Improving Performance of Matrix Multiplication and FFT on GPU

Xiang Cui, Yifeng Chen, and Hong Mei

Key Laboratory of High Confidence Software Technologies, Ministry of Education
School of Electronics Engineering and Computer Science, Peking University
Beijing, China
cuixiang08@sei.pku.edu.cn, {cyf, meih}@pku.edu.cn

Abstract—In this paper we discuss about our experiences in improving the performance of two key algorithms: the single-precision matrix-matrix multiplication subprogram (SGEMM of BLAS) and single-precision FFT using CUDA. The former is computation-intensive, while the latter is memory bandwidth or communication-intensive. A peak performance of 393 Gflops is achieved on NVIDIA GeForce GTX280 for the former\(^\text{1}\), about 5% faster than the CUBLAS 2.0 library. Better FFT performance results are obtained for a range of dimensions. Some common principles are discussed for the design and implementation of many-core algorithms.

Keywords—CUDA; GPU; matrix multiplication; FFT;

I. INTRODUCTION

Current GPU architecture provides improved programmability for high-performance computing. A GPU device consists of a set of multiprocessors, each with several stream processors. The off-chip global device memory is accessible from all multiprocessors. Each multiprocessor has a much faster but small shared on-chip memory as well as a number of 32-bit registers.

The CUDA programming model\(^\text{3}\) targets application development for such platforms. Its execution model consists of a hierarchy of parallelization layers: grids, thread blocks, warps and threads. Each processor runs a number of threads. A thread block is a batch of threads running on one multiprocessor and therefore all threads in a thread block share some part of the shared memory. A grid consists of several thread blocks. Because there can be more blocks than multiprocessors, different thread blocks of a grid are scheduled for the multiprocessors dynamically. Threads within a block are grouped in warps, each running as an SIMD unit. The threads of a thread block are scheduled to occupy the multiprocessor exclusively each time. More details of CUDA can be found in\(^\text{3}\).

CUDA provides a low entry level for learning many-core programming, but obtaining peak performance can be a very challenging task. A programmer must understand all the constraints of the hardware platforms and devise an algorithm that reaches a right balance for all the performance-affecting factors. For example, the hardware vendor provides two commonly used libraries for programmers:

1) the CUBLAS library for basic linear algebraic computation,
2) and the CUFFT library for FFT.

Unfortunately, some subprograms in earlier versions (CUBLAS 1.0 and CUFFT 1.1)\(^\text{2}\) show poor performances. In this paper, we focus on two crucial subprograms: the computation-intensive matrix-matrix multiplication program and bandwidth/communication-intensive 1D FFT. The huge gap between the vendor’s implementation and the currently known best implementations indicates how hard it would be for a common application developer to write an efficient CUDA program.

Indeed, to get efficient CUDA codes, one must not only understand the GPU architecture but also have practical experiences with the basic test codes that can reach the limits for memory bandwidth, flops and on-chip memory allocation. In this paper, our experiences obtained from such tests will be summarized as several programming principles. Improved performances for SGEMM and FFT are obtained in comparison to the latest efficient implementations\(^\text{7}, \text{6}, \text{3}\). Our solutions are tested on GPU NVIDIA GeForce GTX280 for single-precision floats. We found that drivers of different versions have significantly different performance results, so we present all the test results with corresponding driver version number specified.\(^\text{3}\)

Section 2 discusses about our basic optimization principles. Section 3 and Section 4 study matrix multiplication and 1D FFT with performance results shown and analyzed respectively.

II. BASIC OPTIMIZATION PRINCIPLES

A number of guidelines on improving CUDA program performance can be found in\(^\text{7}, \text{6}, \text{3}\). In this section,

\(^{2}\)CUBLAS 1.0 and CUFFT 1.1 were available when doing this work. Later, CUBLAS 2.0 was released with better performance than 1.0. When this paper is to print, NVIDIA released the CUFFT 2.3 with improved performance of FFT also.

