Small Discrete Fourier Transforms on GPUs

S. Mitra and A. Srinivasan
Dept. of Computer Science, Florida State University, Tallahassee, FL 32306, USA
{mitra,asriniva}@cs.fsu.edu

***Abstract* – Efficient implementations of the Discrete Fourier Transform (DFT) for GPUs provide good performance with large data sizes, but are not competitive with CPU code for small data sizes. On the other hand, several applications perform multiple DFTs on small data sizes. In fact, even algorithms for large data sizes use a divide-and-conquer approach, where eventually small DFTs need to be performed. We discuss our DFT implementation, which is efficient for multiple small DFTs. One feature of our implementation is the use of the asymptotically slow matrix multiplication approach for small data sizes, which improves performance on the GPU due to its regular memory access and computational patterns. We combine this algorithm with the mixed radix algorithm for 1-D, 2-D, and 3-D complex DFTs. We also demonstrate the effect of different optimization techniques. When GPUs are used to accelerate a component of an application running on the host, it is important that decisions taken to optimize the GPU performance not affect the performance of the rest of the application on the host. One feature of our implementation is that we use a data layout that is not optimal for the GPU so that the overall effect on the application is better. Our implementation performs up to two orders of magnitude faster than cuFFT on an NVIDIA GeForce 9800 GTX GPU and up to one to two orders of magnitude faster than FFTW on a CPU for multiple small DFTs. Furthermore, we show that our implementation can accelerate the performance of a Quantum Monte Carlo application for which cuFFT is not effective. The primary contributions of this work lie in demonstrating the utility of the matrix multiplication approach and also in providing an implementation that is efficient for small DFTs when a GPU is used to accelerate an application running on the host.**

I. INTRODUCTION

The Discrete Fourier Transform (DFT) is an important numerical kernel, used in a wide variety of scientific and engineering applications. A 1-dimensional DFT is defined as follows. The input to the transform is a vector **x** of N complex numbers. The output is a vector **y** of N complex numbers which is defined by:

$$\mathbf{y} = \mathbf{W}\,\mathbf{x}, \tag{1}$$

where **W** is an $N \times N$ complex matrix with $(j,k)^{th}$ element $e^{-i2\pi kj/N}$, $j, k \in \{0, 1, \ldots, N-1\}$, where i is the imaginary unit and $e^{i2\pi/N}$ is a primitive N^{th} root of 1. We consider out-of-place algorithms (**x** and **y** are distinct). DFTs are defined in higher dimensions too. If **x** is a 2-D array, then **y** is also a 2-D array. It can be obtained by first applying 1-D DFTs on each column and then 1-D DFTs on each row of the result of the previous step. Each step of the 2-D DFT can be expressed as the product of two matrices as shown later in equation (2). A 3-D DFT extends the 2-D DFT by applying a set of 1-D DFTs along each of three directions. Note that the amount of data movement for a transform in a higher dimension is significantly more than that in a lower dimension. Therefore, one needs to optimize for data movement too, rather than directly applying the lower dimensional algorithm multiple times.

The definition of the inverse transform is similar to that above, except that it involves division by N.

Direct computation of the DFT from the definition in equation (1) takes $O(N^2)$ time. Fast Fourier Transform (FFT) algorithms reduce this to O(N log N) by exploiting certain redundancies in the computation. This efficiency improvement for the 1-D transform carries over to higher dimensional transforms too. Furthermore, the FFT algorithms may also be more accurate than the direct computation.

FFT algorithms are usually expressed recursively, using a divide and conquer approach, where the problem size is successively reduced. The actual implementations, on the other hand, may be iterative.

Due to its better asymptotic time complexity, the DFT is computed, in practice, using FFT algorithms. In view of its importance, optimized FFT implementations are available on all platforms used for High Performance Computing. GPUs are becoming increasingly popular in High Performance Computing, and NVIDIA provides *cuFFT* [7] for its

platform. Other works too report implementations optimized to take advantage of the architectural features of GPUs. We summarize such related work on FFTs in section II, and describe relevant GPU features in section III.

The above implementations provide tremendous speedup over CPU for large data. However, for small data, they are not competitive with CPUs. On the other hand, many important applications require the use of small DFTs, such as in certain applications in quantum mechanics [12], finance [8], etc. In fact, the first application served as the motivation for this work. Even the FFT algorithms break up the problem, at a certain stage, into small enough pieces that an efficient algorithm for small DFTs will lead to better performance.

The basic idea behind our approach, explained further in section IV, is the use of matrix multiplication for small DFTs. Even though the asymptotic computational cost is higher than that of FFTs, it has the advantages of regular memory access and computational patterns. When the data size is not small enough to make it effective, it can be combined with an FFT algorithm in order to improve performance. In particular, we combine it with the mixed-radix algorithm. We explain details of the optimizations performed with our implementation in section V, and compare it against cuFFT on a GPU and FFTW [6] on a CPU, in section VI. We show that we can get up to two orders of magnitude improvement in performance over cuFFT and up to one to two orders of magnitude improvement over FFTW for multiple small DFTs in higher dimensions. We also show that our implementation can improve the performance of a Quantum Monte Carlo application, which cuFFT could not improve.

