High Performance Discrete Fourier Transforms on Graphics Processors

Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli
Microsoft Corporation
{nagag,dalloyd,yurido,burtons,jmanfer}@microsoft.com

***Abstract*—We present novel algorithms for computing discrete Fourier transforms with high performance on GPUs. We present hierarchical, mixed radix FFT algorithms for both power-of-two and non-power-of-two sizes. Our hierarchical FFT algorithms efficiently exploit shared memory on GPUs using a Stockham formulation. We reduce the memory transpose overheads in hierarchical algorithms by combining the transposes into a block-based multi-FFT algorithm. For non-power-of-two sizes, we use a combination of mixed radix FFTs of small primes and Bluestein's algorithm. We use modular arithmetic in Bluestein's algorithm to improve the accuracy. We implemented our algorithms using the NVIDIA CUDA API and compared their performance with NVIDIA's CUFFT library and an optimized CPU-implementation (Intel's MKL) on a high-end quad-core CPU. On an NVIDIA GPU, we obtained performance of up to 300 GFlops, with typical performance improvements of 2–4× over CUFFT and 8–40× improvement over MKL for large sizes.**

I. Introduction

The Fast Fourier Transform (FFT) refers to a class of algorithms for efficiently computing the Discrete Fourier Transform (DFT). The FFT is used in many different fields such as physics, astronomy, engineering, applied mathematics, cryptography, and computational finance. Some of its many and varied applications include solving PDEs in computational fluid dynamics, digital signal processing, and multiplying large polynomials. Because of its importance, the FFT is used in several benchmarks for parallel computers such as the HPC challenge [1] and NAS parallel benchmarks [2]. In this paper we present algorithms for computing FFTs with high performance on graphics processing units (GPUs).

The GPU is an attractive target for computation because of its high performance and low cost. For example, a $300 GPU can deliver peak theoretical performance of over 1 TFlop/s and peak theoretical bandwidth of over 100 GiB/s. Owens et al. [3] provides a survey of algorithms using GPUs for general purpose computing. Typically, general purpose algorithms for the GPU had to be mapped to the programming model provided by graphics APIs. Recently, however, alternative APIs have been provided that expose low-level hardware features that can be exploited to provide significant performance gains [4], [5], [6], [7]. In this paper we target NVIDIA's CUDA API, though many of the concepts have broader application.

Main Results: We present algorithms used in our library for computing FFTs over a wide range of sizes. For smaller sizes we compute the FFT entirely in fast, shared memory. For larger sizes, we use either a global memory algorithm or a hierarchical algorithm, depending on the size of the FFTs and the performance characteristics of the GPU. We support non-power-of-two sizes using a mixed radix FFT for small primes and Bluestein's algorithm for large primes. We address important performance issues such as memory bank conflicts and memory access coalescing. We also address an accuracy issue in Bluestein's algorithm that arises when using single-precision arithmetic. We perform comparisons with NVIDIA's CUFFT library and Intel's Math Kernel Library (MKL) on a high end PC. On data residing in GPU memory, our library achieves up to 300 GFlops at factory core clock settings, and overclocking we achieve 340 GFlops. We obtain typical performance improvements of 2–4× over CUFFT and 8–40× over MKL for large sizes. We also obtain significant improvements in numerical accuracy over CUFFT.

The rest of the paper is organized as follows. After discussing related work in Section II we present an overview of mapping FFT computation to the GPU in Section III. We then present our algorithms in Section IV and implementation details in Section V. We compare results with other FFT implementation in Section VI and then conclude with some ideas for future work.

II. Related Work

A large body of research exists on FFT algorithms and their implementations on various architectures. Sorensen and Burrus compiled a database of over 3400 entries on efficient algorithms for the FFT [8]. We refer the reader to the book by Van Loan [9] which provides a matrix framework for understanding many of the algorithmic variations of the FFT. The book also touches on many important implementation issues.

The research most related to our work involves accelerating FFT computation by using commodity hardware such as GPUs or Cell processors. Most implementations of the FFTs on the GPU use graphics APIs such as current versions of OpenGL or DirectX [10], [11], [12], [13], [14], [15]. However, these APIs do not directly support scatters, access to shared memory, or fine-grained synchronization available on modern GPUs. Access to these features is currently provided only by vendor-specific APIs. NVIDIA's FFT library, CUFFT [16], uses the CUDA API [5] to achieve higher performance than is possible with graphics APIs. Concurrent work by Volkov and Kazian [17] discusses the implementation of FFT with CUDA. We also use CUDA for FFTs, but we handle a much wider range of input sizes and dimensions.

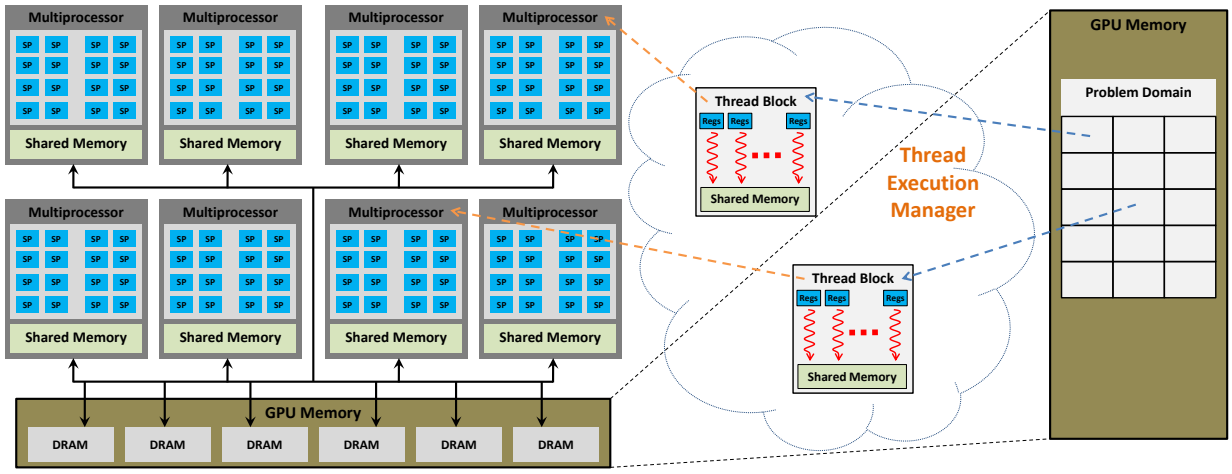

Fig. 1. Architecture and programming model on the NVIDIA GeForce 8800 GPU. On the left, we illustrate a high-level diagram of the GPU scalar processors and memory hierarchy. This GPU has 128 scalar processors and 80 GiB/s peak memory bandwidth. On the right, we illustrate the programming model for scheduling computation on GPUs. The data in the GPU memory is decomposed into independent thread blocks and scheduled on the multiprocessors.

Several researchers have examined the implementation of the FFT on the Cell processor [18], [19], [20], [21], [22]. Our results for large sizes on commodity GPUs are generally higher than published results for the Cell for large sizes.

III. Overview of GPUs and FFTs

A. Overview of GPUs

In this paper we focus primarily on NVIDIA GPUs, although many of the principles and techniques extend to other architectures as well. Fig. 1 highlights the hardware model of a NVIDIA GeForce 8800 GPU. The GPU consists of a large number of scalar, in-order processors that can execute the same program in parallel using threads. Scalar processors are grouped together into multiprocessors. Each multiprocessor has several fine-grain hardware thread contexts, and at any given moment, a group of threads called a *warp*, executes on the multiprocessor in lock-step. When several warps are scheduled on a multiprocessor, memory latencies and pipeline stalls are hidden primarily by switching to another warp. Each multiprocessor has a large register file. During execution, the program registers are allocated to the threads scheduled on a multiprocessor. Each multiprocessor also has a small amount of shared memory that can be used for communication between threads executing on the scalar processors. The GPU memory hierarchy is designed for high bandwidth to the global memory that is visible to all multiprocessors. The shared memory has low latency and is organized into several banks to provide higher bandwidth.

At a high-level, computation on the GPU proceeds as follows. The user allocates memory on the GPU, copies the data to the GPU, specifies a program that executes on the multiprocessors and after execution, copies the data back to the host. In order to execute the program on a domain, the user decomposes the domain into blocks. The thread execution manager then assigns threads to operate on the blocks and write the output to global memory.