\(^{3}\)When compiled with the latest driver CUDA 2.3 after the completion of this paper, both our codes and CUBLAS 2.0 demonstrate lower performance for unknown reasons.
we discuss about performance optimization with some of our own experiences and understanding.

A. An MIMD View of CUDA

Each multiprocessor runs in an SIMD manner: in each clock cycle, all stream processors of a multiprocessor execute the same instruction but operate on different data [3]. The basic SIMD unit is warp. Each warp has the same number of threads. Current GPUs have a warp size of 32. Warps are formed according to thread numbering in a thread block: the first 32 threads in a block form the first warp and so on. Active warps - i.e. all the warps from all running blocks - are time-sliced: a thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessor’s computational resources.

In most CUDA program kernels, the threads of every block have the same code structure but process different data according to the block id and thread id. Although not recommended, different warps can run different codes according their warp IDs. This is achievable by testing the warp id number in a conditional. As long as the threads within the same warp choose the same conditional branch, there will be a small performance penalty for such testing. This allows a code to run in an MIMD manner and perform different instructions.

Figure 1 illustrates how this kind of MIMD mechanism can reduce the usage of the shared memory in communication and hence allow more blocks to be active at the same time (a requirement for enough active threads to hide pipeline latencies and thread synchronization costs). In FFT, the current threads communicate with each other in a "butterfly manner" for matrix transposition. To achieve so, all threads in four warps store their original data in shared memory and then fetch the needed data back after synchronization, as shown in (A). This transposition operation requires a large amount of shared memory, which will limit the number of active blocks (without any mutual communications) that can be allocated to a multiprocessor at the same time.

The shared memory usage can be reduced, as shown in (B), if this operation is decomposed into two smaller operations: the first two warps first place their data in the shared memory for all threads to fetch; and then the other two warps place their data for fetching after synchronization. According to our experiments, kernels with divergent warp instructions could cause a performance reduction of 10-30%. This, however, proves to be beneficial to our FFT algorithms (on certain problem sizes) as the reduction is easily outweighed by releasing the pipeline-latency penalty from lack of enough active threads due to the size limit of the shared memory (see Section 4).

In this kind of mutual communication among active threads, the shared memory essentially becomes the buffer for communication. The code does need to manage the synchronization for the buffer by itself. The size of the buffer is limited by the physical size of the shared memory divided by the number of active blocks (besides some shared memory taken by the kernel’s arguments).

B. Peak Instruction Throughput

A good way to understand the performance issue of many-core programming is to study how the hardware’s peak performances can be achieved in special test codes, which parts of a given code cause performance reductions compared to these codes, and whether the performance bottlenecks can be modified to conform to the special codes. GeForce GTX280 has 240 cores running at 1.295 GHz. Each core is capable of performing 3 single-precision floating-point operations including a pair of MAD and MUL and a special function. The theoretical peak single-precision performance is 933 Gflops (3 flops × 240 × 1.295 GHz).
The peak instruction throughput can be reached by executing register-to-register MAD instructions \( a=b \times a+b \) and \( b=a \times b+a \) in an aggressively unrolled loop. The corresponding PTX codes are in Table 1, which shows that these are two fused MADs. When fully pipelined, each stream processor can issue one MAD operation per clock cycle, yielding a total \( 2 \times 240 \times 1.295 \text{GHz} = 622 \text{Gflops} \), which is almost the result showed in the first line of Table 1.

