) [9] holds high promise in this
regard—CUDA is a new GPU architecture as well as a
programming language based on C, and it allows more
flexible operations beyond stream programming for
such “irregular” kernels by extensive multi-threading
(in the 100s) and the ability for the threads to share
data rapidly via shared memory. The programming
language CUDA allows for a block of threads to be
specified as arrays for easy, SIMD-like specifications.
However, there are still many CUDA peculiarities
(and are common with other GPUs) that makes their
straightforward application difficult to attain the ex-
pected performances. One is that, although memory
bandwidth is abundant (60-100GByte/s), accesses must
be made in very large blocks in a resource-conscious
fashion. This is made difficult by the fact that many
threads could be accessing the memory in random
order, substantially deteriorating performance. Another
difficulty is that, many of the accelerators are installed
in I/O expansion slots such as the PCI-Express in-
terface; therefore the data transfer between the host
CPU and device often occupies a large percentage of
the total execution time. This makes it difficult for
accelerators to improve the performance of memory
intensive applications like FFT. As a result, the cur-
rently reported results of FFT on GPUs [10], [11], [12],
[13] have been only on par with conventional CPUs at
best, indicating that real performance of GPUs have
not been exploited yet despite the hopes.

Especially, we target 3-D FFT [14] which is used
in high performance computing area [15]. Our new
novel 3-D FFT algorithm for CUDA, optimized for
efficient memory access on CUDA GPGPUs, breaks
these barriers significantly, by demonstrating up to 3
times the performance of all 3-D FFT results reported
to date on GPUs, up to 84 GFLOPS per card, largely
equaling the performance seen on a single processor
of the latest NEC SX-9, and several times faster than
the most recent (quad-core) CPUs. We achieve this by
various careful programming techniques, including (1)
on-chip shared memory utilization, (2) optimizing the
number of threads and registers through appropriate
localization, and (3) avoiding low-speed stride memory
accesses.

The capacity of the device memory of currently
available products is almost 512MByte which is suf-
cient to support up to size 256^3 in single precision,
out-of-place complex-to-complex 3-D FFT, applicable
to many areas especially nano-science and life science.
For simplicity, the data size for each dimension is as-
sumed to be power of two in this paper. Our kernel ap-
plied to a real proteomic docking application achieves
orders of magnitude boost in power&cost vs. perfor-
ance metrics. The off-card bandwidth limitation is
alleviated with application kernels confinement within
the card, which is applicable in our case as well as with
many other applications. Still, the ideal solution being
facilitation of faster GPU interfaces, and we assess
the penalties associated with bus transfer, where the
GPGPU advantage does not disappear but diminishes
substantially. Some of our results are particular to 3-D
FFT, but many aspects are sufficiently general to be
applicable to other numerical kernels.

2. Architectural Overview of CUDA

A GPU can be regarded as a many-cores processor
supporting numerous fine-grain threads. Conventional
GPUs, however, did not allow any data exchange
between the processor cores except through exter-
nal memory. Consequently, GPU applications were
largely in nature stream processing, which performs
identical operations onto each element of the input
arrays. NVIDIA CUDA environment provides sets
of on-chip, fast shared memories for data exchange
between threads, as well as flexible access to the device
memory. This in theory greatly broadens the scope
of application kernels that can be effectively executed
on CUDA GPUs provided they exhibit substantial
parallelism, even irregular codes.

GeForce 8800 GTX with G80 core is the first gen-
eration CUDA GPU, consisting of sixteen Streaming
Multiprocessors (SM). Each SM further consists of
eight processor cores called Streaming Processors (SP),
in addition to 8192 registers, 16Kbyte shared memory,
constant cache memory and texture cache memory.
In effect, GeForce 8800 GTX is a massively-parallel
multi-core processor embodying 128 SPs in total.

As is with conventional GPUs, CUDA GPUs are
Single Instruction Multiple Data (SIMD) processor
which executes from the same instruction stream on
each SP. The present-day CUDA GPUs support up
to 768 active threads on each SM. Running a large
number of threads hides the latency of accessing
registers and device memory. The threads are divided
into groups of 32 called the warp, and instructions for
a warp are issued together.

