Auto-tuning of Fast Fourier Transform on Graphics Processors

Yuri Dotsenko[†] Sara S. Baghsorkhi[‡] Brandon Lloyd[†] Naga K. Govindaraju[†]

[†]Microsoft Corporation [‡]University of Illinois at Urbana-Champaign

{yurido, dalloyd, nagag}@microsoft.com bsadeghi@illinois.edu

Abstract

We present an auto-tuning framework for FFTs on graphics processors (GPUs). Due to complex design of the memory and compute subsystems on GPUs, the performance of FFT kernels over the range of possible input parameters can vary widely. We generate several variants for each component of the FFT kernel that, for different cases, are likely to perform well. Our auto-tuner composes variants to generate kernels and selects the best ones. We present heuristics to prune the search space and profile only a small fraction of all possible kernels. We compose optimized kernels to improve the performance of larger FFT computations. We implement the system using the NVIDIA CUDA API and compare its performance to the state-of-the-art FFT libraries. On a range of NVIDIA GPUs and input sizes, our auto-tuned FFTs outperform the NVIDIA CUFFT 3.0 library by up to 38× and deliver up to 3× higher performance compared to a manually-tuned FFT.

Categories and Subject Descriptors D.2.2 [*SOFTWARE ENGINEERING*]: Design Tools and Techniques—Software libraries

General Terms Performance, Algorithms

Keywords Fast Fourier Transform, FFT, GPU, high performance, auto-tuning, performance analysis, performance tuning

1. Introduction

Graphics Processing Units (GPUs) are becoming increasingly popular in high-performance computing due to their low cost and high performance. Recent GPUs such as the NVIDIA Fermi GPUs have better support for double precision arithmetic widely used in scientific workloads and ECC memory for reliable computations. Over 0.5 Teraflop/s of double precision and a Teraflop of single precision performance combined with a peak memory bandwidth of over 150 GB/s, make GPUs ideal computing platforms for memory-intensive workloads such as Fast Fourier Transforms (FFTs).

FFTs are fundamental to many fields of science and engineering. They are widely applied in many high-performance problems including fluid mechanics, biomedical engineering, cryptography, wave simulations, etc. Recent research has demonstrated that GPUs can significantly accelerate performance of FFTs on GPUs as compared to CPUs [5, 11, 18]. With the increasing adoption of GPGPU computing in commodity applications such as games and multimedia, hardware vendors are making significant improvements to each new generation of GPUs. Thus GPU capabilities are improving more rapidly than those of CPUs. Optimizing libraries to leverage these new capabilities is tedious and time consuming. Auto-tuning techniques are necessary to handle the increasing complexity in GPU compute and memory subsystems. Nukada and Matsuoka [10] use tuning for optimizing 3D FFT performance on GPUs, but their approach has limited search space and is unlikely to be efficient for arbitrary FFT configurations. In this paper, we present a complete FFT framework to optimize an FFT library for arbitrary dimensions and sizes.

Our main contributions are as follows:

- **FFT Kernel Generation:** We present an extensive analysis of factors affecting performance of an FFT kernel and design an FFT kernel generator to construct variants amenable to auto-tuning. These include algorithmic variants for FFT kernel generation such as Stockham or transposed Stockham formulations, accounting for the impact of the stride of concurrent memory accesses, padding for shared memory exchanges, twiddle factor optimizations, and I/O specializations.
- **Pruning Heuristics:** We present heuristics that significantly prune the auto-tuning search space. We account for the GPU computation and memory models to motivate our pruning heuristics in order to achieve high GPU resource utilization and memory performance.
- **Runtime Execution and Size:** We present profile-based techniques to reduce the number of variants that need to be stored in the FFT library.

We implemented an auto-tuning framework that automatically optimizes FFT kernel variants using the NVIDIA CUDA API and tested its performance on different NVIDIA GPU architectures. On a large distribution of sizes, the average performance of our auto-tuned libraries is 150, 125 and 50 Gigaflops/s for single precision on NVIDIA GTX280, GTX260 and 8800GTX GPUs, respectively. For double precision, we achieve an average performance of over 23 and 18 Gigaflops/s on NVIDIA GTX280 and GTX260 GPUs, respectively. Our algorithms achieve over 4× higher performance than NVIDIA CUFFT 3.0 for the majority of the sizes and $> 30\%$ higher performance over the CUFFT best-case performance. Compared to Intel MKL performance on a high-end dual quad-core Xeon CPU, we achieve on average 12× and 3.3× higher performance for single and double precision for a wide range of 1D FFTs.

2. Related work

In this section, we provide a brief overview of the related work on auto-tuning techniques that particularly target FFTs on GPUs and CPUs. Van Loan [17] provides an overview of FFT algorithms and their variants.

The earliest FFT implementations on GPUs [9, 15, 8, 6, 16, 4] used graphics APIs such as DirectX or OpenGL to implement FFTs. The performance of these algorithms were limited by the restrictions present in these APIs such as lack of scatter support, access to local storage for data reuse and fine grain synchronization primitives. Recent research has focused on improving performance of FFTs on GPUs using computing languages such as CUDA [5, 11, 18]. These FFT algorithms achieve high performance on GPUs but require significant manual tuning to optimize the kernels. Because the hardware characteristics of GPUs varies widely with newer generation GPUs offering better memory hierarchy support, including larger local storage and register file sizes, memory bus widths, etc., it is non-trivial to optimize these algorithms for a range of GPUs.

Researchers have explored using auto-tuning algorithms on CPUs and FPGAs to optimize FFTs. Frigo and Johnson [3] presented an auto-tuning framework for optimizing FFT codelets on CPUs and Cell processors. Spiral [13] is a generator for optimized FFT libraries on FPGAs and CPUs. Chellappa et al. [1] later extended the algorithms to work well on Cell processors. These techniques are complimentary to ours. The factors that affect FFT performance on GPUs can be quite different than those on a CPU.