### Table I

**TWO CONSECUTIVE INSTRUCTIONS IN AN UNROLLED LOOP THAT TESTS INSTRUCTION THROUGHPUT AND THE CORRESPONDING PTX CODES.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Gflops</th>
<th>PTX codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a=b \times a+b; )</td>
<td>620</td>
<td><code>mad.f32 %f1, %f1, %f2, %f2; mad.f32 %f2, %f1, %f2, %f1;</code></td>
</tr>
<tr>
<td>( b=a \times b+a; )</td>
<td>64</td>
<td><code>mov.f32 %f3, 0f3f828f5c;</code></td>
</tr>
<tr>
<td>( a=b+b+1.01f; )</td>
<td>515</td>
<td><code>mad.f32 %f1, %f1, %f2, %f2; mad.f32 %f2, %f1, %f2, %f1;</code></td>
</tr>
<tr>
<td>( b=a+b+a; )</td>
<td>484</td>
<td><code>mov.f32 %f4, 0f3f83d70a;</code></td>
</tr>
<tr>
<td>( a=b \times c+a; )</td>
<td>515</td>
<td><code>mov.f32 %f3, 0f3f828f5c;</code></td>
</tr>
<tr>
<td>( b=a \times b+a; )</td>
<td>44</td>
<td><code>mad.f32 %f2, %f1, %f2, %f3; mad.f32 %f2, %f1, %f2, %f1;</code></td>
</tr>
<tr>
<td>( a=b \times a+c; )</td>
<td>544</td>
<td><code>mov.f32 %f3, 0f3f828f5c;</code></td>
</tr>
<tr>
<td>( b=a+b+a; )</td>
<td>515</td>
<td><code>mad.f32 %f2, %f1, %f2, %f3; mad.f32 %f2, %f1, %f2, %f1;</code></td>
</tr>
<tr>
<td>( a=b \times a+c+c; )</td>
<td>544</td>
<td><code>mov.f32 %f3, 0f3f828f5c;</code></td>
</tr>
<tr>
<td>( b=a+b+a+1; )</td>
<td>530</td>
<td><code>mad.f32 %f2, %f1, %f2, %f3; mad.f32 %f2, %f1, %f2, %f1;</code></td>
</tr>
</tbody>
</table>

Table 1 shows that whenever an MAD involves more than two registers or contains a constant, extra MOV instructions will be generated with performance significantly reduced.

It is found that if one of the operands is in the shared memory, the MAD instructions run at about 66% of their peak 620 Gflops, i.e. about 410 Gflops [7]; if more than one operand is in the shared memory, extra MOV instructions will be generated. That means the register files should be preferred to the shared memory whenever this is possible in a code.

### C. Pipeline Latency

It is noted in [7] and [3] that instruction pipeline latency is properly hidden as long as there are at least 192 active threads per multiprocessor. Our experiments show that 256 is the minimum number to achieve the peak performance. Table 2 lists different active thread numbers per multiprocessor and their corresponding performances for repeated register-to-register MAD instructions.

### Table II

**DIFFERENT ACTIVE THREAD NUMBER PER MULTIPROCESSOR AND THE CORRESPONDING PERFORMANCE.**

<table>
<thead>
<tr>
<th>Threads/block</th>
<th>128</th>
<th>64</th>
<th>192</th>
<th>64</th>
<th>256</th>
<th>64</th>
<th>128</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks/MP</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Threads/MP</td>
<td>128</td>
<td>128</td>
<td>192</td>
<td>192</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>Gflops</td>
<td>405</td>
<td>405</td>
<td>515</td>
<td>515</td>
<td>620</td>
<td>620</td>
<td>620</td>
<td></td>
</tr>
</tbody>
</table>

When there are 256 or more threads (i.e. the number of threads per block times the number of blocks) per multiprocessor, the peak performance of 620 Gflops is reached. For every instruction is register-to-register calculation, all warps are always active in this experiment.

### D. Number of Blocks, Pipeline Latency and Synchronization

As the issuing order of warps within a block is not specified, shared-memory accesses must be synchronized for data exchange. Adding thread synchronizations into a kernel normally reduces performance, but in some codes, interestingly, adding additional thread synchronizations may improve performance, a phenomenon possibly due to better coalesced device memory accesses (see Section 2.5).