Thread Block is a group of threads, where they are
executed on the same SM so that data exchange be-
tween the threads is possible using the shared memory
of the SM. The number of active thread blocks on each
SM is automatically determined from the resources
Table 1. Specifications of NVIDIA GeForce 8 series GPUs

<table>
<thead>
<tr>
<th>Model</th>
<th>Core</th>
<th>Process</th>
<th>SM #</th>
<th>Clock</th>
<th>GFLOPS</th>
<th>Capacity</th>
<th>Interface</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>8800 GT</td>
<td>G92</td>
<td>65nm</td>
<td>14</td>
<td>1.500 GHz</td>
<td>336</td>
<td>512MByte</td>
<td>1800MHz</td>
<td>57.6 GByte/s</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>G92</td>
<td>65nm</td>
<td>16</td>
<td>1.625 GHz</td>
<td>416</td>
<td>512MByte</td>
<td>1940MHz</td>
<td>62.0 GByte/s</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>G80</td>
<td>90nm</td>
<td>16</td>
<td>1.350 GHz</td>
<td>345</td>
<td>768MByte</td>
<td>384-bit</td>
<td>86.4 GByte/s</td>
</tr>
</tbody>
</table>

Table 1. Specifications of NVIDIA GeForce 8 series GPUs

requested by a thread block such as registers, shared memory, and number of threads.

All the current CUDA GPUs are based on the same architecture, but varies in different architectural parameters such as clock frequencies of core and memory, the number of SMs, the capacity and bandwidth of device memory, and the speed of the PCI-Express interface. Table 1 shows the specifications of GeForce 8 series CUDA GPUs used in our experiments. “GeForce 8800 GTS” is a G92 core GPU card with 512MByte memory (“GeForce 8800 GTS 512”). The G92 core is more power efficient and sports the higher performance than the older G80 core as it uses the 65nm process rule instead of the 90nm of the G80 on the 8800 GTX card. However, the memory bandwidth of 8800 GTX is the highest among the three GPUs. In comparison the peak performance of the latest AMD Phenom 9500 Quad-Core processor is 35.2 GFLOPS in double precision, and the memory bandwidth is less than 10GByte/s under the STREAM benchmark; so, 8800 GTS exhibits 5–6 times performance advantage. The question, of course, is why such a performance advantage does not translate into real-world application performance based on a 3-D FFT kernel for previous work. This is largely attributed to the unoptimized use of memory system of GPUs that are specialized for stream programming, as we describe next.

2.1. GPU Memory System Properties

Since the memory bandwidth of GPUs are competitive with expensive vector processors, as well as embodying multiple parallel compute units, we base our algorithm on the multirow FFT algorithm [16], [17], [18] known to be quite suitable for vector processors. The multirow FFT computes multiple 1-D FFTs simultaneously, and the same operations can be reused as a part of the overall 3-D FFT. The computation of multiple 1-D FFTs can be vectorized easily, being mutually independent, and so is suitable for massively-parallel SIMD processors such as GPUs.

Although CUDA allows flexible memory access, efficient memory access patterns are actually restricted on GPUs. Here we give two notable points in the implementation of our fast 3-D FFT algorithm.

Since the multirow FFT algorithm performs memory accesses to multiple streams generated from multiple 1-D FFTs running in parallel, its performance depends on the number of streams—our preliminary measurements have shown that using a larger number of streams reduces the number of memory access in total, but it also decreases the overall memory bandwidth, as the maximum number of active threads on each SM is limited by the available register resource. For 8800 GTX, the bandwidth decreased from 71.7 GBytes/s for a single stream down to 30.7 GBytes/s for 256 streams.

Secondly, each SP can execute one memory access operation per cycle. There, collective memory access operations of a half-warp, i.e. 16 threads can be coalesced into one access operation onto a single block of memory access by the hardware. There are several restrictions on coalescing, however: a) each thread must access successive addresses in the order of the thread number, b) only 32, 64, or 128 bit memory accesses can be coalesced, and c) the address accessed by the first thread of the half-warp must be aligned to either 64, 128, or 256 byte boundaries, respectively. Otherwise, coalescing does not occur, and multiple memory accesses are issued for each thread, even if they access a same memory block, resulting in substantial degradation of memory access performance. This is aggravated by the fact that, unlike classical vectors with numerous parallel memory banks with short access latencies for stride accesses, modern GPUs employ GDDR memories which are optimized for successive memory access operations, incurring heavy relative penalties for non-successive accesses. For this reason, we should not only satisfy the conditions for coalesced memory access but also perform successive memory access to large block using multiple warps or thread blocks.

3. A Fast 3-D FFT for CUDA Environment

3-D FFT consists of 1-D FFTs for each XYZ dimensions. To compute 1-D FFTs on the processors, all data along a dimension needs to be transferred from/to memory. The data transfer is fast only if they are stored on successive memory addresses. Especially, the cache memories of CPUs can be filled in efficiently.
Assuming the data for dimension X are contiguous, 1-D FFTs for dimension Y and Z force a kinds of stride memory accesses.

Many of typical implementations of 3-D FFTs [18], [19], [20] use explicit transposes to avoid the stride memory accesses. Here, we show the conventional six-step 3-D FFT algorithm.