The results of our work will enable applications needing several small DFTs to make effective use of GPUs. Developers of FFT implementations can also use our implementation in the sub-problems of their computations, in order to improve their performance further. It will also be worth studying if this approach can be used on other architectures too.

II. RELATED WORK

A large body of research exists on FFT algorithms and their implementations on various architectures. The modern use of FFTs dates to the Cooley-Tukey algorithm [13]. It is based on the observation that the matrix **W** of equation (1) contains only N distinct entries for an FFT of size N. They used a divide and conquer approach to reduce the time complexity to O(N log N). Several variants of this approach were then introduced by others. Pease [14] introduced a tensor-product based algorithm. This approach is more effective for vectorization. Sorensen and Burrus have compiled a database of efficient FFT algorithms [3]. Van Loan [4] provides a matrix-based interpretation of different FFT variants.

For the GPU, Nakuda *et al*. have reported that through effective usage of on-chip shared memory, optimized usage of threads and registers and avoidance of low speed stride memory access, their 3D FFT kernel could achieve speeds up to 84 GFlops/card for power of 2 FFTs and up to sizes 256^3 [5].

Govindaraju *et al*. [1] have implemented 1-D and 2-D complex power-of-two and non-power-of-two FFTs. They obtain improvement by factors of 2-4 over cuFFT for large data sizes. They have also considered small data sizes. For a single DFT, the GPU implementation is much slower than a CPU implementation. However, with a large number of independent DFTs (they have considered $2^{23}/N$ independent DFTs of size N each in 1-D), they obtain substantial performance improvement.

Apart from the use of matrix-multiplication, one important difference between our approach and that of [1] is that many of our algorithmic choices are dictated by the constraints imposed by applications of interest to us. For example, [1] assumes that data resides on the GPU DRAM. We, on the other hand, have considered both cases, data residing on the host and data residing on the GPU DRAM. We need to consider the former, because it will enable an FFT-intensive application to accelerate the computation by just replacing their FFT calls with that from our implementation. On the other hand, if the application developer has ported much of their computation to the GPU, then one can assume that the data is present on the GPU DRAM, and then use the corresponding GPU implementation, which saves on the data transfer overhead.

We have also considered 3-D DFTs, because applications of interest to us require these. We have also optimized our implementations so that they outperform CPU implementations with even a small number of DFTs, because applications that we have analyzed generate approximately 1-100 independent DFTs typically. In contrast, the results of [1] assume 100,000 – 1,000,000 independent DFTs for small data sizes.

We have also used constraints imposed by the application to avoid certain optimizations. For example, CPU applications often use an array of complexes, which makes better use of cache. This can

cause inefficient use of shared memory banks on the GPU. GPU implementations, such as [1], use two distinct arrays to avoid this problem. However, this will require the application to change its data structure on the CPU, requiring substantial programming effort by the application developer, and also accepting the loss in performance of the CPU portion of the code. Alternatively, the DFT routine can convert the data layouts, compute, and reconvert it, which incurs additional overhead. Tests on small DFTs in the range 4-24, 4^2-24^2, and 4^3-24^3 in the 1-D, 2-D, and 3-D cases respectively show that the conversion time is typically larger than the DFT time when several simultaneous DFTs are computed, as we show in tables 1 and 2 below for the 1-D and 2-D cases. We have, therefore, used an array of complexes, even though we could have obtained better performance on the GPU by changing the data structure. If an entire application has been ported to the GPU, then it will be preferable to use two arrays of real to store the complex numbers on the GPU. For example, [11] has performed a rewrite of essentially an entire Quantum Monte Carlo application, involving over a hundred kernels, to the GPU. But FFT does not account for a large fraction of time there, unlike with the type of Quantum Monte Carlo application that we consider here.

N	DFT computation time on device Time: µs/DFT	Conversion time on host Time: µs/DFT
4	0.021	0.069
8	0.029	0.126
12	0.036	0.185
16	0.042	0.238
20	0.07	0.285
24	0.076	0.335

Table 1: DFT computation time per DFT vs data layout conversion time for 8192 1-D DFTs of size N.

N	DFT computation time on device Time: µs/DFT	Conversion time on host Time: µs/DFT
4	0.043	0.252
8	0.214	0.710
12	0.55	1.51
16	1.14	2.620
20	1.96	4.05
24	3.19	5.81

Table 2: DFT computation time per DFT vs data layout conversion time for 8192 2-D DFTs of size NxN.

III. GPU ARCHITECTURE

We have implemented our code using the CUDA programming framework and run it on two different NVIDIA GPUs. The results presented are those on a GeForce 9800 GTX. We summarize its architectural features below [2]. Other GPUs have similar features.

The above GPU contains 16 Streaming Multiprocessors (SM) that perform computation and 500 MB of DRAM memory to store global data, accessible by all the SMs and the host CPU. Each SM contains 8K 32-bit registers and 16KB shared memory. Each SM also contains a cache to store constant data (and also for texture data, which we don't use). Each SM can run up to 768 threads simultaneously.