More recently, researchers have explored auto-tuning techniques for improving performance of algorithms such as GEMM and FFTs on GPUs. Ryoo et al. [14] used program optimization carving strategies to explore the optimization space and applied them to matrix multiplication. Li et al. [7] described auto-tuning techniques for optimizing GEMM on GPUs. Nukada and Matsuoka [10] presented an auto-tuning framework to optimize 3D FFT algorithms for mixed-radix sizes (the size of a dimension is $2^i 3^j 5^k$) on GPUs and demonstrated significant performance improvement against prior state-of-the-art FFT algorithms on GPUs. They efficiently optimize 1D FFTs of small sizes that are located contiguously in GPU memory and fit into shared memory; however, this set represents only a tiny subset of all FFT configurations. They perform a brute-force search on the entire space that is very large. We employ a similar approach for small 1D FFTs; however, we use a larger search space to utilize the GPU more efficiently and present space-pruning heuristics that reduce the auto-tuning time significantly. For other FFT configurations, our approaches are entirely different. Nukada and Matsuoka do not take advantage of fast on-chip shared memory, severely restricting the search space. This leads to sub-optimal performance, especially for FFTs of larger sizes (e.g., 1D FFTs) that require several passes over data located in GPU memory. In contrast, we exploit the shared memory to perform auto-tuning on a much larger space. This enables us to achieve higher performance for most FFT configurations due to reduction in the number of passes and more efficient GPU utilization. Another important difference is that we select the best performing FFT primitives taking into account input and output strides, which has significant impact on performance for most FFT configurations. We devise heuristics to reduce the auto-tuning time and to maintain reasonable profile size. While Nukada and Matsuoka's techniques can be applied to compute optimized FFTs for particular sizes, they may not be practical for optimizing an FFT library handling arbitrary dimensions and sizes. We present the first practical auto-tuning FFT framework on GPUs that can generate an optimized FFT library for arbitrary sizes and dimensions.

3. Overview

3.1 Graphics Processing Units

A GPU is made up of a number of multi-processors designed to handle a large number of threads. The execution model consists of many threads all running the same program called a *kernel*. All threads can access data stored in a high latency, high bandwidth global memory. The GPU schedules threads on each multi-processor in groups of a fixed number of threads called *thread blocks*. Threads within a thread block can communicate with each other through a processor's low latency local storage, referred to as shared memory, and can synchronize using a barrier primitive. Thread blocks are further decomposed into small groups known on NVIDIA GPUs as *warps*. Threads within a warp share a common instruction counter and are processed in SIMD fashion on the multi-processor, with divergent branches handled by predication. Each thread is also allocated a fixed number of the private registers. Kernel performance on a GPU is affected by computational load, multi-processor resource utilization, and memory performance. AMD and NVIDIA GPU architectures are similar, but AMD uses different nomenclature (thread blocks/*groups*, shared memory/*local memory*, warps/*wavefronts*).

Each multi-processor has a limited amount of shared memory, registers, thread contexts, and thread block contexts. The number of threads that can be concurrently scheduled on the multi-processor, or *occupancy*, is constrained by these resources. Because the latency of global memory is handled by switching between active thread warps whenever a stall occurs, high occupancy is important to overlap memory accesses with computation. Multiple blocks can be scheduled on a multi-processor to provide additional warps. A kernel that uses too many registers, too much shared memory, or thread blocks that are too small may have low occupancy, which can lead to lower performance. Thread blocks with large number of threads can also degrade performance because fewer blocks are available to cover stalls caused by synchronization.

Memory performance can vary based on the memory access patterns in the kernel. On NVIDIA GPUs, when threads within the same half-warp access the same banks of the interleaved shared memory, a bank conflict occurs and the accesses are serialized. Bank conflicts should be avoided. When thread warps access contiguous locations in global memory, the accesses are coalesced into a smaller number of accesses and achieve higher performance. Global memory performance can also be affected by the access stride. Even coalesced accesses at large strides can exhibit poor performance due to "partition camping" where accesses are concentrated in just a few DRAM banks. The best memory performance comes by avoiding memory accesses altogether and storing the data in registers, but the number of registers available per thread is typically quite small.

3.2 Fast Fourier Transform

The Discrete Fourier Transform (DFT) of a sequence $x = x_0, \ldots, x_{N-1}$ and its inverse are defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{kn}, \qquad x_n = 1/N \sum_{k=0}^{N-1} X_k \omega_N^{-kn},$$

where $k \in [0, N-1]$ and $\omega_N = e^{-2\pi i/N}$. A multi-dimensional DFT is computed by performing a multi-DFT independently along each dimension. To describe multi-dimensional sequences we will introduce some notation. We denote the dimensions of a multi-dimensional sequence by a tuple with the size of each dimension, represented by a capital letter, listed in reverse order, e.g. an $N_1 \times N_2 \times N_3$ sequence is denoted (N_3, N_2, N_1). We denote a specific element in the sequence by the tuple (n_3, n_2, n_1) with the corresponding small letter variables representing indices $n_i \in [0, N_i - 1]$. Assuming that the sequence is stored in a row-major array, the linear address of an element (n_3, n_2, n_1), denoted

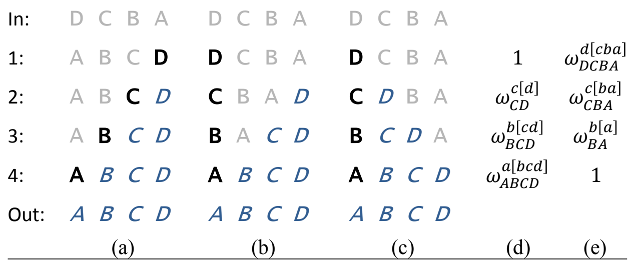

In: D C B A D C B A D C B A
1: A B C D D C B A D C B A 1 $\omega_{DCBA}^{d[cba]}$
2: A B C D C B A D C D B A $\omega_{CD}^{c[d]}$ $\omega_{CBA}^{c[ba]}$
3: A B C D B A C D B C D A $\omega_{BCD}^{b[cd]}$ $\omega_{BA}^{b[a]}$
4: A B C D A B C D A B C D $\omega_{ABCD}^{a[bcd]}$ 1
Out: A B C D A B C D A B C D
(a) (b) (c) (d) (e)

Figure 1. (a) Cooley-Tukey, (b) Stockham, and (c) Transposed Stockham FFT algorithms using a 4-dimensional partitioning (D, C, B, A). The partition being transformed in each step is shown in bold. Untransformed partitions are shown in gray and transformed partitions in italicized blue. (d) Pre-twiddle and (e) post-twiddle factors.