Step 1. Compute 1-D FFTs for dimension X.
Step 2. Transpose from \((x,y,z)\) to \((z,x,y)\).
Step 3. Compute 1-D FFTs for dimension Z.
Step 4. Transpose from \((z,x,y)\) to \((y,z,x)\).
Step 5. Compute 1-D FFTs for dimension Y.
Step 6. Transpose from \((y,z,x)\) to \((x,y,z)\).

In this algorithm, the computations of 1-D FFTs are very fast. But three problems remain: (1) there are no computations during the transpose, (2) the transpose operations are still relatively slow, and (3) the number of memory access increases. They are serious for memory intensive applications such as FFT.

Using stride memory access may minimize access to the main memory, but its throughput becomes much lower than sequential memory access, especially for GPUs for the reasons stated above. Although this can be slightly alleviated by accessing multiple lines at once, the maximum allowed number of the lines is very small for GPUs because that is limited by the small capacity of the shared memory.

Each SM of CUDA GPUs contains a shared memory (currently 16Kbytes) that facilitates very fast data exchange between the threads within the SM. Although we can use the shared memory much like cache memory of CPUs, capacity is quite small, and care must be taken for their effective use in the algorithms.

Since CUDA kernels including FFT usually consist of two phases for latency hiding of memory access, namely (1) copies between the device memory and shared memory, and (2) computation using the data on shared memory, we need to perform ‘double buffering’ which effectively halves the available shared memory capacity down to 8Kbytes. As coalesced memory access transfers at least 64 byte block at once, with 16 half-warp threads, in the case of 3-D FFT of size 256\(^3\), 16Kbyte memory space must be allocated on the shared memory for 256 blocks, which of course twice the available capacity. Larger FFTs will require even more shared memory which is simply not available.

### 3.1. Our Bandwidth Intensive 3-D FFT kernel

There are many studies on the implementation of the FFT on a vector processors [16], [21]. Since GPU memories provide extremely higher bandwidth compared with generic CPUs, the algorithms for vector processors should be applicable for GPUs. But GPU memories are optimized especially for sequential memory accesses, whereas the memory system of vector processors also allow high speed stride accesses as far as no bank conflicts occur. Therefore, we propose a fast 3-D FFT algorithm for CUDA that only conducts sequential memory access (thus avoiding stride accesses), while confining the shared memory usage to be within the allotted size (currently 16Kbytes).

The algorithm is a combination of a transform for X axis using shared memory, followed by multirow FFTs for Y and Z axis. However, there are several issues to be resolved: despite that multirow FFT does not require any communication between threads, data for each thread must be stored in registers, and as such a large number of registers need to be allocated, but this is not possible for large FFTs with current hardware. For example, if the multirow FFT algorithm used for 256-point FFT, each thread needs more than 512 + \(\alpha\) registers resulting in allocation of 1024 registers per thread. As a result, only eight threads can be executed on each SM, thereby not satisfying the conditions for coalesced memory access, and finally as a result, performance will fall flat due to extremely poor memory bandwidth.

To avoid this problem, we perform four 16-point FFTs to compute a single 256-point FFT, i.e., the multirow FFT algorithm is used not for 256-point FFTs but for those 16-point FFTs. In practice, we implement the kernels of 16-point FFT with 51 or 52 registers, allowing 128 threads to run on an SM for sustaining very high memory bandwidth to the limit of the hardware. This strategy has its pros and cons; compared with direct 256-point FFT, the number of memory access doubles with 16-point FFTs. But the overall performance with 16-point FFTs turns out to be better, due to the effect of coalesced memory access. In fact, with 16-point FFT, we have observed more than 38GBytes/s of effective memory bandwidth while for the 256-point FFT we observe less than 10GBytes/s. The general lesson learned here is that, for memory bandwidth-intensive kernels one must carefully craft and tune the algorithm, with appropriate memory performance models and/or measurements such that occurrence of coalesced memory access would become the top priority, even if it seems intuitively wasteful. This is different from traditional vectors where hardware support for stride and other memory access patterns were very rich, so vector parallelization preceded over everything else, including stride and/or order of memory accesses.