B. Overview of FFTs

The forward Discrete Fourier Transforms (DFT) of a complex sequence $x = x_0, \ldots, x_{N-1}$ is an N-point complex sequence, $X = X_0, \ldots, X_{N-1}$, where $X_k = \sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N}$. The inverse DFT is similarly defined as $x_k = \frac{1}{N} \sum_{j=0}^{N-1} X_j e^{2\pi ijk/N}$. A naïve implementation of DFTs requires $O(N^2)$ operations and can be expensive. FFT algorithms compute the DFT in $O(N \log N)$ operations. Due to the lower number of floating point computations per element, the FFT can also have higher accuracy than a naïve DFT. A detailed overview of FFT algorithms can found in Van Loan [9]. In this paper, we focus on FFT algorithms for complex data of arbitrary size in GPU memory.

C. Mapping FFTs to GPUs

Performance of FFT algorithms can depend heavily on the design of the memory subsystem and how well it is exploited. Although GPUs provide a high degree of memory parallelism, the index-shuffling stage (also referred to as bit-reversal for radix-2) of FFT algorithms such as Cooley-Tukey can be quite expensive due to incoherent memory accesses. In this paper, we avoid the index-shuffling stage using Stockham formulations of the FFT. This, however, requires that we perform the FFT out-of-place. Fig. 2 shows pseudo-code for a Stockham radix-R FFT with specialization for radix-2. In each iteration, the algorithm can be thought of combining the R FFTs on subsequences of length N_s into the FFT of a new sequences of length RN_s by performing an FFT of length R on the corresponding elements of the subsequences.

The performance of traditional GPGPU implementations of FFT using graphics APIs is limited by the lack of *scatter* operations, that is, a thread cannot write to an arbitrary location in memory. The pseudo-code shown in Fig. 2 writes to R different locations each iteration (line 29). Without scatter, R values must be read for each output generated rather than

reading R values for every R outputs [14]. GPUs and APIs that support writing multiple values to the same location in multiple buffers can save the redundant reads, but must either use more complex indexing when accessing the values written in a preceding iteration, or after each iteration, they must copy the values to their proper location in a separate pass [15], which consumes bandwidth. Thus scatter is important for conserving memory bandwidth.

Fig. 2 also shows pseudo-code for an implementation of the FFT on a GPU which supports scatter. The main difference between `GPU_FFT()` and `CPU_FFT()` is that the index j into the data is generated as a function of the thread number t, the block index b, and the number of threads per block T (line 13). Also, the iteration over values of N_s are generated by multiple invocations of `GPU_FFT()` rather than in a loop (line 3) because a global synchronization between threads is needed between the iterations, and for many GPUs the only global synchronization is kernel termination.

For each invocation of `GPU_FFT()`, T is set to N/R and the number of thread blocks B is set to M, where M is the number of FFTs to process simultaneously. Processing multiple FFTs at the same time is important because the number of warps used for small-sized FFTs may not be sufficient to achieve full utilization of the multiprocessor or to hide memory latency while accessing global memory. Processing more than one FFT results in more warps and alleviates these problems.

Despite the fact that `GPU_FFT()` uses scatter, it still has a number of performance issues. First, the writes to memory have coalescing issues. The memory subsystem tries to coalesce memory accesses from multiple threads into a smaller number of accesses to larger blocks of memory. But the space between consecutive accesses generated during first few iterations (small N_s) is too large for coalescing to be effective (line 29). Second, the algorithm does not exploit low-latency shared memory to improve data reuse. This is also a problem for traditional GPGPU implementations as well, because the graphics APIs do not provide access to shared memory. Finally, to handle arbitrary lengths, we would need to write a separate specialization for all possible radices R. This is impractical, especially for large R. In the next section we will discuss how we address each of these issues.

Because GPUs vary in shared memory sizes, memory, and processor configurations, the FFT algorithms should ideally be parameterized and auto-tuned across different algorithm variants and architectures.

IV. FFT Algorithms

In this section, we present several FFT algorithms — a global memory algorithm that works well for larger FFTs with higher radices on architectures with high memory bandwidth, a shared memory algorithm for smaller FFTs, a hierarchical FFT that exploits shared memory by decomposing large FFTs into a sequence of smaller ones, mixed-radix FFTs that handle sizes that are multiples of small prime factors, and an implementation of Bluestein's algorithm for handling larger prime factors.

```
1  float2* CPU_FFT(int N, int R,
2                  float2* data0, float2* data1) {
3    for( int Ns=1; Ns<N; Ns*=R ) {
4      for( int j=0; j<N/R; j++ )
5        FftIteration( j, N, R, Ns, data0, data1 );
6      swap( data0, data1 );
7    }
8    return data0;
9  }
10
11 void GPU_FFT(int N, int R, int Ns,
12               float2* dataI, float2* dataO) {
13   int j = b*N + t;
14   FftIteration( j, N, R, Ns, dataI, dataO );
15 }
16
17 void FftIteration(int j, int N, int R, int Ns,
18                   float2* data0, float2*data1){
19   float2 v[R];
20   int idxS = j;
21   float angle = -2*M_PI*(j%Ns)/(Ns*R);
22   for( int r=0; r<R; r++ ) {
23     v[r] = data0[idxS+r*N/R];
24     v[r] *= (cos(r*angle), sin(r*angle));
25   }
26   FFT<R>( v );
27   int idxD = expand(j,Ns,R);
28   for( int r=0; r<R; r++ )
29     data1[idxD+r*Ns] = v[r];
30 }
31
32 void FFT<2>( float2* v ) {
33   float2 v0 = v[0];
34   v[0] = v0 + v[1];
35   v[1] = v0 - v[1];
36 }
37
38 int expand(int idxL, int N1, int N2 ){
39   return (idxL/N1)*N1*N2 + (idxL%N1);
40 }
```

Fig. 2. Reference implementation of the radix-R Stockham algorithm. Each iteration over the data combines R subarray of length N_s into arrays of length RN_s. The iterations stop when the entire array of length N is obtained. The data is read from memory and scaled by so-called *twiddle factors* (lines 20–25), combined using an R-point FFT (line 26), and written back out to memory (lines 27–29). The number of threads used for `GPU_FFT()`, T, is N/R. The `expand()` function can be thought of as inserting a dimension of length N_2 after the first dimension of length N_1 in a linearized index.

We also discuss extensions to handle multi-dimensional FFTs, real FFTs, and discrete cosine transforms (DCTs).

A. Global Memory FFT

As mentioned in Section III.B, the pseudo-code for `GPU_FFT()` in Fig. 2 can lead to poor memory access coalescing, which reduces performance. On some GPUs the rules for memory access coalescing are quite stringent. Memory accesses to global memory are coalesced for groups of CW threads at a time, where CW is the coalescing width. CW is 16 for recent NVIDIA GPUs. Coalescing is performed when each thread in the group access either a 32-bit, 64-bit, or 128 bit word in sequential order and the address of the first thread is aligned to ($CW\times$ word size). Bandwidth for non-coalesced accesses is about an order of magnitude slower. Later GPUs have more relaxed coalescing requirements. Coalescing is performed for any access pattern, so long as all the threads access the same word size. The hardware issues memory transactions in blocks of 32, 64, or 128 bytes while seeking to

```
1   void exchange( float2* v, int R, int stride,
2                  int idxD, int incD,
3                  int idxS, int incS ){
4     float* sr = shared, *si = shared+T*R;
5     __syncthreads();
6     for( int r=0, ; r<R; r++ ) {
7       int i = (idxD + r*incD)*stride;
8       (sr[i], si[i]) = v[r];
9     }
10    __syncthreads();
11    for( r=0; r<R; r++ ) {
12      int i = (incS + r*incS)*stride;
13      v[r] = (sr[i], si[i]);
14    }
15  }
```

Fig. 3. Function for exchanging the R values in `v` between T threads. The real and imaginary components of `v` are stored in separate arrays to avoid bank-conflicts. The second synchronization avoids read-after-write data hazards. The first synchronization is necessary to avoid data hazards only when `exchange()` is invoked multiple times.

minimize the number and size of the transactions to satisfy the requests. For both sets of coalescing requirements, the greatest bandwidth is achieved when the accesses are contiguous and properly aligned.

Assuming that the number of threads per block $T = N/R$ is no less than CW, our mapping of threads to elements in the Stockham formulation ensures that the reads from global memory are in contiguous segments of at least CW in length (line 23 in Fig. 2). If the radix R is a power of two, the reads are also properly aligned. Writes are not contiguous for the first $\lceil \log_R CW \rceil$ iterations where $N_s < CW$ (line 29), although under the assumption that $T \geq CW$, when all the writes have completed, the memory areas touched do contain contiguous segments of sufficient length. Therefore, we handle this problem by first exchanging data between threads using shared memory so that it can then be written out in larger contiguous segments to global memory. We do this by replacing lines 28–29 with the following:

```
int idxD = (t/Ns)*R + (t%Ns);
exchange( v, R, 1, idxD,Ns,  t,T );
idxD = b*T*R + t;
for( int r=0; r<R; r++ )
  data1[idxD+r*T] = v[r];
```

The pseudo-code for `exchange()` can be found in Fig. 3.