$[n_3 n_2 n_1]$, is $((n_3 \cdot N_2 + n_2)N_1 + n_1$. Consider a multi-DFT on the second dimension. It is computed by performing $M = N_3 \cdot N_1$ DFTs of size N_2. The elements of the sequence in each DFT share a common DFT index $m = [n_3 n_1]$ and vary in the index n_2.

The term "Fast Fourier Transform" (FFT) refers to an entire family of algorithms for computing the DFT. An FFT factorizes the DFT to reduce the number of computations from O(N^2) to O($N \log N$). An FFT of a single sequence can be computed by partitioning it into multiple dimensions and transforming each dimension, similar to multi-dimensional DFTs. Figure 1 shows several possible ways to arrange this computation. The Cooley-Tukey algorithm first reverses the partitions and then transforms each partition in-place. The Stockham and Transposed Stockham algorithms perform a multi-dimensional transpose in each step that results in the correct order for the output.

One difference from the multi-dimensional DFT is that each step requires that each element be scaled by a *twiddle factor* either before or after the DFT. We refer to this operation as *pre-twiddle* and *post-twiddle*, respectively. Pre-/post-twiddle correspond to what are commonly known in the FFT literature as decimation-in-time and decimation-in-frequency algorithms, respectively. The twiddle factor for each element is $\omega_{N_i L}^{n_i l}$, where n_i is the element's index in the partition being transformed N_i, L is the total number of elements in the untransformed partitions for post-twiddle or the transformed partitions for pre-twiddle, and l is the linearization of the partitions in L with indices in-order for pre-twiddle and reversed for post-twiddle (see Figure 1). Because L is empty for the first step of pre-twiddle and the last step of post-twiddle, the twiddle factors are 1.

If all of the partitions are the same size R, the algorithm is referred to as a *radix-R* FFT. When the partition size varies, the algorithm is referred to as a *mixed radix* FFT.

4. FFT kernels

To compute an $M \times N$ multi-FFT on the GPU, we partition the data, if necessary, and invoke a multi-FFT kernel to transform each partition. Each thread block of the kernel performs $M_b \leq M$ FFTs of size N_b, the size of the the current partition. We combine the FFT, the multi-dimensional transpose, and the multiplication by twiddle factors in a single kernel. For small partition sizes, each thread reads elements of a single FFT directly into registers and performs the computation. For larger partitions, the kernel applies an FFT recursively. In this case, each thread holds data for only one subpartition at a time in registers and must exchange data with other threads through shared memory at each step. Finally, the data is scaled (for inverse) and written back to global memory. Figure 2 shows the overall structure of the FFT kernel.

1. Read data from global memory into registers
2. Global pre-twiddle (if any)
3. Compute M_b FFTs. For each subpartition:
   (a) Local pre-twiddle
   (b) Compute FFT for data in registers
   (c) Exchange data through shared memory
4. Global post-twiddle (if any)
5. Scale and write data to global memory

Figure 2. Overall structure of an FFT kernel.

There are many possible variations for an FFT kernel. Some of the variations depend on the precision and data type of the input, e.g. single or double, real or complex. Others are high level algorithmic variations that may have impact on performance, e.g., partitioning scheme, partition sizes, and pre- vs. post-twiddle. Finally, some variants differ by the low-level optimizations applied to the FFT kernel such as optimized memory access patterns for particular configurations or the number of FFTs to compute in a single thread block. These low-level optimizations are based on hardware characteristics that can significantly vary across different GPUs. Not all variations have impact on performance. An important contribution of this research was to determine factors that significantly affect performance and to express them in a form applicable to autotuning.

While it is possible to write a generic kernel that takes into account all possible variations, it would be quite slow. To achieve high performance it is critical to specialize the FFT kernel. For example, making certain variables constant may enable the compiler to produce more efficient code via loop unrolling, dead code elimination, and control flow simplification. Dead code elimination is particularly important on GPUs because eliminating a code block that utilizes a lot of registers may reduce the register count for the entire kernel, leading to higher occupancy and performance. Specializing a kernel for specific configurations can eliminate the costly branches required for a more general code. The downside of specialization is a larger space of parameters to tune the FFT kernel for and therefore, larger profile size and auto-tuning time. In this section, we will discuss the various components of the FFT kernel and the specializations that we use.

4.1 Framework

Global memory performance is not always uniform for reads and writes and tends to degrade at higher strides. To better conform to the memory performance of a particular GPU we include multiple variants for the high-level FFT algorithm that have different memory access patterns. We use both the Stockham and Transposed Stockham algorithms. The former reads with high stride and writes with lower strides while the latter reads from lower strides and writes to high stride. We specialize the kernel based on the algorithm type to make sure certain assumptions hold about the relative size of the input and output strides.

4.2 FFT size decomposition

Given a large FFT size, we decompose it into smaller partitions that can be computed within a thread block. Within a thread block, we decompose each partition further into subpartions that can be performed entirely in registers by a single thread. We refer to the size of these subpartitions as the FFT radices. We hand-optimized codelets to compute an $N_t \times M_t$ multi-FFT for $N_t \in \{2, 3, 4, 5, 8\}$. We also have routines for $N_t \in \{6, 9, 10, 12, 15, 16\}$ which simply apply the Stockham algorithm in registers using the optimized codelets.

The choice of the decomposition in a thread block can significantly affect the resources used such as the number of registers, the number of active threads and the amount of shared memory used. The smallest radix in a given decomposition defines the maximum number of threads necessary to execute. Larger radices require more registers to perform the FFT computation. Therefore, in a given decomposition, the largest radix defines the maximum number of registers that each thread requires for the FFT. The combination of these two factors can significantly vary the performance of the underlying kernel as it affects the occupancy on a GPU multiprocessor.

A thread block can be two-dimensional. Threads in one dimension cooperate to compute the FFT of a single sequence. Threads in the other dimension work on independent FFTs. We refer to the size of each dimension as T_n and T_m, respectively. Note that $T_m = M_b$. We define step i of kernel FFT by a triplet $\langle N_t^i, M_t^i, T_n^i \rangle$, where N_t^i is the register FFT radix, M_t^i is the number of FFTs batched together in registers, and T_n^i is the number of threads used for a single register FFT.