Using 16-point FFTs for transforms along the Y and Z axes, our 3-D FFT algorithm consists of the
The following pseudo code shows the details. Splitting the indices for Y and Z into \(16 \times 16\), the input data is stored in a five-dimensional array ‘V’, and an alternative array ‘WORK’ is also allocated. The five for loops correspond to the five steps of our 3-D FFT algorithm above. When the sizes of Y axis and Z axis are equivalent, the operations in step 1 and 3 are same as those in step 2 and 4, respectively.

```c
COMPLEX V(256,16,16,16,16)
COMPLEX WORK(256,16,16,16,16)

for Z1,Y2,Y1,X
  WORK(X,*,Y1,Y2,Z1)=FFT256_1(V(X,Y1,Y2,Z1,*))
for Y2,Y1,Z2,X
  V(X,Z2,*,Y1,Y2)=FFT256_2(WORK(X,Z2,Y1,Y2,*))
for Y1,Z1,Z2,X
  WORK(X,*,Z2,Z1,Y1)=FFT256_1(V(X,Z2,Z1,Y1,*))
for Y2,Y1,Z2,X
  V(X,Y2,*,Z2,Z1)=FFT256_2(WORK(X,Y2,Z2,Z1,*))
for Z1,Z2,Y1,Y2
  V(*,Y2,Y1,Z2,Z1)=FFT256(V(*,Y2,Y1,Z2,Z1))
```

The function FFT256() is the compute kernel that computes a 256-point FFT, while FFT256_1() and FFT256_2() are the compute kernels of the first and second half of 256-point FFT, respectively—the details of these will be described in the next section. The expression “for Z1,Y2,Y1,X” indicates the four level nested loops in the order of the loop counter variables. (In practice, the nested loops are fused into a single physical loop with logical operations to calculate the indices for each loop, but we use the notation above for clarity). The loop is executed by threads and thread blocks in a cyclic fashion, not only to satisfy the conditions for coalesced memory access but also to perform successive memory access to a large block.

In the pseudo code as described above, the 16-point FFTs requires a transpose operation for the input array. More specifically, since a 256-point FFT is divided into 16-point FFTs, the array of input data \(V(X,Y1,Y2,Z1,Z2)\) must be finally stored in the array \(V(X,Y2,Y1,Z2,Z1)\). The four steps of our 3-D FFT perform effectively achieves these transposes, albeit seemingly excessively conducting the transpose operation. The step by step transpose operations might at a first glance seem to be similar to Stockham auto-sort algorithm [17], but our usages are completely different, i.e., the transposes are performed in the order so as to optimize the memory access patterns to maximize the memory bandwidth, as we describe below.

There are four memory access patterns as shown in Table 2 to read or write data for 16-point FFT. The achieved memory bandwidth depends on the combination of the access patterns as shown in Table 3 and Table 4. Here we can observe that the worst case is for both input and output access patterns to be C or D, while if at least one of them is pattern A or B, the achieved bandwidth is very close to those of single stream copy. The reason for this is that, for pattern A and B, the addresses accessed are close enough to each other, such that the memory access becomes similar to that of the single stream copy. As such, our current combination as described avoids the combinations of patterns C or D, thus the five steps as described above. Again, the general lesson here is that, even if it results in slightly excess memory operations, maximization of overall memory access bandwidth by avoiding the
“pitfall” i.e., inefficient memory access patterns, are important, and modeling and/or instrumentation should be conducted at each steps to investigate the bandwidth utilized at each step of the algorithm.

3.2. Implementation of Fast 3-D FFT kernels

As we described above, our 3-D FFT executes five CUDA kernels in total, but each one is different to optimize memory bandwidth. For step 1 to step 4, we employ coarse-grained parallelism, i.e., compute one 16-point FFT transform per thread. This is efficient as no data exchange between the threads is required, and as such no shared memory is allocated, but a large number of registers are required to maintain compute state per each thread. Here, overall performance is governed by the bandwidth of the device memory, as the number of floating point operations is small. The number of registers per thread is determined by the total number of threads required to saturate the memory bandwidth. In our case, we observed that we require at least 128 threads for each SM, restricting the number of registers to 64, which is fortunately sufficient to compute 16-point FFT.

Contrastingly, the kernel for step 5 computes transforms for X axis, whose memory access is successive. Here, we employ fine-grained parallelism using multiple threads of a thread block to compute a single transform in order to enable coalesced memory accesses, allowing very fast data transfer between the device memory and SPs as discussed earlier. Another reason we use fine-grained parallelism is the limitation in the number of registers: to compute a \(2^{256}\)-point FFT, at least \(2^{512}\) registers are necessary, so coarse-grained parallelism would be impractical; on the other hand for fine-grained parallelism computing a \(2^{516}\)-point FFT with 64 threads each thread uses only eight registers to store four complex numbers.

Computation of FFT requires twiddle factors [17], which are triangular functions usually pre-calculated and stored in a table, and each thread may use different value. For the twiddle factors, we can use one of the following four options:

1. registers. This option increases the number of registers, but the fastest.
2. constant memory. The constant memory provides only a 32-bit data in each cycle.
3. texture memory. This is a good option to save the number of registers.
4. calculate each time. This option takes additional processor cycles.

Considering these pros and cons, we selected “texture memory” for step 5, and “registers” for the other steps.