To maximize the reuse of data read from global memory and to reduce the total number of iterations, it is best to use a radix R that is as large as possible. However, the size of R is limited by the number of registers and the size of the shared memory on the multiprocessors. Reducing the number of threads reduces the total number of registers and the amount of shared memory used, but with too few threads there are not enough warps to hide memory latency. We have found that using $T = \max(\lceil 64 \rceil_{R^i}, N/R)$ produces good results, where $\lceil x \rceil_{R^i}$ represents the smallest power of R not less than x.

Bank conflicts: Shared memory on current GPUs is organized into 16 banks with 32-bit words distributed round-robin between them. Accesses to shared memory are serviced for groups of 16 threads at a time (half-warps). If any of the threads in a half-warp access the same memory bank

```
1  template<int R> void
2  FftShMem(int sign, int N, float2* data){
3    float2 v[R];
4    int idxG = b*N + t;
5    for( int r=0; r<R; r++ )
6      v[r] = data[idxG + r*T];
7    if( T == N/R )
8      DoFft( v, R, N, t );
9    else {
10     int idx = expand(t.v,N/R,R);
11     exchange(v,R,1, idx,N/R, t,T );
12     DoFft( v, R, N, t );
13     exchange(v,R,1, t,T, idx,N/R );
14   }
15   float s = (sign < 1) ? 1 : 1/N;
16   for( int r=0; r<R; r++ )
17     data[idxG + r*T] = s*v[r];
18 }
19
20 void DoFft(float2* v, int R, int N,
21            int j, int stride=1) {
22   for( int Ns=1; Ns<N; Ns*=R ){
23     float angle = sign*2*M_PI*(j%Ns)/(Ns*R);
24     for( int r=0; r<R; r++ )
25       v[r] *= (cos(r*angle), sin(r*angle));
26     FFT<R>( v );
27     int idxD = expand(j,Ns,R);
28     int idxS = expand(j,N/R,R);
29     exchange( v,R,stride, idxD,Ns, idxS,N/R );
30   }
31 }
```

Fig. 4. Pseudo-code for shared memory radix-R FFT. This kernel is used when N is small enough that the entire FFT can be performed using just shared memory and registers.

at the same time, a conflict occurs, and the simultaneous accesses must be serialized, which degrades performance. In order to avoid bank conflicts, `exchange()` writes the real and imaginary components to separate arrays with stride 1 instead of a single array of `float2`. When a `float2` is written to shared memory, the two components are written separately with stride 2, resulting in bank conflicts. The call to `exchange()` still results in bank conflicts when R is a power of two and $N_s < 16$. The solution is to pad with N_s empty values between every 16 values. For $R = 2$ the extra cost of computing the padded indexes actually outweighs the benefit of avoiding bank conflicts, but for radix-4 and radix-8, the net gain is significant. Padding requires extra shared memory. To reduce the amount of shared memory by a factor of 2, it is possible to exchange only one component at a time. This requires 3 synchronizations instead of 1, but can result in a net gain in performance because it allows more in-flight threads. When R is odd, padding is not necessary because R is relatively prime w.r.t. the number of banks.

B. Shared Memory FFT

For small N, we can perform the entire FFT using only shared memory and registers without writing intermediate results back to global memory. This can result in substantial performance improvements. The pseudo-code for our shared memory kernel is shown in Fig. 4. As with the global memory FFT, we set the number of threads T to $T = \max(\lceil 64 \rceil_{R^i}, N/R)$. These lower bounds on the thread count also ensure that when the data is read from global memory (lines 4–6), it will be read in contiguous segments

```
1   template<int R> void
2   FftShMemCol(int sign, int N, int strideO,
3               float2* dataI, float2* dataO ){
4     float2 v[R];
5     int strideI = B.x*T.x;
6     int idxI = (((b.y*N+t.y)*B.x+b.x)*T.x)+t.x;
7     int incI = T.y*strideI;
8     for( int r=0; r<R; r++ )
9       v[r] = data[idxI + r*incI];
10    DoFft( x, R, N, t.y, T.x );
11    if( strideO < strideI ) {
12      int i = t.y,  j = (idxI%strideI)/strideO;
13      angle = sign*2*M_PI*j/(N*strideI/strideO);
14      for( int r=0; r<R; r++ ) {
15        v[r] *= (cos(i*angle),sin(i*angle));
16        i += T.y;
17      }
18    }
19    int incO = T.y*strideO;
20    int idxO = b.y*R*incI+expand(idxI%incI,incO,R);
21    if( strideO == 1 ) {
22      int idxD = t.x*N + t.y;
23      int idxS = t.y*T.x + t.x;
24      incO = T.y*T.x;
25      idxO = (b.y*B.x+b.x)*N + idxS;
26      exchange( v,R,1, idxD,T.y, idxS,incO );
27    }
28    float s = (sign < 1) ? 1 : 1/N;
29    for( int r=0; r<R; r++ )
30      data[idxO + r*incO] = s*v[r];
31  }
```

Fig. 5. Pseudo-code for shared memory radix-R FFT along columns used with the hierarchical FFT. `strideI` and `strideO` are the strides of sequence elements on input and output. The kernel is invoked with T_x set to a multiple of R not smaller than CW, $T_y = N/R$, and $B = (\texttt{strideI}/T_x, M/\texttt{strideI})$. The twiddle stage (lines 11–18) and the transposes (lines 19–27) of the hierarchical algorithm are also included in the kernel.

greater than CW in length. However, when $T > N/R$, the data must first be exchanged between threads. In this case, the kernel computes more than one FFT at a time and the number of thread blocks used are reduced accordingly. The data is then restored to its original order to produce large contiguous segments when written back to global memory. When $T = N/R$, no data exchange is required after reading from global memory. Because the data is always written back to the same location from which it was read, the FFT can be performed in-place. As mentioned previously, bank conflicts that occur when R is a power of two can be handled with appropriate padding.

The large number of registers available on NVIDIA GPUs relative to the size of shared memory can be exploited to increase performance. Because the data held by each thread can be stored entirely in registers (in the array `v`), the FFT in each iteration (line 26) can be computed without reading or writing data to memory, and is therefore faster. Shared memory is used only to exchange data between registers of different threads. If the number of registers were smaller, then the data would have to reside primarily in shared memory. Additional memory might be required for intermediate results. In particular, the Stockham formulation would require at least twice the amount of shared memory due to the fact that it is performed out-of-place. Larger memory requirements reduce the maximum N that can be handled.

C. Hierarchical FFT

The shared memory FFT is fast but limited in the sizes it can handle. The hierarchical FFT computes the FFT of a large sequence by combining FFTs of subsequences that are small enough to be handled in shared memory. This is analogous to how the shared memory algorithm computes an FFT of length N by utilizing multiple FFTs of length R that are performed entirely in registers. Suppose we have an array A of length $N = N_\alpha N_\beta$. We first consider a variation of the standard "four-step" hierarchical FFT algorithm [23]:

1) Treating A as $N_\alpha \times N_\beta$ array (row-major), perform N_α FFTs of size N_β along the columns.
2) Multiply each element A_{ij} in the array with twiddle factors $\omega = e^{\pm 2\pi ij/N}$ ($-$ for a forward transform, $+$ for the inverse).
3) Perform N_2 FFTs of size N_α along the rows.
4) Transpose A from $N_\alpha \times N_\beta$ to $N_\beta \times N_\alpha$

N_β is chosen to be small enough that the FFT can be performed in shared memory. If N_α is too large to fit into shared memory, then the algorithm recurses, treating each row of length N_α as an $N_{\alpha\alpha} \times N_{\alpha\beta}$ array, etc. One way to think about this algorithm is that it wraps the original one dimensional array of length N into multiple dimensions, each small enough that the FFT can be performed in shared memory along that dimension. The dimensions are then transformed from highest to lowest. The effect of the multiple transposes that occur when coming out of the recursion is to reverse the order of the dimensions, which is analogous to bit-reversal. The original "four-step" algorithm swaps steps 3 and 4. The end result is the same, except that FFTs are always performed along columns. For example, suppose A is partitioned wrapped into a 3D array with dimensions (N_1, N_2, N_3). The execution of the original and the modified algorithms can be depicted as follows:

$$
\begin{array}{ll}
(\underline{N_1, N_2}, N_3') & (\underline{N_1, N_2}, N_3') \\
(\underline{N_1}, N_2', N_3) & (N_3, \underline{N_1}, N_2') \\
(N_1', N_2, N_3) & (N_3, N_2, N_1') \\
(N_3, N_2, N_1) &
\end{array}
$$

where $'$ indicates an FFT transformation along the specified dimension. The i index in step 2 corresponds an element's index in the transformed dimension (N_α) and the j index corresponds to the concatenation of the indices in the underlined dimensions (N_β). The original algorithm (left) performs all of the FFTs in-place and uses a series of transposes at the end to reverse the order of the dimensions. The entire algorithm can be performed in-place if the transposes are performed in-place. In-place algorithms can be important for large data sizes. In the modified algorithm, the FFT computation always takes place in the current highest dimension and the transposes are interleaved with the computation. This is analogous to the data shuffling in a Stockham formulation of a radix-2 FFT used to avoid bit-reversals.