4.3 Shared memory exchange

When threads cooperate to compute the FFT kernel, they must exchange values between each register FFT. This is done to accomplish the subpartition shuffling. When performing the shared memory exchange, bank conflicts can occur. Bank conflicts can be alleviated with padding, as shown for powers-of-two [5] or for simple cases [10]. Our kernels exhibit more complex patterns due to register FFT batching and processing several FFTs in a block. Instead of deriving an analytical formula, we estimate the cost of each transpose using a machine model. Shared memory accesses and index computation contribute to transpose cost. We compute shared memory locations and corresponding banks that will be accessed during kernel generation. We calculate the degree of conflict D for each half-warp and assume that the cost of shared memory access is D. We estimate the cost of computing the padded index $\mathit{Pad}(i) = i + \lfloor \frac{i}{\mathit{PadStride}} \rfloor \times \mathit{PadSize}$, which is dependent on radix decomposition. We then simulate several padding strides and offsets and select the one that minimizes the transpose cost.

4.4 Twiddle computation

Multi-stage thread block FFT, shown in Figure 2, requires twiddle factors that can be computed in several ways: (1) computed on-the-fly as needed, (2) precomputed in the kernel and stored in registers, (3) precomputed in the kernel in and stored in shared memory, or (4) precomputed on the host and stored in constant memory. Each method presents tradeoff between cycles spent computing twiddle factors, access speed (constant and shared memories are slower than registers), and memory and registers used. Twiddle factors can be computed in the kernel using fast transcendental approximation instructions, higher precision routines that are slower and require many registers, iterative algorithms, or a combination of these. Because it is not always possible determine beforehand which twiddle factor mode will perform best in a given configuration we generate several kernel variants considering the above cases, and auto-tune them.

Modes (2) and (3) achieve the greatest benefit when using a *streaming FFT kernel*, which computes multiple sets of M_b FFTs by wrapping the steps in Figure 2 in a loop. We found that it is actually not necessary to explicitly precompute the twiddle factors outside of the loop because the compiler can hoist on-the-fly computations from inside the loop and cache the values in registers. We carefully choose the number of thread blocks to invoke so as to maximize GPU utilization. Using the formulas from the CUDA Occupancy Calculator [2] and the known shared memory usage and register and thread counts for a kernel, we compute the number of active thread blocks that can run across all the multiprocessors. We invoke this number of thread blocks and compute the number of loop iterations accordingly.

4.5 I/O specializations

FFT kernel performance strongly depends on efficient utilization of memory bandwidth. We support several I/O specializations that optimize common access patterns for FFT kernels. Let S_i and S_o denote the strides of FFT elements for kernel's read and write, respectively. We specialize for $S_i = S_o = 1$ case when the FFT can be computed in one pass. We call this kernel a *row* kernel, since FFTs are computed along the rows of the 2D matrix. The maximum size of row kernel is limited by the amount of available shared memory. The accesses are poorly coalesced for small N. We address this by additionally specializing the kernel to fetch a contiguous block of M_b FFTs into shared memory and perform a *coalesce transpose* [5] to properly rearrange data.

The *column* kernels, which operate along the columns of the 2D matrix, are additionally specialized to improve coalescing. First, we specialize kernels that operate at large strides (*LS* kernels) $\max(S_i, S_o) \geq M_b$; this optimizes memory accesses for the most common case. However, *LS* kernels do not coalesce memory accesses for the second important case when $S_i = 1$ or $S_o = 1$. We optimize I/O for this case by performing a coalesce transpose, and call these kernels *SSC* – small-stride coalesced. *SSC* kernels also coalesce data for small power-of-two strides $S \leq M_b$, since we always choose M_b to be a power-of-two. The remaining cases are hard to optimize efficiently due to complexity of the FFT memory access patterns. We handle them with the *SSN* (small-stride no-coalescing) kernels without coalescing specialization. *SSN* kernels are used rather infrequently, primarily to transform small-sized lower dimensions of multidimensional FFTs.

5. Auto-tuning of FFTs

Our FFT auto-tuning system performs offline tuning; *i.e.*, it collects a performance profile for a combination of the GPU, compiler and GPU driver. No new kernels are generated or profiled at runtime. The auto-tuner does a parameter sweep over the tuning space and creates a performance profile that, for each possible step of a global FFT, contains the runtime estimation and parameters of the best performing kernel variant. Profiling is done for a large data volume, e.g., 128 MB, since GPUs are typically used for large data sizes to fully leverage the available massive parallelism. The collected profile information is used at runtime to create the best partitioning and kernels to use for mixed radix FFT.

We split the space into three parts that we call the FFT space, stride space, and kernel space. The FFT space includes:

Precision type : single (SP), double (DP)
Data type : complex, real
Direction : forward or inverse

Radix : $N_b = 2^i 3^j 5^k$ of the FFT size decomposition
Algorithmic variants : Stockham, Transposed Stockham
Twiddle type : pre- or post-twiddle

The stride space represents pairs of input and output strides $\langle S_i, S_o \rangle$. A global FFT step corresponds to a combination of one FFT space point P_{FFT} and one stride pair.

A point P_K in the kernel space represents one specialized FFT kernel. The kernel space includes:

Kernel type : row or column
Decomposition of the kernel radix into $\langle N_t^i, M_t^i, T_n^i \rangle$
Number of FFTs in a thread block – M_b
Twiddle reuse type : single or streaming FFT kernel
Twiddle type : computed or table twiddle
Block type : full (M_b FFTs per block) or partial ($\leq M_b$ FFTs)
Row kernel I/O type : coalesced or uncoalesced
Column kernel I/O type : *LS*, *SSC*, *SSN*
Merging of global twiddle with I/O : yes or no
Pow2 global twiddle stride : use shifts instead of divisions
Iterative global twiddle : use trig. identities to apprx. sin/cos

We discussed most of the codelet specializations in Section 4.