Since shared memory has 16 banks which are accessible in parallel, we employ a padding technique for efficient data exchange without bank conflicts. To save the amount of shared memory to be allocated, real parts are exchanged at first, and then the imaginary parts are exchanged.

3.3. Extension to larger 3-D FFT

To compute an FFT which is larger than the capacity of the device memory, we divide the large FFT into multiple small FFTs. For example, a 3-D FFT of size \(512^3\), which requires at least 1GByte and cannot fit on a 512MByte card, is split into eight 3-D FFTs of size \(512 \times 512 \times 64\) as follows.

Step 1. Repeat the following steps for eight sets.

1A. Send 64 XY-planes to device.
1B. Compute 3-D FFT of size \(512 \times 512 \times 64\).
1C. Multiply twiddle factors.
1D. Receive from device.

Step 2. Repeat the following steps for eight sets.

2A. Send 64 XY-planes to device.
2B. Compute 3-D FFTs of size \(1 \times 1 \times 8\).
2C. Receive from device.

More detailed pseudo code is described below.

```plaintext
COMPLEX VIN(512,512,512)
COMPLEX WORK(512,512,512)
DO I=1,8
   DO J=1,64
      SEND_TO_GPU(VIN(*,*,I+J*8-8))
   END DO
   FFT512X512X64()
   MULTIPLY_TWIDDLE(I)
   DO J=1,64
      RECV_FROM_GPU(WORK(*,*,I+J*8-8))
   END DO
   END DO
   DO I=1,64
      DO J=1,8
         SEND_TO_GPU(WORK(*,*,I*8+J-8))
      END DO
      FFT1X1X8()
      DO J=1,8
         RECV_FROM_GPU(VIN(*,*,I+J*64-64))
      END DO
      END DO
```

As the data is transferred twice via PCI-Express interface, the performance is greatly restricted by its
Table 5. The configuration of the system used for performance evaluations

<table>
<thead>
<tr>
<th></th>
<th>AMD Phenom 9500, 2.2GHz, Quad-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>AMD 790FX</td>
</tr>
<tr>
<td>Chipset</td>
<td>DDR2-800 SDRAM 1GBx4</td>
</tr>
<tr>
<td>OS</td>
<td>Fedora Core 8, Linux 2.6.23, 64-bit</td>
</tr>
<tr>
<td>Driver</td>
<td>NVIDIA Linux driver 169.09</td>
</tr>
<tr>
<td>Software</td>
<td>CUDA Toolkit 2.0beta2</td>
</tr>
<tr>
<td></td>
<td>GCC 4.1.2</td>
</tr>
</tbody>
</table>

GFLOPS = Bandwidth Intensive Kernel

Figure 1. The performance of 3-D FFT of size $256^3$

transfer speed; we investigate the overhead in the performance evaluation section.

4. Performance Evaluation

We evaluate performance of our 3-D FFT implementation using GeForce 8800 GT/GTS/GTX cards. Table 5 shows the configuration of the system used for the performance evaluation. 8800 GT/GTS cards and the AMD 790FX chipset are fully compliant with high speed PCI-Express 2.0 interface, while 8800 GTX only supports PCI-Express 1.1.

4.1. On-board Performance

Figure 1 shows the performance of 3-D FFT of size $256^3$. To calculate GFLOPS values, the number of floating-point operations of size $N^3$ is assumed to be $15N^3 \log_2 N$. CUFFT3D indicates the 3-D FFT routine of NVIDIA’s CUFFT library, included in CUDA Toolkit version 1.1. Here we compare on-board performance; all the values in the table do not include overhead of data transfer between host and device. We see that our bandwidth intensive kernel is more than three times faster than CUFFT on all the cards we used. In addition, our kernel is about twice faster than conventional algorithm using transposes.

Table 6. The elapsed time (ms) and achieved bandwidth (GByte/s) in each step of conventional algorithm using transposes

<table>
<thead>
<tr>
<th>Model</th>
<th>Step 1, 3, 5</th>
<th>Step 2, 4, 6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>8800 GT</td>
<td>5.74</td>
<td>46.7</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>5.09</td>
<td>52.7</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>5.52</td>
<td>48.5</td>
</tr>
</tbody>
</table>

Table 7. The elapsed time (ms) and achieved bandwidth (GByte/s) in each step of our bandwidth intensive kernel

<table>
<thead>
<tr>
<th>Model</th>
<th>Step 1, 3</th>
<th>Step 2, 4</th>
<th>Step 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Time (ms)</td>
<td>Time (ms)</td>
</tr>
<tr>
<td>8800 GT</td>
<td>6.65</td>
<td>40.4</td>
<td>6.70</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>6.09</td>
<td>44.1</td>
<td>6.23</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>4.39</td>
<td>61.2</td>
<td>4.70</td>
</tr>
</tbody>
</table>

Conventional 3-D FFT algorithm requires three transpose steps (Step 2,4,6) in addition to the computation steps of 1-D FFTs (Step 1,3,5). Table 6 shows the elapsed time and achieved bandwidth in each step. The transpose steps attain very poor memory bandwidth, which is nearly equal to the bandwidth of copying 256 streams.