To reduce the number of passes over the data, we use the modified algorithm and perform the FFT, the twiddle, and

the transpose all in the same kernel. Pseudo-code is shown in Fig. 5. This version of the FFT assumes that `strideI`, the stride between elements in a sequence (the product of the dimensions preceding the one transformed), is greater than 1 and that product of all the dimensions is a power of R. The data accesses to global memory for a single FFT along a dimension greater than 1 are not contiguous. To obtain contiguous accesses, we transform a block of M_b sequences at the same time, where M_b is a power of R no smaller than CW. One side benefit of this is that when R is a power of two, padding is no longer required to avoid bank conflicts in `exchange()` because $M_b = CW = 16$ is the same as the number of banks. However, performing such a large number of FFTs simultaneously means that the N must be partitioned in dimensions of shorter length due to limits on the number of registers and the size of shared memory.

Cases where the strides of sequence elements on input and output, `strideI` or `strideO`, are less than M_b require special handling. When `strideI` $\geq M_b$ and `strideO` $= 1$, we rearrange the data in shared memory so that it can be written out in large contiguous segments (lines 22–26). `strideI` can be 1 only if the preceding dimensions have the trivial length of 1, in which case the FFT can be computed with `FftShMem()` from Fig. 4. For all other cases, specialized code is required to handle the reading and writing of partial blocks. An alternative is to first transpose the high dimension to dimension 1, perform the FFT with a variant of `FftShMem()` that includes the twiddle from step 2, and then transpose from dimension 1 to the final destination. However, these transposes require separate passes over the data and may sacrifice some performance.

Because global memory FFT algorithm does not involve global transposes of the data, it can actually be faster than the hierarchical FFT for large N on GPUs with high memory bandwidth. We use auto-tuning to determine at which point to transition from the hierarchical FFT to the global memory FFT.

D. Mixed-radix FFT

So far we have considered algorithms for radix-R algorithms for which $N = R^i$. To handle mixed-radix lengths $N = R_0^a R_1^b$, the value used for R can be varied in the iterations of a radix-R algorithm. For example, for $N = 2^a 3^b$, we can run a iterations with $R = 2$ and b iterations with $R = 3$ using either the global or shared memory FFTs. If 2^a and 3^b are small enough to fit in shared memory, but N is too large, then we can perform the computation hierarchically by setting $N_\alpha = 2^a$ and $N_\beta = 3^b$. Specializations of `FFT<R>()` can be manually optimized for small primes. When N is a composite of large primes, we use Bluestein's FFT.

E. Bluestein's FFT

The Bluestein's FFT algorithm computes the DFT of arbitrary length N by expressing it as a convolution of two

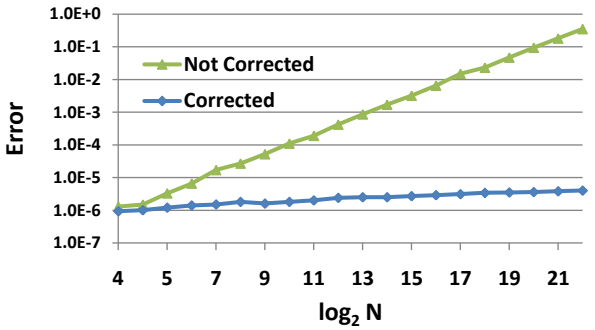

Fig. 6. Comparison of numerical accuracy of Bluestein's FFT algorithm with and without correction.

subsequences a and b:

$$X_k = b_k^* \sum_{j=0}^{N-1} a_j b_{k-j}$$

where $b_j = e^{\frac{\pi i j^2}{N}}$, $a_j = x_j \cdot b_j^*$, and the * operator represents conjugation. The convolution can be computed efficiently as the inverse FFT of $A \cdot B$, where A and B are FFTs of a and b, respectively. The FFTs can be of any length not smaller than $2N - 1$. For example, an optimized radix-R FFT can be used. In order to improve the performance for small sequences, we perform the entire convolution in shared memory using an algorithm similar to `FftShMem()`.

When N is large, care must be taken to avoid problems with numerical accuracy. In particular, a problem arises in the computation of b_j. Because $e^{2\pi i x}$ is periodic, we can rewrite b_j as $e^{2\pi i \frac{j^2}{2N}} = e^{2\pi i f} = e^{2\pi i \text{frac}(f)}$, where $f = j^2/(2N)$ and $\text{frac}(f) = f - \lfloor f \rfloor$. From this we can see that b_j will be inaccurate when f is so large that few, if any, bits are used for its fractional component. To overcome this issue we refine f by discarding large integer components. We compute an $f' = \mathbf{rm}/(2N)$, where $\mathbf{rm} = j^2 \mod 2N$. We assume that $N \in [0, 2^{30})$, which would require over 2^{35}B, or 32GiB, to compute the DFT with a power-of-two FFT (2 buffers with 2^{31} elements for A and B with 8 bytes per element), well above the memory capacities of current GPUs (typically 0.5-1GiB). We start with an estimation of $\mathbf{rm}$ as follows:

$$\mathbf{rm} \approx j^2 - 2N\lfloor f \rfloor,$$

where f is calculated using 32-bit floating point arithmetic. Let $j^2 = a_h 2^{32} + a_l$ and $2N\lfloor f \rfloor = b_h 2^{32} + b_l$, where a_h, a_l, b_h, and b_l are all unsigned 32-bit integers. We compute the lower 32 bits of the multiplications, a_l and b_l, using standard unsigned multiplication. To obtain the upper 32 bits, a_h and b_h, we use an intrinsic `umulhi()`. We then compute f' using modular arithmetic:

$$f' = \text{frac}\left((a_h - b_h)\frac{2^{32} \bmod 2N}{2N}\right) + \frac{(a_l - b_l) \bmod 2N}{2N}.$$

This process produces a value of f' with much improved precision that results in higher accuracy (see Fig. 6). This process can be generalized to support larger N if desired.

GPU	Core Clock (MHz)	Shader Clock (MHz)	Multi-processors	Peak Performance (GFlops)	Memory Clock (MHz)	Memory (MB)	Bus Width (bits)	Peak Bandwidth (GiB/s)	Driver
8800 GTX	575	1350	16	518	900	768	384	80	175.19
8800 GTS	675	1625	16	624	970	512	256	59	175.19
GTX280	650	1300	30	936	1150	1024	512	137	177.41

Fig. 7. **GPUs used in experiments.** Each multiprocessor can theoretical perform 24 floating point operations (8 FMAD/MUL) per shader clock. The GPUs use GDDR3 RAM capable of two memory operations per clock. The warp width for all the GPUs is 32. For our performance results we used the driver versions listed here, unless otherwise specified.

F. Multi-dimensional FFTs

Multi-dimensional FFTs can be implemented by performing FFTs independently along each dimension. However, performance tends to degrade for higher dimensions where the stride between sequence elements is large. This can sometimes be overcome by first transposing the data to move the highest dimension down to the lowest dimension before performing the FFT. This process can be repeated to cycle through all the dimensions. By using a kernel like `FftShMemCol` that combines the FFT with a transpose, separate transpose passes over the data can be avoided.

G. Real FFTs and DCTs

FFTs of real sequences have special symmetry. This symmetry can be used to transform a real FFT into a complex FFT of half the size. Similarly, trigonmetric transforms, such as the discrete cosine transform (DCT) can be implemented with complex FFTs through simple transformation on the data. We implement real FFTs and DCTs with wrapper functions around the FFT algorithms that we have presented in this section. We refer the reader to Van Loan [9] for more details.