In the CUDA framework, a GPU can be used to accelerate a process running on the host in the following manner. The process running on the host copies relevant data to the GPU and then calls a CUDA kernel. Once the kernel completes, its output data may be copied back to the host memory. The kernel call causes the creation of a number of threads, which are partitioned into blocks. Threads of the same block run on one SM and multiple blocks may be mapped to the same SM if sufficient resources are available. Threads within a block can synchronize efficiently and have access to the same shared memory. Threads on different blocks cannot synchronize safely, in a general situation. If such synchronization is needed, then it has to be accomplished by multiple kernel calls occurring in sequence, which will incur over ten micro-seconds of overhead. In contrast, synchronization within a block is supported by hardware.

The threads within a block are partitioned into groups of 32, called warps. The scheduler schedules threads for execution at the granularity of warps [1]. In any instance, all threads in a warp need to execute the same instruction, except that some threads may not execute anything. Consequently, there is a performance penalty if threads within a warp diverge on a branch.

Access to DRAM incurs a latency of hundreds of clock cycles. This latency can be hidden through simultaneous multi-threading (single instruction multiple threads, in particular); when one warp is stalled on data, another warp can be scheduled for execution. There is no context switch penalty. DRAM latency can also be reduced by using shared memory for data that is used multiple times. Access to shared memory has latency of one clock cycle.

The exact details of the memory access patterns also influence performance. For example, accesses by threads with adjacent indices to adjacent DRAM locations can be coalesced, improving memory access performance. Cache for constant data has one-cycle latency, but all threads in a warp need to access the same data for this to be effective. In shared memory access, we need to be aware that access by multiple threads in the same half warp to the same memory bank is serialized, unless all threads access the same location.

Yet another limitation to data flow is between the host and GPU, which are connected through PCIe x16. The bandwidth here is 4 GB/s in each direction or 8 GB/s each if PCIe 2 is used. If this bandwidth limits performance, then more of the computation needs to be performed on the GPU, so that data is produced and consumed on the GPU.

There are some additional issues one needs to be aware of. The memory is not ECC, and so there is some possibility of memory corruption. The arithmetic is also not IEEE-754-2008 compliant. These limitations are eliminated in the latest Fermi architecture from NVIDIA.

IV. DFT FOR SMALL DATA SIZE

In this section we provide further details on the DFT algorithm using matrix multiplication and the algorithm that combines it with the Mixed Radix algorithm.

A. *DFT using Matrix Multiplication*

1-D DFTs use matrix-vector multiplication as shown in equation (1). This can be computed in $O(N^2)$ time. 2-D DFT of input array X can be computed as follows.

$$\mathbf{Y} = \mathbf{W}\,\mathbf{X}\,\mathbf{W}^T, \qquad (2)$$

where $\mathbf{W}^T$ is the transpose of $\mathbf{W}$.

The 3-D DFT is computed by first computing 2-D DFTs for N 2-D planes in the x-y direction of the input array, and then applying N^2 1-D DFTs in the direction orthogonal to these planes. Before this step, we transpose the x-y and y-z planes of the result of the first step, so that the 1-D DFTs can be performed on adjacent data using matrix-vector multiplication. Finally, the above transpose operation is repeated to get results to the right locations. We perform DFTs for a plane at a time in the 3-D DFT code; that is, the same set of threads of a single block computes corresponding elements of each plane, one plane at a time.

B. *DFT based on Mixed Radix Algorithm*

The mixed radix method is a well known method for computing DFTs when the size is a composite number. We will summarize the computation structure for a 1-D transform without going into the mathematical aspects. Let $N = N_x \times N_y$. The 1-D data is considered as a 2-D array of size $N_x \times N_y$. 1-D DFT is applied to each column. Then, each element is multiplied by a certain twiddle factor, and then 1-D DFT is applied to each row. Basically, it is like a 2-D DFT, except that the elements are multiplied by certain factors between the two steps. If we use matrix multiplication to implement the transforms in each step of this algorithm, then the total number of complex multiplications is fewer than that with matrix multiplication used with 1-D transforms. In our results presented here, we choose N_x to be either 2 (for N=4,6), or 3 (for N=18) or 4. When multiple factorings were possible, we empirically evaluated the performance of each and chose the best one.

The 2-D transform is performed by considering the input as a 3-D array of dimension $N_x \times N_y \times N_z$, and then performing a computation analogous to that above; we perform two sets of 1-D transforms, where each set of 1-D transforms is as above, involving two matrix multiplications, multiplication by twiddle factors, and transpose. The 3-D transform applies 2-D transforms and 1-D transforms as mentioned earlier, except that each of these is replaced by their mixed radix implementations.

V. DFT IMPLEMENTATION

We have implemented our DFT library on the GPU using the CUDA programming framework for complex single precision DFTs. The results are reported for a GeForce 9800 GTX GPU. The host was a 2.6 GHz AMD Dual Core 885 Opteron Processor running Linux kernel version 2.6.18. The host code has been compiled with gcc version 4.1.2 and the kernel code has been compiled with NVCC version 2.0. The timer used is *gettimeofday* with a resolution of 1 micro second.

Our goal is to compute several small DFTs. Each DFT is computed by a separate block of threads, because synchronization is required between threads computing a single DFT. Several blocks are run when we need to compute several DFTs.

Applications may require computation of DFTs under two different situations. If the input data is produced on the host and the GPU is used to accelerate only the DFT, then the data has to be transferred to the GPU,

the DFT computed there, and the data transferred back. If the input data is produced on the GPU, then it is sufficient to compute the DFT on the data present on the GPU DRAM. We evaluate the performance of the implementations under both situations.