### Table III

**PERFORMANCE COMPARISONS FOR DIFFERENT MAD/SYNCHRONIZATION RATIOS AND BLOCK NUMBERS PER MULTIPROCESSOR.**

<table>
<thead>
<tr>
<th>MAD/synchronization</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>one block per MP (Gflops)</td>
<td>292</td>
<td>398</td>
<td>484</td>
<td>544</td>
</tr>
<tr>
<td>two blocks per MP (Gflops)</td>
<td>324</td>
<td>421</td>
<td>502</td>
<td>551</td>
</tr>
</tbody>
</table>

The first two rows of the Table 3 illustrate different synchronization / multiply-and-add ratios and the corresponding achieved Gflops when register-to-register calculations are performed. The number of active threads is 256 here. In Section 3, the performance of matrix-matrix multiplication is improved by reducing the total number of synchronizations. A synchronization following device memory accesses will force a multiprocessor to idle. If there is more than one active block, then the thread scheduler will automatically execute the other block and may better overlap device-memory access latency with computation. This idea of using more blocks to deal with thread synchronization will also be applied in the optimization of FFT presented in Section 4. Allocating more active blocks to an MP faces the size constraints of the register files and the shared memory, whose precise usage can only be determined by inspecting into the generated files after compilation.

### E. Device Memory Bandwidth

The device memory space is not cached, so it is important to adhere to the right access pattern to achieve the maximum memory bandwidth between the device memory and the processor cores [3]. GPU is capable of reading 32-bit, 64-bit or 128-bit words from the device memory into registers in a single instruction and the device memory addresses simultaneously accessed by each thread of a half-warp during the execution of a single read or write instruction should be arranged so that the memory accesses can be coalesced into a single contiguous aligned memory access.

In order to test the device memory bandwidth, we execute the code in Figure 2 for a large amount of data. The variable b can either be float or float2 for 32-bit access or 64-bit access respectively. The _syncthreads operation in the
Table V
DIFFERENT SAMPLES DISCUSSED IN THIS PAPER.

<table>
<thead>
<tr>
<th>Sample Type</th>
<th>MAD/byte</th>
<th>MAD Type</th>
<th>Intensive Type</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix multiplication</td>
<td>3.8MAD/B</td>
<td>Register-shared memory</td>
<td>computation-intensive</td>
<td>Used</td>
</tr>
<tr>
<td>FFT N=16</td>
<td>1.25MAD/B</td>
<td>Register-register</td>
<td>memory-intensive</td>
<td>Not used</td>
</tr>
<tr>
<td>FFT N=2048</td>
<td>3.44MAD/B</td>
<td>Register-register</td>
<td>memory/communication-intensive</td>
<td>Used</td>
</tr>
<tr>
<td>FFT N=4096</td>
<td>3.75MAD/B</td>
<td>Register-register</td>
<td>memory/communication-intensive</td>
<td>Used</td>
</tr>
</tbody>
</table>

b = A[ blockIdx.x * NUM_THREADS_PER_BLOCK + threadIdx.x ];
b.x = 1.0f;
__syncthreads(); // Tests are made for whether this
// __syncthreads() is present or not.
A[blockIdx.x * NUM_THREADS_PER_BLOCK + threadIdx.x] = b;

Figure 2. The code that tests device memory bandwidth by reading and writing contiguous aligned memory.

Table IV
DEVICE MEMORY BANDWIDTH RESULTS (IN GBYTES/SEC) FOR DIFFERENT THREAD NUMBERS PER THREAD BLOCK WHEN SYNCHRONIZATION IS EITHER PRESENT OR ABSENT.

<table>
<thead>
<tr>
<th>Threads/Block</th>
<th>32-bit without sync</th>
<th>32-bit with sync</th>
<th>64-bit without sync</th>
<th>64-bit with sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>57.1</td>
<td>114.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>102.3</td>
<td>123.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>119.1</td>
<td>121.6</td>
<td>124.5</td>
<td>124.8</td>
</tr>
<tr>
<td>256</td>
<td>117.5</td>
<td>122.1</td>
<td>123.9</td>
<td>124.5</td>
</tr>
</tbody>
</table>

code may either be present or left out. Table 4 lists the performance results.