V. Implementation

We implemented our FFT library using NVIDIA's CUDA API for single-precision data. We have implemented global memory and shared memory FFT kernels for radices 2, 4, and 8. We use radix-8 for as many iterations as possible. When N is not divisible by 8, we use radix-2 or radix-4 for the last iteration. We have also implemented radix-3 and radix-5 for shared memory.

We use a number of standard optimization techniques that are not presented in the pseudo-code for the sake of clarity. The most important optimization is constant propagation. We use templates to implement specialized kernels for a number of different sizes and thread counts. Where possible we also use bit-wise operations to implement integer multiply, divide, and modulus for power-of-two radices. We also compute some values common to all threads in a block using a single thread and store them in shared memory in order to reduce some computation.

Current GPUs limit the maximum number of threads per thread block and thread blocks per computation grid. On the current GPUs, these limits are 512 and 65535 respectively. These limits restrict the input sizes that can handled. We overcome these limits by virtualizing. Thread indices are virtualized by adding loops in the kernels so that a single thread does the work of multiple virtual threads. Thread blocks are virtualized by invoking the kernel multiple times and adding an appropriate offset to the thread block index for each invocation. Virtualization adds some overhead and code complexity. Supporting it directly in the runtime would enable easier programming on GPUs.

When the size of the FFT is too large for shared memory, we use either the global memory or the hierarchical algorithm. On all of the GPUs we tested, the performance of the hierarchical algorithm degrades for larger N while the performance of the global memory algorithm is nearly constant. At some point there is a cross-over where the global memory algorithm becomes faster. We determine the cross over point at runtime and use the fastest algorithm for a given size.

VI. Results

A. Experimental methodology

We tested our algorithms on three different NVIDIA GPUs: 8800GTX, 8800GTS, and GTX280. The specifications for these GPUs are summarized in Fig. 7. One of the key difference between the GPUs is the memory bandwidth. The GTX280 has the most bandwidth and the 8800GTS has the least. The GTX280 also has more multiprocessors, which give it the highest peak performance. We used recent versions of the drivers. We found, however, that an older version of the driver for the GTX280 (177.11) gave significantly different performance results. Results obtained with this driver are marked with $(*)$ in Figs. 11, 12, and 14. We ran our experiments on a high-end Windows PC equipped with an Intel QX9650 3.0GHz quad-core processor and 4GB of DDR3 1600 RAM. This processor consists of two dual-core dies in the same package with each pair of cores sharing a 6 MB L2 cache.

We compared our algorithms to NVIDIA's CUDA FFT library (CUFFT) version 1.1 for the GPU and Intel's Math Kernel Library (MKL) version 10.2 on the CPU. The MKL tests utilized four hardware threads and used out-of-place, single precision transforms. The input and output arrays were aligned to a multiple of the cache line width. We report performance in GFlops, which we compute as

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

where D is the total number of dimensions, $M_d = E/N_d$ is the number of FFTs along each dimension, and E is the total number of data elements. We follow common convention and

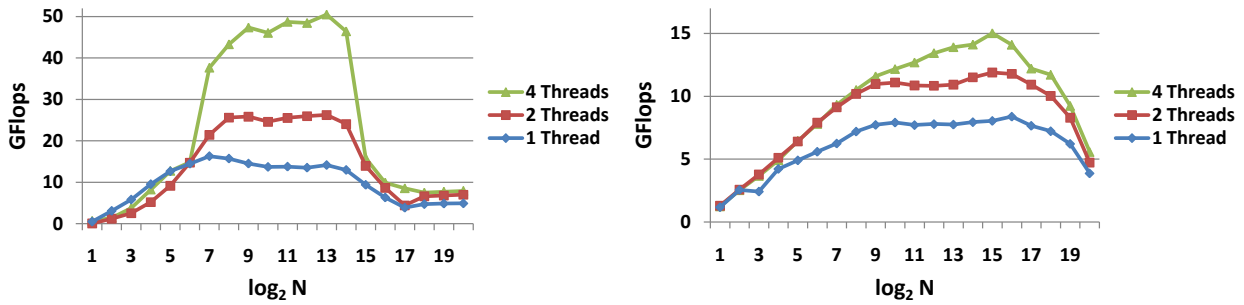

Fig. 8. **MKL with varying numbers of threads.** (Left) Single 1D FFT per thread (M = thread count). Because we use the minimum time over repeated runs on the same data, when the data can fit in the cache, the cache may be hot for these runs. Performance increases with the number of iterations in the FFT algorithm ($\log_2 N$ for radix-2) because of increased reuse of data in the cache. Performance peaks between $N = 2^{10}$ and $N = 2^{17}$ at 52 GFlops. Because pairs of cores share a 6MB L2 cache, performance begins to degrade at about $N = 2^{18}$ due to increased conflicts between cores in the cache. From $N = 2^{20}$ on, the size of the data ($2^{20} \times 2$ (input and output) $\times 2$ (real and imaginary components) $\times 4$ (bytes per float) $= 2^{24}$ bytes) exceeds the 12 MB aggregate L2 cache size of the processor and the performance becomes I/O limited. (Right) Varying number of FFTs with $M = E/N$, where $E = 2^{24}$. The performance with 4 threads is essentially the same as for 2 threads, except for between $N = 2^{10}$ and $N = 2^{17}$ where there is sufficient data reuse without conflicts between cores in the shared caches.

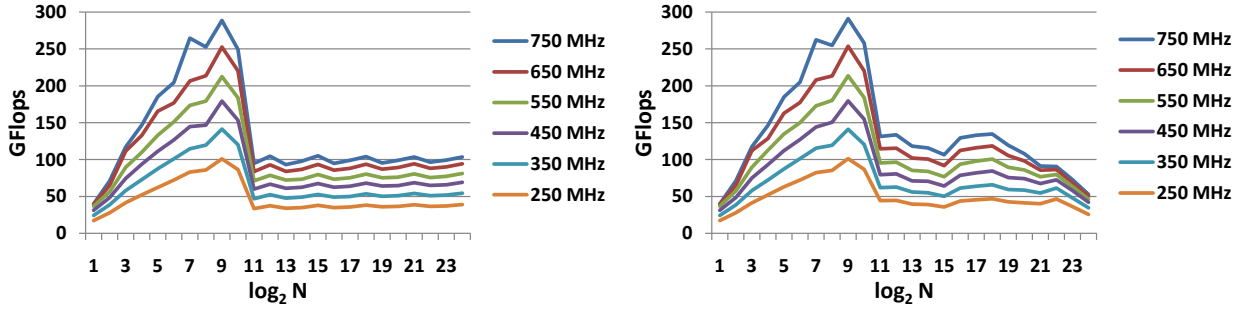

Fig. 9. **Varying core clock rate on GTX280.** The FFTs are performed in shared memory for $N \in [2, 2^{10}]$. For $N > 2^{10}$ we show the performance of the global memory algorithm (left) and the hierarchical algorithm (right). The global memory algorithm shows small oscillations due to use of radix-2 and radix-4 for the last iteration. The performance of the hierarchical algorithm drops off as N increases. For all but the smallest sizes, performance scales linearly with clock rate.

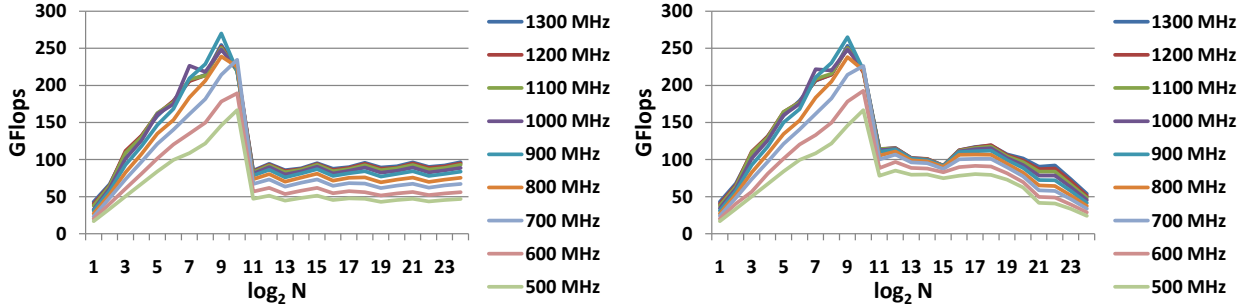

Fig. 10. **Varying memory clock rate on the GTX280.** The FFTs are performed in shared memory for $N \in [2, 2^{10}]$. For $N > 2^{10}$ we show the performance of the global memory algorithm (left) and the hierarchical algorithm (right). The FFT becomes compute bound for higher memory clock rates, especially for larger sizes in the shared memory kernel.

use the same equation for all the algorithms, regardless of radix. The execution time is obtained by taking the minimum time over multiple runs. The time for library configuration and transfers of data to/from the GPU is not included in the timings. Unless stated otherwise, performance reported for the GPU algorithms were obtained on the GTX280. To measure accuracy, we perform a forward transform followed by an inverse transform on uniform random data. We then compare the result to the original input and divide the root mean squared error (RMSE) and maximum error by 2.