Our bandwidth intensive 3-D FFT algorithm consists of five CUDA kernels as shown in Section 3.2. Table 7 shows the elapsed time and achieved bandwidth in each step. Since step 3 consists of the same operations as step 1, we do not distinguish them in the table. Also, step 4 is regarded being equivalent to step 2. Although step 1, 3, 5 of conventional algorithm and step 5 of bandwidth intensive algorithm use the same 1-D FFT kernel, there is a little difference in performance, which comes from that the former is out-of-place and the latter is in-place.

For each of step 1 to 4, SPs load input data from device memory, execute a small number of operations for 16-point FFT, and then store the outputs to device memory. Thus the performance of these steps is largely limited by bandwidth of device memory. In these steps, 8800 GTX, which sports the largest memory bandwidth, achieves the best performance.

On the other hand, in step 5, which computes 256-point FFTs, the ratio of floating-point operations to memory access is about twice that of the other steps. Computing a 256-point FFT using multiple threads requires shared memory access. With 64 threads, each thread embodies four data values, and a 256-point FFT requires data exchange via shared memory at least three times. Due to such characteristics, performance of step 5 is more sensitive to the clock frequency of
Table 8. The performance of 65536 sets of 256-point 1-D FFTs

<table>
<thead>
<tr>
<th>Model</th>
<th>Our Implementation</th>
<th>CUFFTID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(ms)</td>
<td>GFLOPS</td>
</tr>
<tr>
<td>8800 GT</td>
<td>5.72</td>
<td>117</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>5.17</td>
<td>130</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>5.52</td>
<td>122</td>
</tr>
</tbody>
</table>

Table 9. The elapsed time (ms) of 3-D FFT of size $256^3$ on 8800 GTS, with shared memory, texture memory, or non-coalesced global memory access

<table>
<thead>
<tr>
<th></th>
<th>X axis</th>
<th>Y&amp;Z axes</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared memory</td>
<td>5.17</td>
<td>24.7</td>
<td>29.9</td>
</tr>
<tr>
<td>Texture memory</td>
<td>5.11 + 8.43</td>
<td>24.7</td>
<td>38.3</td>
</tr>
<tr>
<td>Not coalesced</td>
<td>5.13 + 14.3</td>
<td>24.7</td>
<td>44.2</td>
</tr>
</tbody>
</table>

SPs than the other steps. In practice, we observe that 8800 GTS is faster than 8800 GTX in this step, because its total peak performance of SPs is better than GTX. The GTX still excels in bandwidth but is lower than those in the other steps, indicating shortage of SPs, i.e., compute power. With 8800 GTS and GT, they are largely unchanged, however, so the performance limiting factor is still memory bandwidth with these two cards.

4.2. Detailed Analysis of Fine-Grained Parallelism

Step 5 involves parallel computing of 65536 sets of 256-point 1-D FFTs in sequence. Here, we show elapsed time and speed in GFLOPS in Table 8. (The table also includes performance of 1-D FFT routine of CUFFT, where we observe that our FFT greatly outperforms CUFFT.) As discussed above, although the performance of step 5 with 8800 GTX card is seemingly limited by lack of SPs rather than memory bandwidth, the table shows that the measured GFLOPS in step 5 is only about 30% of its peak floating-point performance. Investigating a cubin file of our implementation, we have found there are many other instructions than FP operations, such as shared memory access. Moreover, many of FP operations are not combined into FMA operation. That wastes half of the FMA units’ capability, and limits the performance of 1-D FFTs.

4.3. Effect of Shared Memory

We also measured the performance improvements by the shared memory of CUDA, by implementing a version of our 3-D FFT without using shared memory. This will be a good metric for future GPU design to determine how much resource need to be diverged to non stream programming features. Table 9 shows the elapsed time of 3-D FFT of size $256^3$ on 8800 GTS. Since the transforms for Y and Z axes do not use the shared memory, therefore the elapsed times of those steps remain unchanged. Without shared memory, we are forced to use global memory for data exchange between threads. For this reason, we cannot use fine-grained parallelism, so the transforms for X axis are also divided into two steps of 16-point FFTs. Although we optimized code so that the accesses to the device memory are coalesced as much as possible, the FFT algorithm fundamentally requires at least one data exchange between threads such that we must either utilize texture memory or non-coalesced memory access for the second step. As a result, we observed that the second step takes longer than the first step. Here, performance using texture memory is much higher than non-coalesced memory access, but using shared memory further outperforms them substantially for step five, and overall we observe more than 25% performance advantage.