We now describe few different implementation choices we had, and present empirical results which helped us make the optimal choice.

We could place the *W* matrix in either constant memory or in shared memory. Placing it in constant memory would free up more space for other data. On the other hand, if different threads of a warp access different data, then the accesses would not make effective use of constant cache. We could read the input data into shared memory and operate on it there, or we could keep the input data in DRAM and read them into registers. Use of shared memory, of course, enables efficient reuse of data. On the other hand, it reduces the number of blocks that can run on the same SM because the size of the shared memory per SM is limited, and if the amount of shared memory used per block is large, then we will not be able to run multiple blocks on each SM. Running multiple blocks on each SM can hide the DRAM access latency better.

In table 3 and figure 1, each thread computes one element of the DFT. The first two columns of table 3 show that reading data in shared memory is much preferable, and the last two columns show that keeping *W* in shared memory is slightly preferable. Based on this figure, we decided to keep *W* in shared memory and read the input data into shared memory.

Number of DFTs	Input in DRAM	Input read in shared memory	
	W in constant memory Time: µs/DFT		W in shared memory Time: µs/DFT
1	26.1	19.8	16.5
16	3.61	1.31	1.05
256	3.32	0.833	0.743

Table 3: 2-D DFT performance comparison with data in different memory locations for a 12×12 2-D DFT with matrix multiplication.

Using the CUDA visual profiler, we determined that the reason for better performance with *W* in shared memory is that there are 24% fewer warp serializations with *W* in shared memory than with *W* in constant memory. Note that only 12 threads out of a warp of 32 threads access the same constant memory location when we use constant memory. Use of shared memory for the input data is preferable because the limit on the number of threads per SM does not permit enough blocks to run concurrently so as to hide the DRAM access latency well when the data is read directly from DRAM[1].

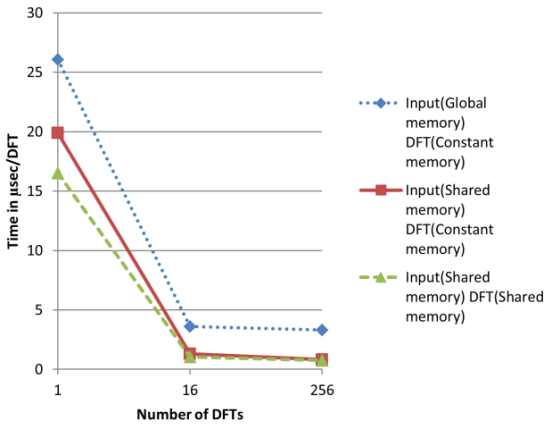

Figure 1: 2-D DFT performance comparison with data in different memory locations for a 12×12 2-D DFT with matrix multiplication.

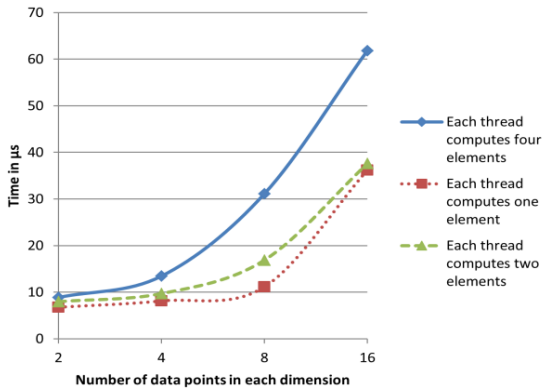

Figure 2: Performance comparison when each thread computes one element *vs* when each thread computes more than one element for a 2-D DFT using matrix multiplication.

We next determine whether it is preferable to have a thread compute multiple elements or only one element. If a thread computed multiple elements, then some of the initial overheads may be amortized over more useful operations. Furthermore, it may permit more blocks to run concurrently, by reducing the number of threads per block. Figure 2 in the previous

[1] This issue is handled in [1] by having each thread compute for a large number of elements. This causes performance to be improved when a large number of DFTs are being computed. But then, one needs a very large number of independent DFTs to get good performance.

page shows that it is preferable to have each thread compute one element. However, when data size is larger, this will exceed the number of permissible threads per block, and so we need to use a thread to compute multiple elements. Figure 2 shows that with larger data, it is preferable to have each thread compute two elements, rather than four. Note that this result relates to the computation for a single DFT. If we had a large number of DFTs, then the extra parallelism from multiple blocks can compensate for poorer performance from multiple threads computing for a single element. However, the applications that we have looked at do not appear to have a need for a very large number of independent DFTs simultaneously.

For 3-D DFTs of size N×N×N, N 2-D DFTs are computed followed by N^2 1-D DFTs. Table 4 shows that for a 3-D DFT using matrix multiplication, and for large data sizes (N=24), it is preferable to use more shared memory and compute 2 planes of 2-D DFTs per iteration rather than less shared memory and compute 1 plane per iteration.

N	Each iteration computing 2 planes of 2-D DFTs thereby requiring N/2 iterations Time: ms	Each iteration computing 1 plane of 2-D DFTs thereby requiring N iterations Time: ms
12	0.387	0.310
16	0.789	0.618
20	1.66	1.43
24	2.58	3.02

Table 4: Performance comparison of a 3-D DFT of size N×N×N using matrix multiplication with different numbers of 2-D DFTs being computed simultaneously per iteration.