According to the experiment results, coalesced 64-bit accesses deliver a little higher bandwidth than coalesced 32-bit accesses: the maximum bandwidth of 64-bit access and 32-bit access are 124.8 GB/s and 121.6 GB/s respectively on GTX280 when each thread block contains 128 threads (noting that __syncthreads improves performance slightly, see Section 2.6). This small difference does affect the overall Gflops in most bandwidth-bound examples (see Section 3).

F. Device-memory Transaction

The device-memory accesses of all active threads are organized into memory transactions [3]. Although there is little detailed information about how these memory transactions are organized, our experiments do show that sometimes adding thread synchronizations into a code improves the overall performance, as repeated memory accesses (with non-deterministic latencies) can be better coalesced if the threads are periodically synchronized. In Section 4, the performance of 1D FFT when N=16 is improved using this principle.

G. Steps to Achieve High Performance

Many-core programming can be easy to start with but very hard to obtain near peak performance. This is evident from the fact that some subprograms from the vendor’s own original libraries CUBLAS 1.0 and CUDA 1.1 perform rather poorly compared to more recently published codes.

There are many performance-critical factors to consider, address and balance. We thus list the general steps that programmers could follow to obtain high performance:

Step 1: try to design an algorithm that exposes as much data parallelism as possible and decompose the problem into tasks of small sizes that can fit into the constraints of GPU architecture.

Step 2: calculate the ratio of computation and device memory access to determine whether it is computation-intensive or memory-bandwidth-intensive. With GTX280, the peak register-register MAD performance is 620 Gflops and register-shared memory MAD performance is 410 Gflops, while the device memory bandwidth is 120GB/s. Thus the lines between computation-intensive and memory-intensive roughly lie at 2.6 MAD/byte and 1.7 MAD/byte respectively. If the algorithm is memory-intensive, the device-memory accesses should follow the principles in Section 2.5 and 2.6. If it is computation-intensive or communication-intensive, the shared memory must be handled carefully. For example, matrix-matrix multiplication uses shared memory to hold the input data to reduce device-memory access, while in 1D FFT of size 1024 threads within a block use the shared memory to exchange intermediate results after computation. Different samples discussed in this paper are listed in Table 5.

Step 3: try to use thread synchronization to coordinate warp execution and improve the efficiency of device-memory transactions, according to Section 2.6.

These steps should be repeated for performance optimization. For example, the register usage of a CUDA program is often subject to unpredictable compiler optimization and can only be known precisely after compilation. The allocation of registers and shared memory directly affects the number of active blocks, which is strongly linked to performance.

III. MATRIX MULTIPLICATION (SGEMM)

The above programming principles are applied to single-precision matrix multiplication (the SGEMM operation of BLAS). This problem is computation-bound but can be memory-bandwidth-bound if not programmed properly.

The task of computing the product C of two matrices A and B can be split among multiple thread blocks as follows: each thread block computes one subblock Csub for C, and each thread within the thread block computes several elements of Csub. Csub is the product of two rectangular
matrices: the sub-matrix of A that has the same row range of Csub, and the sub-matrix of B that has the same column range of Csub (see Figure 3). To meet GPU’s constraints, these two rectangular matrices are divided into subblocks (called Asub and Bsub) and Csub is computed as the sum of the products of these subblocks. These subblocks in A and B don’t have to be square. The area size of Csub determines the number of times that each element must be loaded to the GPU chip. If Csub is too small, the computation-bound SGEMM may become memory-bound.

Suppose Asub, Bsub and Csub to be \( m \times k \), \( k \times n \) and \( m \times n \), and A, B and C are partitioned to \( M \times K \), \( K \times N \) and \( M \times N \) of these subblocks. Then computing one Csub requires fetching K blocks of Asub and Bsub from A and B. Because there are \( M \times N \) Csubs in C, these operations sum up to \( M \times N \times K \times m \times n + M \times N \times K \times k \times n = M \times m \times N \times n \times K \times k \times (1/m+1/n) \) floats from the device memory.