4.4. Performance with Data Transfer Overhead

So far, we have discussed on-board performance of our 3-D FFT implementation. However, if an application program uses GPU as an offload engine that simply performs FFT for data on host memory, data transfer is required between host and device for each 3-D FFT. Table 10 shows overhead of data transfer and the overall performance with transfer time being inclusive. We observe that the performance becomes heavily degraded, since the PCI-Express interface is far slower than bandwidth of device memory. In fact, 8800 GTX, which achieves the best on-board performance, is now the slowest card, since it is a product of older generation supporting only PCI-Express 1.1.

Fortunately, not all the applications require frequent data transfer between host and device. If we could integrate other computation into GPU and the core working set largely becomes resident on the card for longer duration, transfer is largely reduced and the performance will be close to full speed on the device. One of such applications we are working on is ZDock [22], which simulates protein-protein docking. By rotating and translating the Ligand protein, the best docking positions are determined by scoring scheme. Its kernel computation is 3-D convolution based on 3-D FFT to calculate scores for all the translations at once. By integrating all such other operations into the GPU,
Table 10. The performance of 3-D FFT of size $256^3$ including the data transfer between host and device

<table>
<thead>
<tr>
<th>Model</th>
<th>Host-to-Device</th>
<th>Device-to-Host</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCI-Express</td>
<td>Time(s)</td>
<td>GFLOPS</td>
</tr>
<tr>
<td>Host-to-Device</td>
<td>Time(ms)</td>
<td>GByte/s</td>
<td></td>
</tr>
<tr>
<td>8800 GT</td>
<td>2.0 x16</td>
<td>25.9</td>
<td>5.18</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>2.0 x16</td>
<td>25.7</td>
<td>5.21</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>1.1 x16</td>
<td>47.6</td>
<td>2.82</td>
</tr>
</tbody>
</table>

Table 11. The performance of 3-D FFT of size $256^3$ in single precision using FFTW library 3.2alpha2 on CPUs

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock</th>
<th>Core</th>
<th>Time(ms)</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Phenom</td>
<td>2.20GHz</td>
<td>4</td>
<td>195</td>
<td>10.3</td>
</tr>
<tr>
<td>Intel Core</td>
<td>2.66GHz</td>
<td>4</td>
<td>188</td>
<td>10.7</td>
</tr>
</tbody>
</table>

data transfer is largely eliminated; the host program only sends input data and receives small data about the best docking positions.

Even in applications where we cannot move all the computation into GPU, we still have chances to reduce data transfer overhead. The latest devices support asynchronous transfers, which enable overlap between data transfer and computation on the device. Thus if the application has other independent tasks, the transfer overhead becomes smaller.

4.5. Comparing with Latest CPUs

Table 11 shows the performance of 3-D FFT of size $256^3$ using the single precision routine of the FFTW library [23] version 3.2alpha2. OpenMP and SSE extension are enabled, and all of four CPU cores are used. Our 3-D FFT on GPU greatly outperforms FFTW on a CPU, even if we include the transfer time.

Since currently available CUDA GPUs support only single precision operations, they are not useful for applications that require higher accuracy. However, GPUs with double precision support are starting to appear. We plan on implementing a double precision version and making comparative analysis as soon as such cards and the appropriate programming support are available.

4.6. Other Problem Sizes

Our 3-D FFT algorithm does not depend on problem size, although the program itself must be tailored for each major sizes. Figure 2 and Figure 3 show the performance of 3-D FFT of size $64^3$ and $128^3$ in addition to $256^3$. Since smaller problem sizes decrease the ratio of floating-point operations to memory accesses, the GFLOPS number becomes smaller, whereas achieved memory bandwidth is almost unchanged, indicating that we are bandwidth-limited. Here again, our 3-D FFT still outperforms the CUFFT library by several factors.

Next, Table 12 shows the performance for larger size, $512^3$. Since it requires larger memory than the capacity of device memory, data transfer between host
Table 12. The performance of 3-D FFT of size $512^3$. Times in second, and PCI-Express interconnect is running in highest mode supported by GPU