Yet another parameter that we needed to fix was the maximum register count. In 3-D DFTs, by default, the number of registers used did not permit all the threads needed per block to run for the larger end of the data sizes that we use. We, therefore, set nvcc compiler option flag *-maxrregcount* to 24, in order to limit the register usage.

VI. EXPERIMENTAL RESULTS

In this section, we show performance results on 2-D and 3-D complex DFTs, comparing our matrix multiplication and mixed radix implementations against cuFFT on the GPU and FFTW on the CPU. We were not able to get the source code for [1], and so could not compare against it. In any case, the code for [1] requires data in a different layout, and the data conversion overhead would make the use of the GPU ineffective when the rest of the application on the host uses an array of complexes. We also show the performance improvement on a Quantum Monte Carlo application for electronic structure calculations. Except where we mention otherwise, the results are for the case where the input data is already on the GPU.

A. Results for 2-D DFT

We first compare the performance of the four algorithms for a single DFT in table 5 and figure 3. Our algorithms use a single block. cuFFT also appears to use one block, because synchronization overheads are unlikely to yield good performance with multiple blocks for this data size. We also implemented an optimized iterative power of two Cooley-Tukey algorithm in order to compare it against cuFFT.

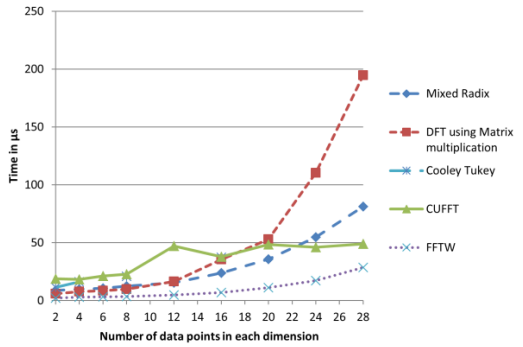

Figure 3: Performance comparison of a single 2-D FFT. The marks for Cooley Tukey are almost hidden by cuFFT.

N	Mixed Radix Time:µs	Matrix Multiplication Time:µs	Cooley Tukey Time:µs	CUFFT Time:µs	FFTW Time:µs
2	8.68	5.92	11.3	18.7	2.14
4	9.86	7.81	16.1	18.3	2.87
6	10.5	8.57		21.2	3.14
8	12.3	9.82	20.8	22.7	3.41
12	15.4	16.4		47.1	4.78
16	23.8	35.4	37.3	37.9	6.81
18	35.8	52.9		48.3	11.2
24	54.7	110		46.1	17.1

Table 5: Performance comparison of a single 2-D FFT of size N×N.

It is not surprising that FFTW is faster in this case; with one block, only $1/16^{th}$ of the GPU is used. We also see that matrix multiplication is better than mixed radix only for data sizes at most 8×8 (of course, the latter uses smaller matrix multiplications as its component), and is better than cuFFT for size up to 16×16. We also see that mixed radix is faster than cuFFT, except for size 24, where they are roughly the

same. We can also see that cuFFT and Cooley-Tukey have roughly the same performance for these data sizes (in fact, the marks for the latter are almost hidden by the former) suggesting that cuFFT uses one block.

N	Mixed Radix Time: µs/DFT	Matrix Multiplication Time: µs/DFT	Cooley Tukey Time: µs/DFT	CUFFT Time: µs/DFT	FFTW Time: µs/DFT
4	0.043	0.038	0.115	18.3	2.87
8	0.214	0.206	0.353	23.5	3.41
12	0.550	0.716		45.8	4.78
16	1.14	1.95	1.96	35.4	6.81
20	1.96	3.09		47.6	11.2
24	3.19	6.71		46.4	17.1

Table 6: Performance comparison of 512 2-D FFTs of size N×N.

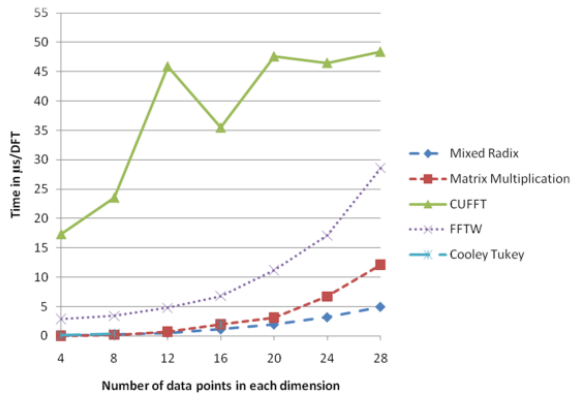

Figure 4: Performance comparison of 512 2-D FFTs of size N×N. The marks for Cooley-Tukey are almost hidden by those for DFT using matrix multiplication for N=4, 8 and 16.

When we use several blocks, table 6 and figure 4 shows that mixed radix is much faster than either cuFFT or FFTW. Matrix multiplication too is faster than either, but by a smaller factor for larger sizes and by a bigger factor for smaller sizes. Mixed radix is also substantially faster than Cooley-Tukey, but by a smaller factor. This suggests that if cuFFT were modified so that it could work on multiple DFTs simultaneously, its performance would improve substantially, and be better than FFTW for N<=16. When we analyze 3-D DFTs, we explain the reason for mixed-radix performing better than cuFFT. The reason is similar for 2-D DFTs.