B. Scaling

We first examine the scaling properties of MKL w.r.t. the number of threads and the scaling of our algorithms with respect to core and memory clock rates on various GPUs. MKL parallelizes the computation of multiple FFTs by assigning a thread to each FFT. Fig. 8 shows the performance of MKL for a varying number of threads. MKL performs very well for a small number of small FFTs (small M and N), but for large FFTs the performance becomes I/O bound. Performance also degrades for large numbers of FFTs even if N is small.

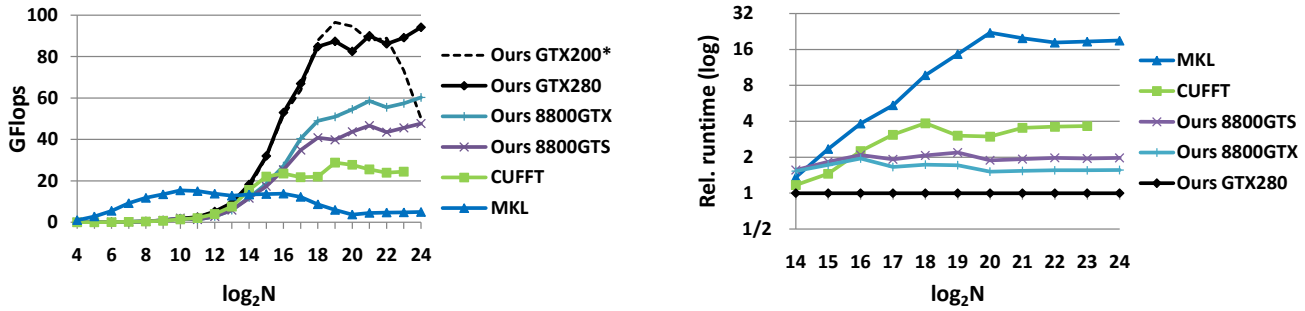

Fig. 11. **Single 1D power-of-two FFTs.** (Left) Performance of our algorithms on multiple GPUs, CUFFT on the GTX280, and MKL. The dashed line is for performance on an older driver. (Right) Run time relative to our algorithms on GTX280 (zoomed on large values of N). MKL shows lower performance because it uses only one thread for single FFTs. The performance of the GPU algorithms is low for small N due to relatively large latencies. For larger N, the GPU algorithms perform much better than MKL on the CPU.

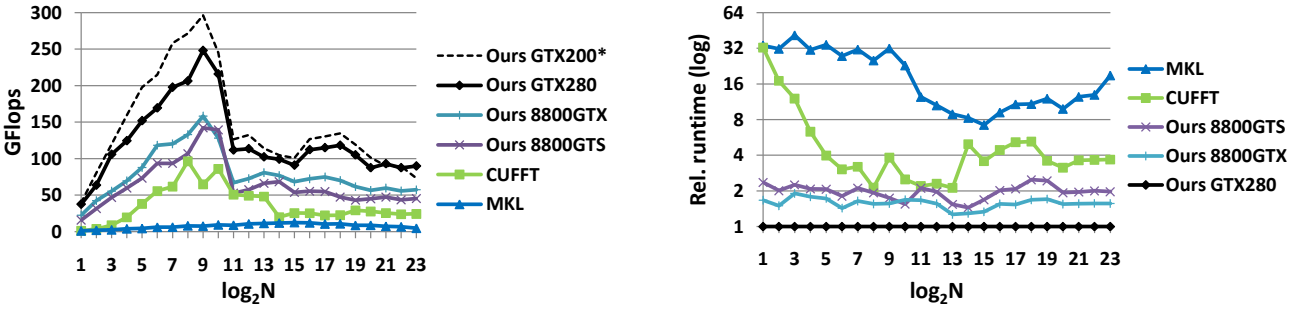

Fig. 12. **Batched 1D power-of-two FFTs.** (Left) Performance of our algorithms on multiple GPUs, CUFFT on the GTX280, and MKL. The dashed line is for performance on an older driver. (Right) Run time relative to our algorithms on GTX280 (zoomed on large values of N). The number of FFTs M is chosen as E/N, where $E = 2^{23}$, the largest value supported by CUFFT. For large N on the GTX280, our FFTs are up to 4 times faster than CUFFT and 19 times faster than MKL.

Fig. 9 and Figure 10 shows the performance of our 1D FFTs on the GTX280 at varying core and memory clock rates, respectively, for both the global and hierarchical algorithm. Both algorithms scale linearly with the core clock, while the scaling for the memory clock is less than linear for higher rates, especially for the shared memory kernels for $N \in [2^7, 2^{10}]$. This indicates that the kernels become compute bound for these higher memory clock rates.

C. Comparisons

Fig. 11 shows the performance for single 1D power-of-two FFTs of varying size. The performance on both the GPU and the CPU is lower for a single FFT than for batched FFTs. Multiple FFTs are needed to utilize all threads with MKL on the CPU and to hide memory latencies for small N on the GPU. For this reason, the rest of our results were obtained by using batched FFTs where total elements E is large and the number of FFTs in the batch, M, is E/N. Batched FFTs are also used for higher dimensional FFTs. Fig. 12 shows the performance of batched 1D power-of-two FFTs. Here the performance on the GPU for small N is much better. For large N, our FFTs are up to 4 times faster than CUFFT and 19 times faster than MKL. Fig. 13 shows a comparison of our 1D shared memory, power-of-two FFT with the cases that are handled by the implementation of Volkov and Kazian [17].

Fig. 14 shows performance for 2D FFTs. For 2D, the performance of our library for large N is up to 3 times faster than CUFFT and 61 times faster than MKL.

We also compared performance for non-power-of-two FFTs. Fig. 15 shows the performance for prime factor FFTs. We currently support powers of 2, 3, and 5. Fig. 18 shows the relative performance of these kernels. The performance for radix For larger primes we use Bluestein's algorithm. We can infer from Fig. 16 that MKL also uses Bluestein's for larger primes. CUFFT, however, uses a direct computation of the DFT which has $O(N^2)$ complexity and has poor accuracy. For large prime sizes, our FFTs achieve up to 11 times speedup over MKL.

Fig. 17 highlights the accuracy of the FFT algorithms. In general, MKL has lower error than the GPU algorithms. The error is the lowest for all algorithms for 1D power-of-two FFTs. Here the errors for the GPU algorithms are quite similar.

Implementation\ Data size	8	16	64	256	512	1024
Volkov and Kazian 08	102	124	229	222	298	260
Ours	120	160	215	271	297	245

Fig. 13. **Comparison with the cases handled by the FFT implementation of Volkov and Kazian [17]** These performance numbers were obtained using a GTX280 with driver version 177.11. The numbers are comparable.