In the following implementation, A, B and C are assumed to be \( \text{MATRIX WIDTH} \times \text{MATRIX WIDTH} \) square matrices and the Csub is set to be \( 16 \times 256 \). The total MADs are \( \text{MATRIX WIDTH}^3 \), and the total bytes read from device memory are \( (\text{MATRIX WIDTH}^3) \times (1/16+1/256) \times 4 \), so the MAD/byte ratio is about 3.8 and this is a computation-intensive problem.

A. Implementation of Matrix Multiplication

The algorithm is implemented for row-major arrays. The thread number is 128 and organized in a \( 4 \times 32 \) setup. Csub is set to be \( 16 \times 256 \). Each thread block computes one subblock Csub, and each thread computes two columns of Csub. The flow chart of each thread is outlined in Figure 4.

First the shared memory is allocated to store \( 16 \times 32 \) subblock of A and registers to store 16 float2s i.e. 32 floats of two columns in result Csub for every thread. After computing pointers in A, B and C using thread ID (Note that threads in one thread block are organized in two-dimensional \( 4 \times 32 \)), there is a loop. In each round, one thread block fetch \( 16 \times 32 \) elements from A and then another inner loop is executed in which each thread reads one float2 from B, multiplies these two floats with corresponding operands coming from shared memory, and add the results to registers where the two columns of result Csub are stored.

Figure 3. Matrix multiplication: \( A \times B = C \) with all arrays in row major.

Thread block with 128 threads for each C subblock:

1. Initiate C subblock (16x2 per thread)
2. Repeat for all A subblocks with the same rows:
   1. Load an A subblock (16x32)
   2. Synchronization
3. Repeat for 32 rows of the next B subblock
   1. Load the next row of B subblock (float2 per thread)
   2. Update C subblock (2x16 per thread) from A and B subblocks
   3. Synchronization
4. Write back C subblock

Figure 4. Thread block function for matrix-matrix multiplication.

Figure 5. Comparison with other implementations of matrix multiplication. These performances were obtained using GTX280 with driver version 178.28. We also tested our code on GTX285 and got an impressive performance of 445 Gflops.
Our code
16
31.5KB
2.2KB
Gflops
3.8
64-bit
2.4
2.5
256MB
1.1KB
375
8388608
128
64
MAD/B
2.4
applied
16
Principle
7.5KB
30
16
32-bit
4
32
3.2
16
63
16

ized into three categories: when the signal size \(N=8\) or \(16\),
complex FFT whose different signal sizes can be character-
comparison are listed in Table 6.

is significantly reduced (see Section 2.4). Details of this
in [7]. Consequently the total number of synchronizations
\(\times\) block loads a subblock of \(16\times 64\)
the peak device-memory bandwidth. Finally, each thread
from device memory each round. This is known to achieve
128 threads and each thread reads a float2 i.e. a 64-bit word
nade memory back to device memory.

Another example that we consider is one-dimensional
real or the imaginary part, and hence only one thread block
of share memory is needed for transposition of either the
array transpositions between them via the shared memory.
To transpose 2048 float pairs of one signal in shared memory
each thread can get one signal of 16 coefficients while
each warps read the device memory in a coalesced manner.

no data exchange between threads is needed, all calculation
of one signal can be performed in registers by each thread
without accessing the shared memory; when \(N=64\), 256, 512,
1024, 2048 or 4096, data exchange is performed through the
shared memory; when \(N\) is more than 8192, device memory
is needed to transpose matrices. This usually is done by
calling kernels twice.