<table>
<thead>
<tr>
<th></th>
<th>Step 1</th>
<th>Step 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H-to-D</td>
<td>D-to-H</td>
<td>H-to-D</td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>GB/s</td>
<td>GB/s</td>
<td>GB/s</td>
</tr>
<tr>
<td>8800 GT</td>
<td>0.216</td>
<td>0.360</td>
<td>0.206</td>
</tr>
<tr>
<td></td>
<td>4.96</td>
<td>0.043</td>
<td>5.20</td>
</tr>
<tr>
<td>8800 GTS</td>
<td>0.217</td>
<td>0.287</td>
<td>0.207</td>
</tr>
<tr>
<td></td>
<td>4.95</td>
<td>0.042</td>
<td>5.19</td>
</tr>
<tr>
<td>8800 GTX</td>
<td>0.419</td>
<td>0.224</td>
<td>0.381</td>
</tr>
<tr>
<td></td>
<td>2.56</td>
<td>0.031</td>
<td>2.81</td>
</tr>
<tr>
<td>FFTW</td>
<td>–</td>
<td>0.332</td>
<td>0.339</td>
</tr>
</tbody>
</table>

and device occurs as shown in Section 3.3. Although data transfer occupies a large part of elapsed time, performance is still up to 50% faster than that of FFTW on a quad-core CPU. In addition, CPU cores are idle during computation on GPU and transfers, so they can be utilized for other purposes within or outside the application subject to acceleration.

4.7. Power Efficiency

Table 13 shows the power consumption of the whole system when computing 3-D FFT of size $256^3$ repeatedly. The components of the system are same as listed in Table 5. When using CPU for computation of FFT, we installed NVIDIA RIVA128, an old, low-power GPU, to minimize its power consumption. Compared with the computation on CPU, GPUs have about four times higher power efficiency (GFLOPS/Watt).

5. Conclusion

GPUs provide an extremely higher memory bandwidth in addition to the computation power required for high performance computing. Intuitively, they seemingly enable the acceleration of more complex computations such as FFT in an easy fashion, much as much more expensive vector processors had excelled in these types of problems. Unfortunately, conventional implementations of FFT including vendor-optimized libraries such as CUFFT still do not exploit the performance of GPUs to their full extent in these problems, as they do not sufficiently exploit the special nature of their memory system. We have proposed a 3-D FFT algorithm which is optimized for CUDA GPUs. For efficient access to the device memory, our 3-D FFT algorithm consists of computations with shared memory for transforms along X axis, and multirow FFT algorithm for transforms along Y and Z axes. In performance evaluations, we showed our bandwidth intensive 3-D FFT algorithm outperforms CUFFT library, conventional algorithm using transposes, and FFTW library on CPUs significantly, by analyzing the various factors on how we should parallelize the different steps in the algorithm, the effect of shared memory, how to optimize memory bandwidth, as well as loss in performance by Host-to-card memory transfers. Overall, we found that, with careful programming to maximize memory bandwidth, we can achieve very good performance, up to the point where memory bandwidth is saturated. One should carefully analyze different steps in the algorithm to determine how it should be parallelized, so as to avoid stride memory access, and exploit the memory system optimized for (simpler) stream programs, while being conscious of register resources. Local memory helps to alleviate stride memory access and should be heavily utilized. We believe that such a methodology will be applicable to other “sparse” problems as well, and hope that our methodology will serve a guideline on how to program and optimize such programs on GPUs to attain tremendous performance gains, just as being achieved for more tightly-coupled / stream applications.

Acknowledgment

This research is partially supported by Core Research of Evolutional Science and Technology (CREST) project “ULP-HPC: Ultra Low-Power, High-Performance Computing via Modeling and Optimization of Next Generation HPC Technologies” of Japan Science and Technology Agency (JST), and the Microsoft Technical Computing Initiative project “HPC-GPGPU: Large-Scale Commodity Accelerated Clusters and its Application to Advanced Structural Proteomics”. The Microsoft Research and NVIDIA also gave us good technical advice and information that made our work possible. At last, we would like to thank Dr. Daisuke Takahashi at University of Tsukuba who made valuable discussions with us.

References

Table 13. The power consumption of the whole system

<table>
<thead>
<tr>
<th>GPU</th>
<th>Computation</th>
<th>Power (Idle)</th>
<th>Power (FFT 256^3)</th>
<th>GFLOPS</th>
<th>GFLOPS/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIV A128 On CPU</td>
<td>126 Watt</td>
<td>140 Watt</td>
<td>10.3</td>
<td>0.074</td>
<td></td>
</tr>
<tr>
<td>8800 GT On GPU</td>
<td>180 Watt</td>
<td>215 Watt</td>
<td>62.2</td>
<td>0.289</td>
<td></td>
</tr>
<tr>
<td>8800 GTS On GPU</td>
<td>196 Watt</td>
<td>238 Watt</td>
<td>67.2</td>
<td>0.282</td>
<td></td>
</tr>
<tr>
<td>8800 GTX On GPU</td>
<td>224 Watt</td>
<td>290 Watt</td>
<td>84.4</td>
<td>0.291</td>
<td></td>
</tr>
</tbody>
</table>