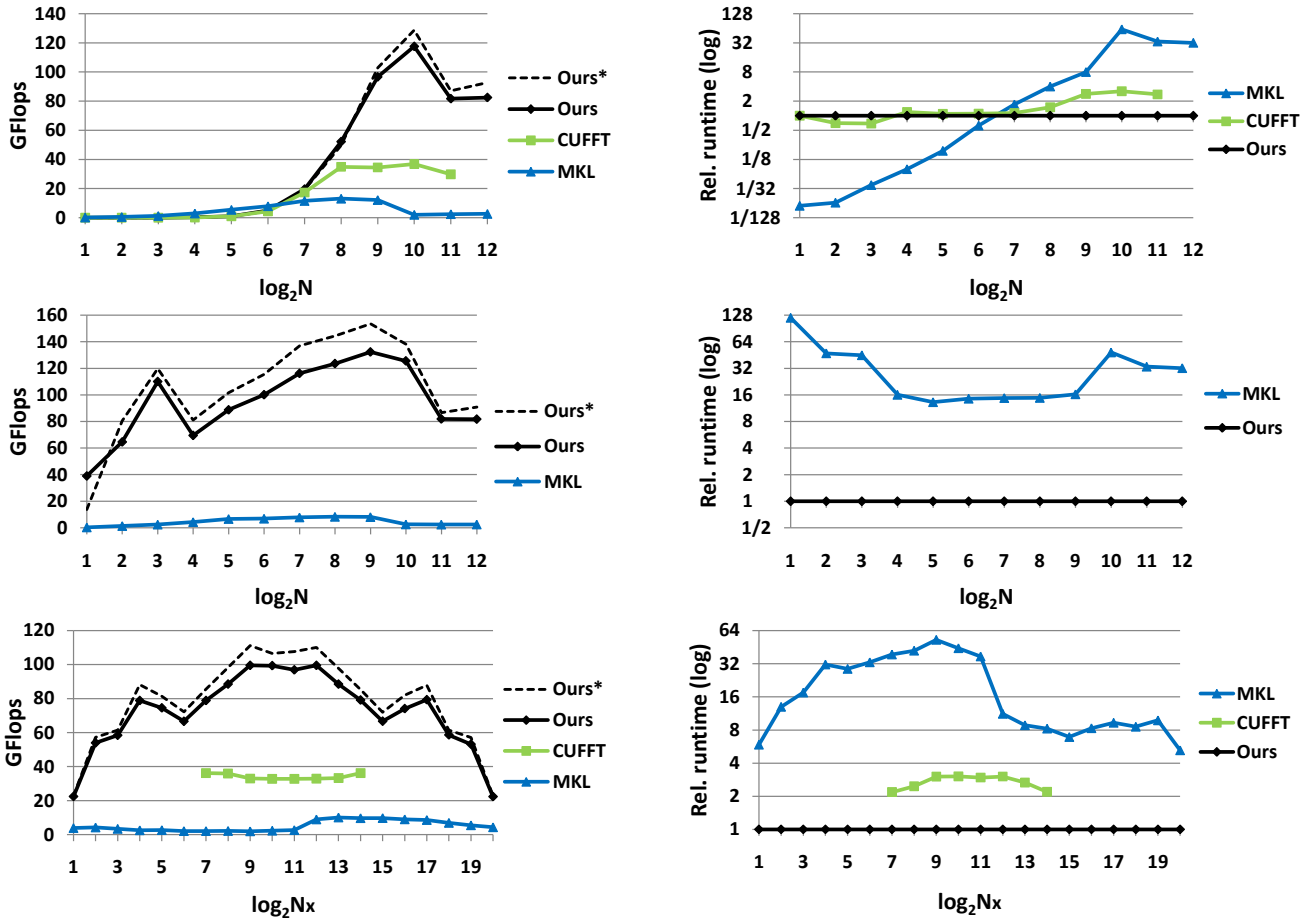

Fig. 14. **2D power-of-two FFTs.** (Top) Performance for single 2D FFTs of size $N \times N$. (Middle) Performance for M 2D FFTs of size $N \times N$, where $M = E/N^2$ and $E = 2^{24}$. CUFFT not shown because it does not support batched 2D FFTs. (Bottom) Performance of a single 2D FFT of size $N_x \times N_y$ where $N_x N_y = 2^{24}$. CUFFT currently only supports 2D FFTs with $N_x, N_y \in [2, 2^{14}]$. The dashed line is for performance on an older driver.

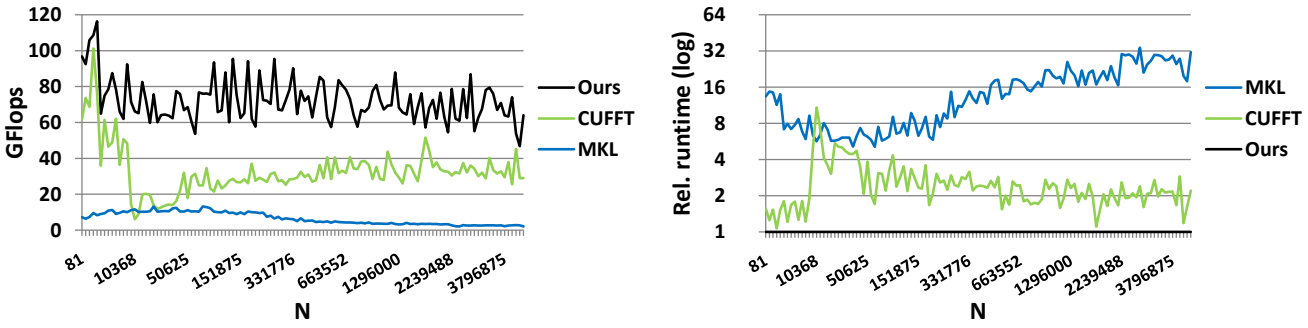

Fig. 15. **1D Mixed-radix FFTs.** (Left) Performance for batched 1D FFTs where $N = 2^i 3^j 5^k$, $i, j, k \neq 0$, $M = E/N$, and $E \approx 2^{24}$. (Right) Run time relative to our algorithms. For large N, our FFTs are typically 2–8 times faster than CUFFT and 5–32 times faster than MKL.

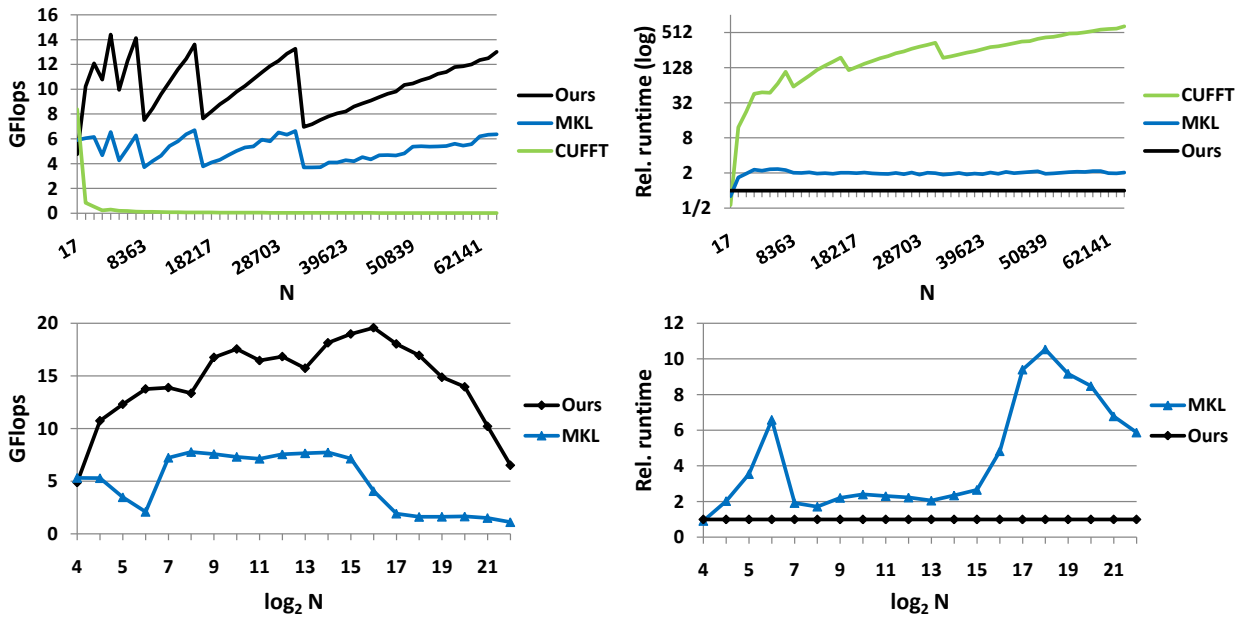

Fig. 16. **1D Prime FFTs.** (Top) Performance for batched 1D FFTs, where $N \in [2^5, 2^{16}]$ is prime. The saw tooth shape of the plot for our FFTs and MKL's is characteristic of the Bluestein algorithm. CUFFT uses an $O(N^2)$ algorithm which is very slow for large N. (Bottom) Batched 1D FFTs where N is the largest prime not greater than 2^i for $i \in [1, 24]$ and $M = E/N$, where $E \approx 2^{24}$. For large prime N, our FFTs are up to 10 times faster than MKL.