B. Results for 3-D DFT

We first compare the performance of the four algorithms for 512 DFTs in figure 5. Mixed radix and matrix multiplication are much better than cuFFT and FFTW. As with 2-D FFT, Cooley-Tukey performed much better than cuFFT but worse than mixed-radix (results are presented in table 7). When a single DFT is computed, FFTW is better, as in 2-D DFTs, but when a large number of DFTs are computed, our GPU implementations are better. Figure 6 shows that the cross over point is at 5 DFTs (which still uses less than a third of the 16SMs available on the GPU) when 3-D DFT using matrix multiplication starts performing better than cuFFT. With mixed radix, it has been determined that the cross-over point is 5 DFTs when it starts performing better than FFTW and 4 DFTs when it starts performing better than cuFFT. With a large number of DFTs, mixed radix performs the best amongst all the algorithms considered, when data is greater than 8×8×8. Matrix multiplication performs best up to 8×8×8 data size.

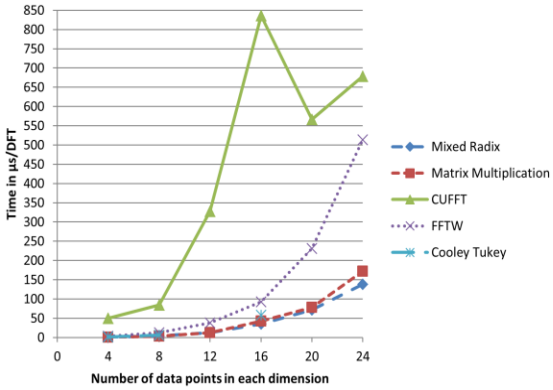

Figure 5: Performance comparison of 512 3-D FFTs of size N×N×N. The marks for mixed radix DFT implementation are almost hidden by those for DFT using matrix multiplication for N=4, 8 and 12.

N	Mixed Radix Time: µs/DFT	Matrix Multiplication Time: µs/DFT	Cooley Tukey Time: µs/DFT	CUFFT Time: µs/DFT	FFTW Time: µs/DFT
4	0.621	0.578	1.06	50.1	3.57
8	4.04	3.43	6.01	84.7	12.2
12	12.4	12.7		327	38.3
16	34.8	42.9	58.2	836	92.6
20	71.9	77.5		566	230
24	138	172		678	513

Table 7: Performance comparison of 512 3-D FFTs of size N×N×N.

We next explain the reason for mixed radix performing better than cuFFT. Based on the performance of Cooley-Tukey, one reason is likely due to the number of blocks used, which is primarily an implementation issue, rather an algorithmic issue. However, this does not account for the entire

difference. As mentioned earlier, matrix multiplication has better memory access and computation patterns which permits coalesced memory accesses and avoids divergent branches. Our mixed radix implementation uses matrix multiplication as its underlying implementation. However, the sizes of those matrices are small. Consequently, it does have un-coalesced memory accesses. However, CUDA visual profiler data suggests[2] that these are about 30% fewer than with cuFFT. Mixed radix also has many branches but few divergent branches. In contrast, cuFFT has 200 times as many divergent branches. The number of warp serializations is roughly the same, with mixed radix having about 10% fewer.

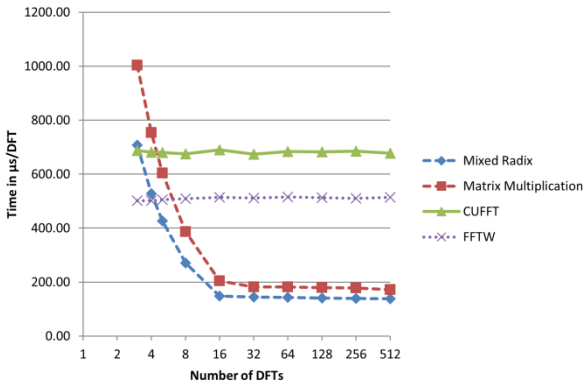

Figure 6: Performance 3-D DFTs of size 24×24×24 as a function of the number of DFTs computed.

The results above assume that the input data already resides in GPU DRAM. We now consider the situation where the data is on the host, and the GPU is used to accelerate the DFT computation alone. We need to take the data transfer cost into account. There are two alternatives that we can consider here. In the synchronous case, the entire data is transferred to the GPU, the DFTs are computed, and then the results are transferred back to the host. In the asynchronous case, streams are used as a data transfer latency hiding strategy. In each stream, the sequence of operations i.e. copying data from the host, kernel execution on this data and copying the data back to the host takes place successively. But different streams are interleaved. In this way, the data transfer and computation are pipelined, with asynchronous memory copy permitting the overlap of computation and communication.

Table 8 shows the results with synchronous data transfer for the GPU implementations. We have not used the matrix multiplication algorithm because we have already established that the mixed radix algorithm performs better than it for the data size shown below. The mixed radix algorithm still performs better than the cuFFT and FFTW, though the data transfer overhead decreases the extent of advantage it has over FFTW. On the other hand, cuFFT is not competitive with FFTW.