Figure 3 gives an overview of our framework. We first generate the FFT space. For each P_{FFT}, we generate several variants of FFT kernels. Using similar variants with different M_b, we perform a preliminary step to prune slow-performing variants. The remaining variants are compiled in parallel to reduce compilation time. Row kernels require one profiling run; however, column kernels require many runs for different values of $\langle S_i, S_o \rangle$. Each new performance measurement updates the performance database that we use to prune slowly performing column kernels. We further reduce profile time and size by profiling only forward kernels and deriving profile of inverse-FFT kernels from the forward kernels. This approximation is reasonable since inverse FFT differs from forward FFT by only a few instructions. In the rest of the section, we present some of our pruning heuristics to reduce the tuning search space.

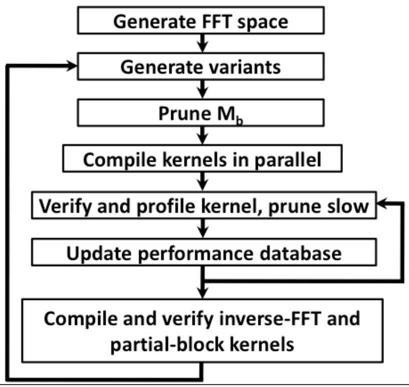

Generate FFT space
Generate variants
Prune M_b
Compile kernels in parallel
Verify and profile kernel, prune slow
Update performance database
Compile and verify inverse-FFT and partial-block kernels

Figure 3. Scheme of our auto-tuning framework.

5.1 M_b pruning

M_b defines how many FFTs are batched in one thread block (TB) and has high impact on occupancy and coalescing. For example, if N_b is small, a kernel with $M_b = 1$ yields very low performance because the total number of FFTs that can run on one multi-processor is limited by the current maximum of 8 resident thread blocks per multi-processor (MP). Executing so few small FFTs per MP significantly under-utilizes the GPU. The inverse is also true: too many FFTs batched together may reduce overall performance because the cost of synchronization increases. In addition, GPU resources may be underutilized due to resource fragmentation, e.g. if a TB uses slightly more than a half of the shared memory, only one TB can be resident on a MP, and the remaining shared memory is not used. M_b value may also affect memory performance. For small N_b, we coalesce memory accesses in row kernels, but we cannot do so if M_b is very small, say 1. It is difficult to analytically determine the impact of M_b on performance for all combinations of N_b and M_b. Clearly, small radix and M_b combinations such as $N_b = 2$ and $M_b = 1$ can be pruned but it is unclear where the pruning boundary is. The decision is even harder because we do not know how many resources, such as registers, are required by a kernel until we compile the kernel. Instead of pruning M_b statically, we use a runtime "probe". We select matching variants, one per M_b, and compile and profile them. We sort M_b according to the probe performance. If probe's performance for a particular M_b is on average $\eta\%$ (say $\eta = 40\%$) slower than the best, we prune all kernels with this M_b, unless there was a run that yielded best performance. Pruning on M_b reduces the time required to compile and profile considerably. It also helps reduce the time to profile slow variants that may take long time to execute. Ranking M_b enhances column kernel pruning by increasing probability that fast variants run before slower ones.

5.2 Decomposition of FFT kernel radix

After $\langle N_t^i, M_t^i, T_n^i \rangle$ are computed, we prune some factorizations; we do not prune permutations of the same factorization because they may provide different number of I/O threads, padding, and code shapes. First, we prune configurations that require too many registers. We do not compile the kernel to reduce auto-tuning time; instead, we use the following heuristic. We prune a variant if the total register count $\textit{NumRegs} = P \times (2 + \textit{TempRegThreshold}) \times R \times T_n \times M_b$ exceeds the maximum register count per thread block, allowed by the architecture. P accounts for data precision: $P = 1$ for SP and $P = 2$ for DP. The factor of 2 is for real and imaginary components. $R = \max_i(N_t^i \times M_t^i)$ is the maximum number of FFT data elements held in registers. The *TempRegThreshold* approximates how many additional, temporary registers are required to run FFTs; we chose $\textit{TempRegThreshold} = 0.8$ so as to not prune too aggressively but to prune configurations that significantly exceed the machine register limit.

Second, we prune decompositions aggregating small radices. Small radices such as 2 and 3 lead to decomposition explosion – it is still fast to generate them, but compilation and profiling may be infeasible. Moreover, the number of shared-memory exchanges is large for decompositions with a lot of stages, and such decompositions are unlikely to perform well due to extra synchronization and instructions necessary to perform the exchanges. We prune a decomposition, if there are two factors N_t^i and N_t^j such that $N_t^k = N_t^i N_t^j$ and N_t^k is also a factor in the decomposition. This means that instead of two stages $\langle N_t^i, M_t^i, T_n^i \rangle$ with $M_t^i = N_t^j M_t^k$, and $\langle N_t^j, M_t^j, T_n^j \rangle$ with $M_t^j = N_t^i M_t^k$, we perform one stage $\langle N_t^k, M_t^k, T_n^k \rangle$ with a larger radix $N_t^k = N_t^i N_t^j$. One stage of a larger radix FFT is faster because there is no shared-memory transpose (an equivalent transpose happens in registers). Moreover, the composite radix N_t^k may be hand-optimized. Note that if there is no "assimilating" larger radix, we do not prune the decomposition. This lets us keep decompositions consisting of only small radices, e.g., $16 = 2 \times 2 \times 2 \times 2$.

5.3 Row and column kernel tuning

The kernel space for row kernels is relatively small, since they do not require algorithmic and twiddle specializations. The row kernels always operate at stride 1, and require only one performance sample per kernel, which takes very little time to collect. Almost all auto-tuning time is spent in compiling the variants. The variant generation time is negligible for both row and column kernels.

The tuning of column kernels is more time consuming. The kernel space is much larger. In addition, the performance of column kernels strongly depends on $\langle S_i, S_o \rangle$ stride values. The general trend is that performance decreases as strides increase, but the decrease is not monotonic or predictable. Therefore, we also perform the parameter sweep for each kernel at varying input and output strides to capture the behavior. However, the stride space is very large. We tune for a total data size of 2^{24} complex elements (SP), for which there are $\approx$115K legal stride pairs that are powers of 2, 3, and 5, which already makes the space too large. Moreover, there are many more legal stride pairs that are not powers of 2, 3, and 5, which may be induced by the lower dimensions of a multidimensional FFT. So, it is unrealistic to collect the exact performance profile for all legal stride combinations. Instead, we sample the stride space and approximate performance of a particular $\langle S_i, S_o \rangle$ pair by its closest stride sample from the same class.