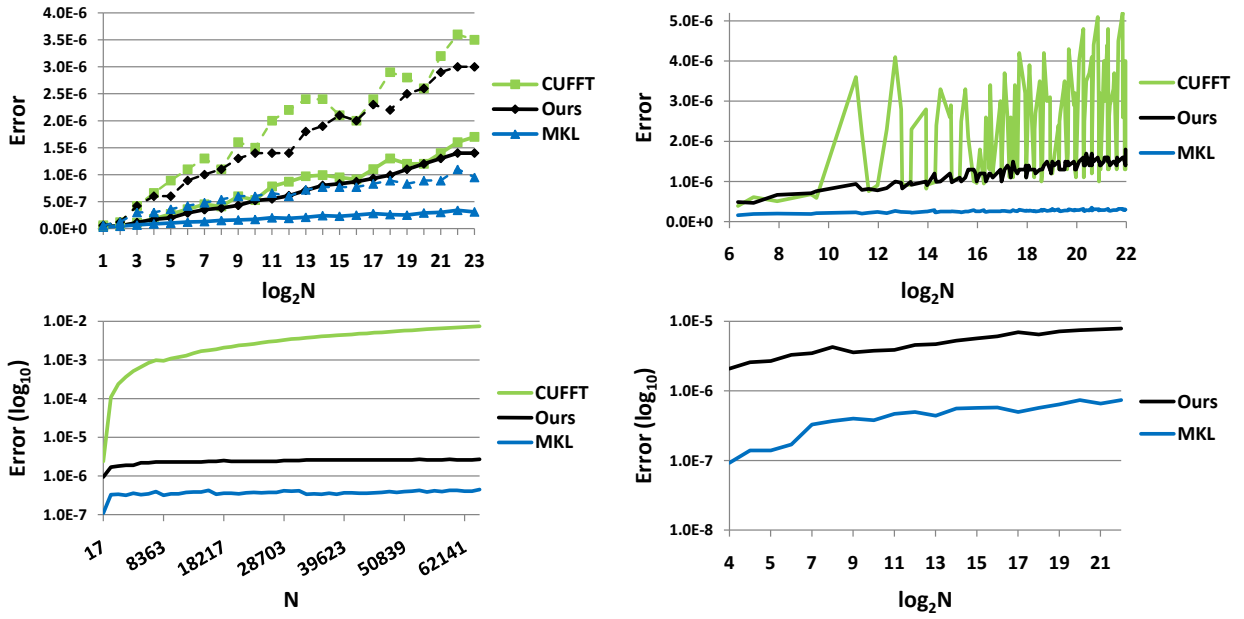

Fig. 17. **Error.** (Top-left) RMSE for 1D power-of-two FFTs. Maximum error, shown with dashed lines, is roughly proportional to RMSE. The constant of proportionality is approximately the same for other algorithms, so we do not included maximum error on the other graphs. The error for the GPU algorithms is about a factor of 5.5 larger than the error for MKL on the CPU for large N. The error scales roughly linearly with N. (Top-right) RMSE for 1D mixed-radix FFTs. The error for our library and MKL is about the same as for powers of two. CUFFT has a slightly higher error range and variance. (Bottom-left) RMSE for FFTs for small prime sizes N. The error of CUFFT grows very rapidly. (Bottom-right) RMSE for FFTs over a large range of prime sizes.

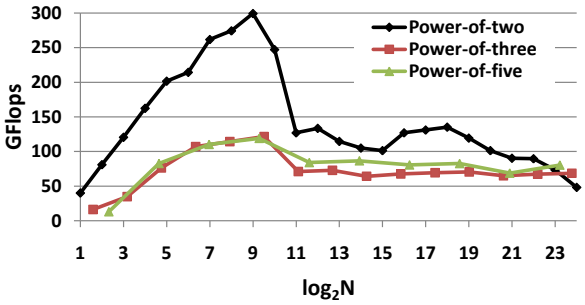

Fig. 18. **Comparison of batched 1D FFT with varying radix-R.** Power of two sizes use combinations of radix-2, radix-4, and radix-8. We use only radix-3 and radix-5 for the other cases. Performance could be improved by radices larger powers of 3 and 5. These performance numbers were obtained using a GTX280 with driver version 177.11.

For mixed-radix FFTs, the error for our library and MKL is about the same, but it goes up by over a factor of 2 for CUFFT. For FFTs of prime sizes, CUFFT's error rapidly balloons. However, the error for our FFTs is on the order of 10^{-6}, even for large sizes.

D. Limitations

Our algorithms are designed for single-precision complex sequences since the majority of currently available GPUs only support single-precision arithmetic. Since our techniques are general, the algorithms can be extended to work efficiently on double-precision inputs. Our algorithms currently work only on data that resides in GPU memory. External memory algorithms based on the hierarchical algorithm can be designed to handle larger data. Computation can also be performed on multiple GPUs. However, for both of these scenarios, data must be transferred between GPU and system memory, which can dramatically lower the performance. On current GPUs, our measurements show that the data transfer time is comparable to FFT computation time.

VII. Conclusions and Future Work

We have presented several algorithms for efficiently performing FFTs of arbitrary length and dimension on GPUs. We choose the algorithm that provides the best performance for a given input size and hardware configuration. Our hierarchical FFT minimizes the number of memory accesses by combining transpose operations with the FFT computation. We also address numerical accuracy issues. Our results indicate a significant performance improvement over optimized GPU-based and CPU-based FFT algorithms.

There are several avenues for future work. We would like to extend our library to use double-precision. One important issue is the computation of the twiddle factors. The $\cos()$ and $\sin()$ functions are currently much more expensive in double precision than single precision. For this reason it is probably better to use a precomputed table of twiddle factors. We would also like to add support for GPUs from other vendors by implementing our library using DirectX 11 Compute Shader API. Another interesting direction is mapping the FFT algorithms onto multiple GPUs.

Acknowledgements

We would like to thank Vasily Volkov for providing benchmark data for their implementation. Many thanks to Chas Boyd, Craig Mundie, Ken Oien, and Peter-Pike Sloan for useful suggestions and support during the course of the project. We would also like to thank Henry Moreton and Sumit Gupta from NVIDIA for hardware support.

References

[1] "HPC challenge," http://icl.cs.utk.edu/hpcc/, 2007.
[2] "NAS parallel benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html, 2007.
[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.
[4] "ATI CTM Guide," Advanced Micro Devices, Inc., 2006.
[5] *NVIDIA CUDA: Compute Unified Device Architecture*, NVIDIA Corp., 2007.
[6] C. Boyd, "The DirectX 11 compute shader," http://s08.idav.ucdavis.edu/boyd-dx11-compute-shader.pdf, 2008.
[7] A. Munshi, "OpenCL," http://s08.idav.ucdavis.edu/munshi-opencl.pdf, 2008.
[8] H. V. Sorensen and C. S. Burrus, *Fast Fourier Transform Database*. PWS Publishing, 1995.
[9] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial Mathematics, 1992.
[10] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003, pp. 112–119.
[11] J. Spitzer, "Implementing a GPU-efficient FFT," *SIGGRAPH Course on Interactive Geometric and Scientific Computations with Graphics Hardware*, 2003.
[12] J. L. Mitchell, M. Y. Ansari, and E. Hart, "Advanced image processing with DirectX 9 pixel shaders," in *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9.0*, W. Engel, Ed. Wordware Publishing, Inc., 2003.
[13] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on the GPU using a split-stream-FFT," in *Proceedings of the Vision, Modeling, and Visualization Conference 2004*, 2004, pp. 395–403.
[14] T. Sumanaweera and D. Liu, "Medical image reconstruction with the FFT," in *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, 2005, pp. 765–784.
[15] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," *Supercomputing 2006*, pp. 6–6, 2006.
[16] *CUDA CUFFT Library*, NVIDIA Corp., 2007.
[17] V. Volkov and B. Kazian, "Fitting FFT onto the G80 architecture," http:www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf.
[18] A. C. Chow, G. C. Fossum, and D. A. Brokenshire, "A programming example: Large FFT on the cell broadband engine," Whitepaper, 2005.
[19] L. Cico, R. Cooper, and J. Greene, "Performance and programmability of the IBM/Sony/Toshiba Cell broadband engine processor," Whitepaper, 2006.
[20] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, 2006, pp. 9–20.
[21] D. A. Bader and V. Agarwal, "FFTC: fastest Fourier transform for the IBM Cell broadband engine," *14th IEEE International Conference on High Performance Computing (HiPC)*, pp. 172–184, 2007.
[22] M. Frigo and S. G. Johnson, "FFTW on the cell processor," http://www.fftw.org/cell/index.html, 2007.
[23] D. H. Bailey, "FFTs in external or hierarchical memory," *Supercomputing*, pp. 23–35, 1990.