# of DFTs	Mixed Radix			CUFFT			FFTW
	Kernel Time: ms/DFT	Data transfer Time: ms/DFT	Kernel + Data transfer Time: ms/DFT	Kernel Time: ms/DFT	Data transfer Time: ms/DFT	Kernel + Data transfer Time: ms/DFT	Time: ms/DFT
16	0.147	0.273	0.420	0.691	0.260	0.951	0.514
32	0.145	0.258	0.403	0.674	0.268	0.942	0.512
64	0.143	0.249	0.392	0.684	0.245	0.929	0.515
128	0.141	0.245	0.386	0.682	0.257	0.939	0.513
256	0.139	0.244	0.384	0.685	0.242	0.927	0.510
512	0.138	0.245	0.383	0.678	0.259	0.937	0.513

Table 8: 3-D DFT performance with synchronous data transfer for the GPU algorithms on 24×24×24 input.

# of DFTs	Mixed Radix			FFTW
	Kernel Time: ms/DFT	Data transfer Time: ms/DFT	Kernel + Data transfer Time: ms/DFT	Time: ms/DFT
16	0.147	0.188	0.336	0.514
32	0.145	0.072	0.217	0.512
64	0.143	0.043	0.186	0.515
128	0.141	0.037	0.178	0.513
256	0.140	0.031	0.171	0.510
512	0.138	0.023	0.161	0.513

Table 9: 3-D DFT performance with asynchronous data transfer for mixed radix on 24×24×24 input.

Table 9 and figure 7 show similar results with asynchronous data transfer for mixed radix. Since we do not have access to the cuFFT source code, we could not implement asynchronous data transfer there. We can see that there is a substantial improvement in the performance of mixed radix. Also it is to be noted that the data transfer cost is not totally hidden by the computation. If we compare the results with table 8, we would expect the data transfer time to be fully hidden, because the data transfer time in each direction, which is half the transfer time shown in

[2] CUDA visual profiler uses data from only one SM. We tried to set up a computation where the results for both algorithms could be directly compared, but it is possible to have some errors due to the lack of control we have over profiling.

table 9, is less than the compute time. However there is an additional overhead which is not hidden. Note that there is also a difference in the memory allocated on the host for asynchronous transfer, which is pinned in memory, so that it will not get swapped.

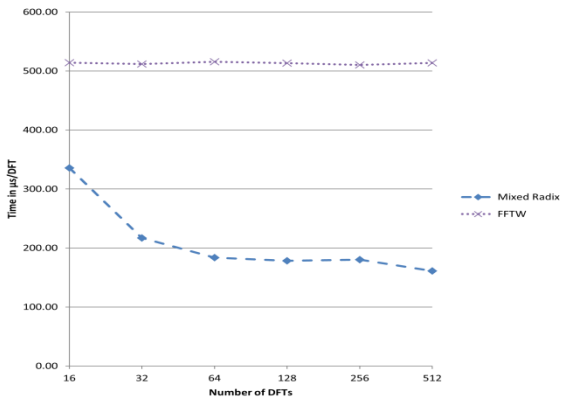

Figure 7: 3-D DFT performance comparison with asynchronous data transfer for input of size 24×24×24.

C. Performance improvement analysis for Quantum Monte Carlo Code

We now consider an Auxiliary Field Quantum Monte Carlo (AFMC) application for electronic structure calculations [12]. In contrast to other Quantum Monte Carlo codes that have been ported to GPUs [9,10,11], this class of calculations has DFT computations as a bottleneck. Profiling results on sample data showed that DFT calculations (including inverse DFT, which too we have implemented) account for about 60% of the time taken. The number of DFTs required and their sizes depend on the physical system simulated and the accuracy desired. In the application we used, the DFTs are 24×24×24. A few different functions call DFTs. The number of independent DFTs in each varies from 1 to 54. The functions that make most calls to DFTs also have the maximum number of independent calls, which is favorable to using our implementation. We accelerate only the DFT calls, letting the rest of the application run on the host.

We would expect the DFT time to decrease by a factor of 2.36 (approximately), based on the results of table 9, and so the performance of DFT computation time to be improved by about 57.6%. However, the speedup is less for the following reason. The code is in double precision. It will take considerable time for us to change the entire code to single precision and verify its accuracy. So, we converted the data needed for the FFT alone to single precision, and converted the output of the DFT back to double precision[3]. The conversion process has a significant overhead, which reduces the speedup.

Sl.#	Number of simultaneous FFT calls	# of FFTW calls	FFTW Time: s	Mixed Radix FFT Time: s	Performance improvement x times
1	27	298836	295	187	1.58
2	1, 27	15130	18	16.4	1.1
3	1, 27	10368	9.46	8.56	1.1
4	54	3456	3.40	2.14	1.59
5	54	1782	1.72	1.02	1.69
6	53	1462	1.61	0.821	1.97
7	1	64	0.029	0.03	0.993
8	1	2	0.004	0.036	0.111

Table 10: Performance comparison of single precision FFTW *vs* mixed radix for a QMC application. Column 2 shows the number of FFT calls made in bulk by each subroutine. Column 3 shows the total number of FFTW calls for each subroutine during 32 iterations of the QMC code.