We use a heuristic to identify sampling stride pairs $\langle S_i, S_o \rangle$. We classify $\langle S_i, S_o \rangle$ into five classes: $\langle \mathit{Pow2}, \mathit{Pow2} \rangle$, $\langle \mathit{CO}, \mathit{CO} \rangle$, $\langle \mathit{CO}, \mathit{NC} \rangle$, $\langle \mathit{NC}, \mathit{CO} \rangle$, $\langle \mathit{NC}, \mathit{NC} \rangle$, where *Pow2* denotes a power-of-two stride, *CO* denotes a stride for which most accesses are coalesced ($\mathit{CO} = \mathit{CW} \times k$, where *CW* is the coalesce width of the GPU, or *CO* is a small power-of-two for a SSC kernel), *NC* denotes other strides that make most accesses uncoalesced. The power-of-two stride pairs form a special class, rather than being a subset of $\langle \mathit{CO}, \mathit{CO} \rangle$, because on some GPUs performance of column kernels significantly degrades for large power-of-two strides, e.g. on the GTX280 that has 8 memory banks. This degradation is probably caused by partition camping and associativity effects.

We partition S_i and S_o dimensions into intervals; this induces a rectangular grid on the stride space. We select a sample pair for each stride pair class and for each block of the grid (if a legal stride pair exists). To keep the auto-tuning time manageable, it is desirable that the number of samples be relatively small. Based on this requirement and the observation that for large strides of the same class, the performance does not change as much as for smaller strides, we partition S_i and S_o dimensions into logarithmically-scaled intervals.

We measure runtime of a generated column kernel for each legal sample stride, and record t_e, time per element, in the performance database. The total number of samples is still very large and we use pruning to abort profiling of slow variants. We prune any kernel that exceeds shared memory or thread count limits. We allow kernels to exceed the maximum register limit by a small threshold, since registers can be spilled. After profiling, we record the best time per stride sample in performance database records. When profiling a column kernel, we count the number of slow samples that take more than $\eta\%$ (say $\eta = 20\%$) more time to run than the best recorded time. When the number of slow samples exceeds a certain threshold (say 20), we declare the kernel slow and do not profile it further. We reset the slow sample count each time the variant shows time that is better than the recorded best.

5.4 Schedule creation

After the tuning is done, we use the performance database to construct a partitioning and collection of kernel variants that delivers high-performance. A large FFT of length N is computed by partitioning the data into multiple dimensions and applying a kernel to each dimension. We refer to such a partitioning as a *schedule*. We take the estimated runtime per element t_e of each partition from the performance database. We collect performance profiles for several kernel radices, therefore there are several decompositions of N. We enumerate all schedules and select the schedule with minimum t_e. A schedule is created and cached before the FFT is executed; the overhead is on the order of milliseconds, which is negligible if the schedule is reused several times.

The performance database does not contain precise t_e for each stride pair. To get an estimated t_e for a given $\langle S_i, S_o \rangle$, we locate the closest sample pair within the same class; we use the Euclidean distance metric. The prediction is not perfect and introduces error. To estimate this error, we selected an FFT configuration for Stockham, pre-twiddle, radix-16 NS kernel and measured its performance for each legal stride pair $\langle S_i, S_o \rangle$, $S_i = 2^{i_1} 3^{j_1} 5^{k_1}$, $S_o = 2^{i_2} 3^{j_2} 5^{k_2}$, $S_o \leq S_i \leq 2^{20}$, and compared measured performance with the predicted. We observe that for roughly 50% of the stride pairs the prediction is less than 10% off and 80% of the stride pairs have a prediction error less than 20%. In the schedules where these stride pair kernels are used, the average prediction error due to our stride pair performance approximation is around 7% for 1D FFTs.

We construct four schedules corresponding to the profiled algorithmic variants: Stockham or Transposed Stockham and pre- or post-twiddle. We then measure runtime of each schedule. The measurement is done lazily, whenever the user code invokes the FFT library. After a few pilot runs, the library decides on the best performing schedule and uses it for all following invocations. Since schedules usually have comparable performance, the impact on overall performance is negligible and is further amortized by several consecutive invocations. Note that different algorithmic variants can be selected for forward and inverse FFT. The runtime selection of the variants allows our library to deliver higher performance, partially compensating for the misprediction error.

5.5 Performance profile size

The collected performance profile is quite large. Each sample includes best measured t_e and corresponding kernel id. The size of runtime sample information can be controlled by reducing the amount of specialization and the number of samples, e.g., we can use fewer radices and coarser sampling intervals.

Many specialized kernels must be included into the performance profile. The *cubin* representation for compiled kernels that we currently use leads to the kernel code being much larger than that of the performance data. We performed an additional study to reduce the number of kernels in the final profile by reducing the amount of kernel specialization in two ways. (1) We combined corresponding forward and inverse FFT kernels into a single, less specialized kernel. To compute the inverse FFT, the input is conjugated before computing the forward FFT and the output is conjugated afterwards. (2) We merged corresponding full- and partial-block kernels. Instead of static full-block specialization we dynamically detect whether an instance of a thread block executes M_b FFTs (full block) and invoke specialized code for this case. Applying both techniques leads to a factor of ≈ 4 reduction in the number of kernels and the overall kernel profile size, but comes at a cost of $\approx 4.5\%$ performance loss on-average.

GPU	Core Clock (MHz)	Shader Clock (MHz)	MP	Peak Perf. (GFlops)	Memory Clock (MHz)	Memory (MB)	Bus Width (bits)	Peak BW (GiB/s)	Registers per MP	Shared memory	Driver
GTX280	650	1300	30	936	1150	1024	512	137	16K	16K	196.21
GTX260	575	1350	24	777	1000	1024	448	104	16K	16K	196.21
8800GTX	575	1350	16	518	900	768	384	80	8K	16K	196.21

Figure 4. GPUs used in experiments. MP is the number of multi-processors. The warp width for all the GPUs is 32.