A common metric [1] estimates that the number of floating-
point operations required by a radix-2 FFT is \(5N\log_2 N\). Thus,
the MAD/byte ratio of 1D FFT is \( (5N\log_2 N)/(2 \cdot 8 \cdot N) = (5\log_2 N)/16\), as each coefficient is a complex
consisting of two floats. The MAD/byte ratios for \(N=16\),
2048 and 4096 are listed in Table 5. It’s memory-intensive
for \(N=16\). Moreover, as discussed before, when \(N=2048\) and
4096, threads need to communicate via shared memory, they
are communication-intensive.

The cases of \(N=16\), 2048 and 4096 are picked out to
illustrate our optimization ideas.

A. 1D FFT of Signal Size 16

When \(N=16\), it’s obviously a bandwidth-bound problem.
So in order to get the peak performance, we have to focus
on the device memory access efficiency according to Section
2.7. To achieve coalesced memory accesses, the original
data is laid out in host memory in a certain order so that
each thread can get one signal of 16 coefficients while
each warp reads the device memory in a coalesced manner.
This restriction can be easily eliminated by performing two
transpositions in shared memory and is not our concern in
this paper. So in this implementation, all that one thread
needs to do is to read 16 coefficients from device memory
to registers, to calculate and to write the result back to device
memory. For each coefficient is a complex, it’s naturally 64-
bit access mode which we discussed early in Section 2.5.
Moreover, according to Section 2.6, we make each thread
block consist 256 threads and add thread synchronization be-
tween calculation and writing results back to device memory
to optimize memory transaction organization towards getting
peak device memory access efficiency and correspondingly
peak performance.

B. 1D FFT of Signal Size 2048 and 4096

When \(N=2048\), the thread number of each block is set
to be 128 and two active blocks are allocated for each
multiprocessor. Here, each thread block computes one signal
of 2048 complex float pairs, and each thread computes 16,
16 and 8 coefficients in registers during three steps with two
array transpositions between them via the shared memory.
To transpose 2048 float pairs of one signal in shared memory
(without considering bank conflict), at least \(2048 \times 4=8\)KB
of share memory is needed for transposition of either the
real or the imaginary part, and hence only one thread block
can be active on one multiprocessor.

<table>
<thead>
<tr>
<th></th>
<th>CUBLAS 2.0</th>
<th>Our code</th>
<th>Principle applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>C’s block</td>
<td>16×64</td>
<td>16×256</td>
<td></td>
</tr>
<tr>
<td>A’s block</td>
<td>16×16</td>
<td>16×32</td>
<td>2.4</td>
</tr>
<tr>
<td>B’s block</td>
<td>16×64</td>
<td>32×256</td>
<td></td>
</tr>
<tr>
<td>Threads/block</td>
<td>64</td>
<td>128</td>
<td>2.5</td>
</tr>
<tr>
<td>Scalar registers per thread</td>
<td>30</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>Registers per block</td>
<td>7.5KB</td>
<td>31.5KB</td>
<td></td>
</tr>
<tr>
<td>Smem per block</td>
<td>1.1KB</td>
<td>2.2KB</td>
<td></td>
</tr>
<tr>
<td>Blocks per MP</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Word read from device memory</td>
<td>32-bit</td>
<td>64-bit</td>
<td>2.5</td>
</tr>
<tr>
<td>Total data amount read from device memory of A and B</td>
<td>160GB</td>
<td>136GB</td>
<td></td>
</tr>
<tr>
<td>Total data amount written to device memory of C</td>
<td>256MB</td>
<td>256MB</td>
<td></td>
</tr>
<tr>
<td>Total MAD</td>
<td>8192×3</td>
<td>8192×3</td>
<td></td>
</tr>
<tr>
<td>MAD/byte ratio</td>
<td>3.2</td>
<td>MAD/B</td>
<td>3.8</td>
</tr>
<tr>
<td>Total number of synchronization</td>
<td>67108864</td>
<td>8388608</td>
<td>2.4</td>
</tr>
<tr>
<td>Performance achieved</td>
<td>375</td>
<td>Gflops</td>
<td>393 Gflops</td>
</tr>
</tbody>
</table>

Finally, each thread writes the final results of two continuous
columns in Csub back to device memory.