For 32 iterations of the QMC code (total run-time of 548 seconds), it was observed that there were eight subroutines that were making calls to FFTW. Each subroutine calls FFTW different numbers of times, and the number of independent calls in each subroutine too differs. These factors cause different speedups in different subroutines, as shown in table 10. With the mixed radix implementation, and for the same number of iterations, the run time of the code was reduced to 426 seconds thereby making the code run about 1.3 times faster[4]. Use of a GPU with good double precision support will improve performance of the DFT further, by eliminating the single to double precision conversion cost. For further performance improvements, more application kernels need to be

[3] With better double precision support that is now available on the Fermi GPU, it does not appear fruitful to change the entire code to single precision in any case. We also verified that the answers agreed with the double precision answers to six significant digits. Single precision has been used for portions of codes in other QMC applications too, without significant loss in accuracy [10].

[4] One might wonder if using the multiple cores with FFTW would make FFTW better. But then, one can also use multiple GPUs on the same processor. In fact, we have a quad core processor with four devices. The performance of our algorithms on it was even more favorable there, because it has PCIe 2, rather than version 1 on the system for which we have reported the results. However, the machine crashed before we could obtain all results, and so we do not present results here.

moved to the GPU. However, the latter is not relevant to the issue addressed by this paper.

VII. CONCLUSION AND FUTURE WORK

We have shown that the asymptotically slower matrix multiplication algorithm can be beneficial with small data sizes. When combined with the mixed-radix algorithm, we obtain an implementation that is very effective for multiple small DFTs. For example, tables 6 and 7 show that our implementation is two orders of magnitude faster than cuFFT for 2-D and 3-D DFTs of sizes 4x4 and 4x4x4 respectively, when 512 simultaneous DFTs are performed with data on the device. It is faster than FFTW by a factor of around 75 for 4x4 2-D DFTs and by a factor of around 6 for 4x4x4 3-D DFTs in similar experiments. The mixed radix algorithm, which uses matrix multiplication as its underlying implementation, outperforms cuFFT and FFTW for the sizes considered here.

In future work, we wish to evaluate its effectiveness on more applications. We also wish to use it for small sub-problems of popular FFT implementations on the GPU and evaluate its effectiveness. (In fact, its use with the mixed radix algorithm is one such test.) Another direction is to extend the range of matrices for which these algorithms can be tried. We can also decrease the number of DFTs required to make effective use of the GPU by using more blocks for a single DFT. This will incur larger overhead due to kernel level synchronization. However, the time for a single DFT computation on a 24×24×24 input is around 2 ms, while the synchronization overhead is of the order of 0.01 ms. Thus, the overhead may be acceptable for 3-D FFT, and will be even lower, relatively, if we compute two or four independent DFTs concurrently.

Acknowledgment: This work was partially supported by an ORAU/ORNL grant under the HPC program.

REFERENCES

[1] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith and J. Manferdelli, *High Performance Discrete Fourier Transforms on Graphics Processors*, SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, (2008).

[2] *NVIDIA CUDA: Compute Unified Device Architecture*, NVIDIA Corp.,(2007), http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf

[3] H.V. Sorensen and C.S. Burrus, *Fast Fourier Transform Database*, PWS Publishing, (1995).

[4] C.V. Loan, *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial Mathematics, (1992).

[5] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, *Bandwidth Intensive 3-D FFT Kernel for GPUs using CUDA*, SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, (2008), 1-11.

[6] M. Frigo and S. G. Johnson, *The Design and Implementation of FFTW3*, Proceedings of the IEEE 93(2005) 216-231.

[7] *CUDA CUFFT Library*, NVIDIA Corp., (2009) http://developer.nvidia.com/object/cuda_2_3_downloads.html

[8] E. Benhamou, *Fast Fourier Transform for Discrete Asian Options*, Society for Computational Economics, (2001).

[9] A.G. Anderson, W.A. Goddard III, and P. Schroder, *Quantum Monte Carlo on Graphical Processing Units*, Computer Physics Communications, 177 (2007) 298-306.

[10] J.S. Meredith, G. Alvarez, T.A. Maier, T.C. Schulthess, and J.S. Vetter, *Accuracy and Performance of Graphics Processors: A Quantum Monte Carlo Application Case Study*, Parallel Computing, 35 (2009) 151-163.

[11] K.P. Esler, J. Kim, and D.M. Ceperley, *Quantum Monte Carlo Simulation of Materials using GPUs*, Proceedings of SC09, poster, 2009.

[12] K.P. Esler, J. Kim, D.M. Ceperley, W. Purwanto, E.J. Walter, H. Krakauer, S. Zhang, P.R.C. Kent, R.G. Hennig, C. Umrigar, M. Bajdich, J. Kolorenc, L. Mitas, and A. Srinivasan, *Quantum Monte Carlo Algorithms for Electronic Structure at the Petascale: the Endstation Project*, Proceedings of SciDAC 2008, Journal of Physics: Conference Series 125 (2008) 012057.

[13] J.W. Cooley and J.W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematic of Computation, 19 (1965) 297-301.

[14] M.C. Pease, *An Adaptation of the Fast Fourier Transform for Parallel Processing*, J. ACM, 15 (1968) 252-264.