We biased the set of kernels included into the final profile towards more frequently used kernels. We select a set of representative FFT configurations and mine the performance database to detect fast kernels that are to be selected during schedule construction. All kernels that were not selected are removed from the database and replaced by the equivalent selected kernels. We constructed the sample profiling set that corresponds to an FFT of size N residing in memory at stride s, which represents a dimension of a multidimensional FFT. $\forall s \in S$, where S is the set of sample strides, we construct schedules for the second dimension N, such that $N = 2^i 3^j 5^k$ and $N \leq \frac{E}{s}$, where E is the total number of data elements. Note that $s = 1$ represents all mixed radix 1D FFTs. To generate set S, we perform a logarithmic sampling of $[1, E)$ to bias sample density towards smaller strides as a kernel's performance varies more for smaller strides than for larger strides.

To further reduce the library size, we apply standard compression algorithms to compress the kernels. As the data flow for the different kernels can be similar, there is significant redundancy and the compression algorithms achieve a high compression ratio. At runtime, the kernels required by the schedules are decompressed and cached for later execution.

6. Implementation and profiling

We implemented an FFT kernel generator that supports all specializations described in Sections 4 and 5. Our generator is written in C++ and targets NVIDIA's CUDA API, but can easily be extended to support DirectX or OpenCL. FFT kernel specializations are expressed either via compile-time constants or via hand-optimized codelets. The generator enumerates the space and, for each point, creates a CUDA function that performs the specific FFT computation. The performance database includes *cubin* kernels and their estimated runtimes. We profiled using a large multi-FFT with as many FFT sequences as can fit into 2^{24} (2^{23}) single (double) precision elements. We collected performance profile for forward-FFT, full-block kernels that support both real and complex inputs.

We profiled all feasible row kernels. The total number of variants is ≈ 300 per GPU; the size of performance samples is negligible. The majority of the auto-tuning time is spent in compiling the kernels.

The FFT space for column kernels is much larger. We specialized kernel radix, FFT algorithm, global twiddle type, global twiddle stride, and kernel I/O. This alone already contributes a factor of $\approx 16\cdot$ (number of radices) to the auto-tuning time and profile size. The majority of auto-tuning time is spent running the kernels. To keep the time and space manageable, we selected 2^i points to partition the stride space and added a few points for smaller strides since performance is more sensitive for smaller stride values; this resulted in 2666 total sample stride pairs. To study the space, we collected data for 53 radices (SP). This results in $\approx$ 522K performance samples and $\approx$ 44K kernels that may be used to run any mixed-radix FFT. Each sample requires 6 bytes: 2 bytes for kernel id, 2 bytes for time estimation, and 2 bytes for kernel variant parameters, totaling to $\approx$ 3MB uncompressed. Kernels are kept in a *cubin* form and compress to $\approx$ 45MB.

We also collected a performance profile for less specialized kernels that do not have forward/inverse FFT and full-/partial-block specializations. We observed $\approx 4\times$ reduction in the profile size and $\approx 4.5\%$ performance loss for a large sample set of FFTs. We conclude that removing these specializations is pragmatic.

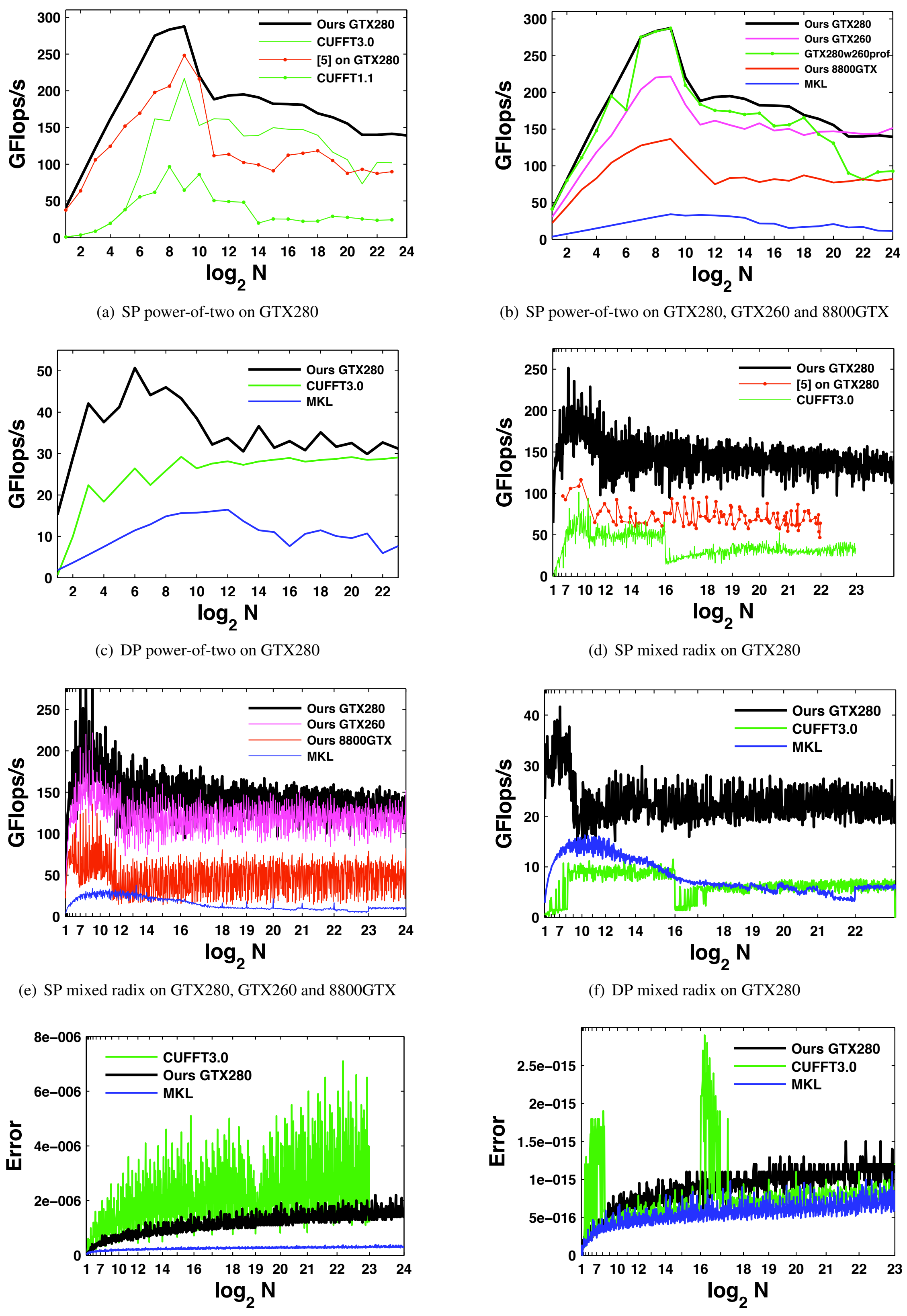
To evaluated the effect of biased kernel selection, we reduced the initial performance database using three sample profiling sets S (see 5.5) of size 1 (s=1 or only 1D FFTs), $\approx$ 500 and 1000 samples. To estimate performance loss, we constructed a larger randomly-generated evaluation set $\{S', N'_s\}$ such that a pair $\langle s', n'_s \rangle$ describes a mixed radix FFT computation of the second dimension n'_s of a 2D FFT $s' \times n'_s$. To our surprise, we found almost no difference in average performance among all four data bases (for both fully and less specialized profiles), indicating that it may be enough to collect only kernels used by the 1D FFTs to yield a good performance profile for all FFT sizes and dimensions. The biased selection for $|S| = 1$ resulted in $\approx 13\times$ reduction in the total number of kernels!