B. Experimental Results and Analysis

Figure 5 compares the Gflops performance of our code
with CUBLAS 1.0 and CUBLAS 2.0. When MATRIX_WIDTH=8192,
which is the largest problem size that can be placed entirely in device memory, our code reaches 393
Gflops, about 5% higher than the 375 Gflops of [7] (now
a subprogram of CUBLAS 2.0). The theoretical limit for
CUDA codes that use shared memory in every MAD is 410
Gflops.

Our implementation shares a similar code structure as that
of [7]. The following differences have contributed to our
better performance: firstly, our Csub has a size of \(16\times 256\),
instead of \(16\times 64\). This reduces data transfer by 15%. Also,
according to Section 2.5, the thread block size consists of
128 threads and each thread reads a float2 i.e. a 64-bit word
from device memory each round. This is known to achieve
the peak device-memory bandwidth. Finally, each thread
block loads a subblock of \(16\times 32\) of A, instead of \(16\times 16\)
in [7]. Consequently the total number of synchronizations
is significantly reduced (see Section 2.4). Details of this
comparison are listed in Table 6.

IV. IMPLEMENTATION AND ANALYSIS OF 1D FFT

Another example that we consider is one-dimensional
complex FFT whose different signal sizes can be character-
ized into three categories: when the signal size \(N=8\) or \(16\),
At least two blocks are needed to hide synchronization latencies. Our solution is to halve the usage of the shared memory by partitioning the transposition operation into two steps: in the first step only half of the threads place their data into the shared memory and let all threads read, and in the second step, the other half of the threads perform this operation. As a result which we already showed in Figure 1, each thread block consumes about a little more than half of 8KB i.e. 4KB shared memory, and thus two thread blocks can be active simultaneously on one multiprocessor. In this implementation, different warps in one thread block behave differently according to their warp IDs (refer to the optimization principle mentioned in Section 2.1). When N=4096, the implementation is similar but there is only one active thread block.

Figure 6. Comparison with other FFT implementations. These performance figures were obtained using GTX280 with driver version 177.26. Note that the cycle in the graph means one implementation must arrange the data layout in host memory before and/or after CUDA kernel invocation and the time consumed is not included in this performance comparison.

C. Experimental Results and Analysis

Forward FFTs are tested for signal batches of total size 64 MB. For example, signal size 1024 corresponds to a batch of 8192, while signal size 2048 corresponds to a batch of 4096 and so on.

Figure 6 compares the performance of our codes with that of NVIDIA’s CUDA FFT library (CUFFT) version 1.1 and that of Volkov and Kazian [9], [8] (which to be included in OpenCL FFT). Also note that in our implementation, when N=2048 and 4096, the FFT is processed by one kernel call and correspondingly no data shuffling rearrangement is required, while some existing codes [8] require the input data to be rearranged by CPU after the kernel calls (which cost was not included in the results).

V. Conclusions

Writing efficient many-core codes is hard. Our work is not entirely intended to show improved algorithms and implementation for SGEMM and FFT. Instead our focus is to study the guidelines that can assist common programmers to obtain high performance. The programming considerations are often both top-down for overall resources consumption and bottom-up for designing special codes that achieve the peak performances. The approaches of these two directions meet in the middle.

The points raised in this paper should be viewed in addition to the advices from the CUDA programming guide and more recent research [7] (for a more complete picture).

The reason that such programming guidelines are important is because these libraries need to be modified for various real applications as well as different hardware systems. If the hardware is upgraded in future, for example by doubling the register files, some previously inefficient algorithms may become efficient and even reach top speed. With more registers, we may completely avoid the shared memory and still meet the requirement for memory bandwidth. The codes that run the fastest on certain platforms are insignificant themselves, but the experiences obtained from such endeavor are useful and can be applied to future applications and new platforms.

In the race for Gflops, we have improved the performance of SGEMM as well as several problem sizes of FFT using the programming guidelines.

REFERENCES