Auto-tuning time depends on parameters selected for tuning (radices, sampled strides, algorithm and twiddle types, etc.), speed of the CUDA compiler, and the GPU(s). For row kernels, compilation time dominates. For column kernels, performance sampling dominates. To obtain a high-performance library requires 2–3 days on a single PC/GPU. Beyond that performance gain diminishes rapidly. Auto-tuning time can be further reduced by using multiple PCs and GPUs.

7. Performance results

Experimental Methodology: We evaluate our approach on three different GPUs: GTX280, GTX260, and 8800GTX with specifications shown in Figure 4. The GPUs differ in memory configuration, the amount of memory bandwidth and compute power. An older 8800GTX has stricter coalescing requirements as discussed elsewhere [12]. We also observed that large-stride memory accesses show better performance on GTX260 than on GTX280 despite less available bandwidth, especially, for strides that are multiples of a large power-of-two.

We compare performance of our auto-tuned library, which uses CUDA 2.3, to that of NVIDIA's FFT library (CUFFT) version 3.0 and Intel's MKL version 10.2, which was run with 8 threads on a 2.27GHz Intel quad-core dual Xeon L5520 machine. We also compare performance of 1D and 3D FFTs with results published in [5, 10]. We measure performance in GFlops/s, which we compute as $\frac{\sum_{d=1}^{D} M_d (5 N_d log_2 N_d)}{\text{execution time}}$, 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 elements. We measure runtime of 100 runs on GPU using CUDA events [12], and use the median as the execution time. We do not measure data transfer times and assume the data is already present in GPU video memory. We measure accuracy by performing a forward transform followed by an inverse transform on complex data uniformly distributed in $[-0.5, 0.5)$. We then compare the result to the original data and divide the root mean squared error by 2.



(a) SP power-of-two on GTX280

(b) SP power-of-two on GTX280, GTX260 and 8800GTX

(c) DP power-of-two on GTX280

(d) SP mixed radix on GTX280

(e) SP mixed radix on GTX280, GTX260 and 8800GTX

(f) DP mixed radix on GTX280

(g) Accuracy of single precision FFTs

(h) Accuracy of double precision FFTs

Figure 5. Performance and accuracy of batched FFTs for single precision (SP) and double precision (DP)

Size	256x256x256	256x256x128	256x128x256	256x128x128	128x128x128	144x144x144	192x192x192	192x256x144
Ours	176	173	179	157	178	156	142	169
AT3dFFT	135	143	135	132	105	117	130	131
CUFFT3.0	failed	121	130	failed	failed	failed	failed	failed

Figure 6. Performance of 3D FFTs in GFlops/s

Power-of-two FFTs: Figure 5(a) shows performance of batched 1D power-of-two SP FFTs. We outperform CUFFT for all sizes by up to 90% for large N and up to 35x for small N. Such a high difference for small N is probably because CUFFT does not implement coalesce transpose. Our auto-tuned implementation outperforms a prior hand-optimized implementation [5] (curve *[5] on GTX280*) by up to 2x by automatically selecting the best-performing kernels and constructing efficient schedules. For $N \in [32, 2048]$, the improvement primarily comes from the optimization of thread block FFT twiddle, while for large N, efficient schedule construction wins over a hand-crafted schedule heuristic. For the previous implementation, there was a significant drop in performance when FFT does not fit into shared memory. This is not the case for the auto-tuned library, because our auto-tuned column kernels are much faster.

Figure 5(b) compares performance across GPUs. For all GPUs, sizes $N \in [2, 2048]$ are done in one pass using row kernels. The computation is bound by memory bandwidth for very small N, and the auto-tuner picks variants that coalesce loads and stores of several small FFTs and rearrange data using a coalesce transpose in shared-memory. Our auto-tuned libraries on GTX280 and GTX260 outperform 8800GTX due to higher compute capabilities and available memory bandwidth. For the same reason, our library on GTX280 outperforms GTX260 for most sizes; however, the performance becomes comparable for very large N. The performance on GTX280 drops faster as size increases due to degradation of effective memory bandwidth for accesses at larger power-of-two strides. The degradation is probably caused by associativity effects, because GTX280 has 8 memory banks, while GTX260 and 8800GTX have 7 and 6 memory banks, respectively. The performance curves are "flatter" for GTX260 and 8800GTX. Our library on GTX280 outperforms *MKL* by $9\times$ on average (up to $15.5\times$).

We also investigated the effect of using a profile from a different card. *GTX280w260prof* shows performance on a GTX280 using a profile obtained on a GTX260. The performance drops significantly for large N. We investigated the situation for $N = 2^{23}$ and found that the optimal schedule predicted for the GTX260 profile is $\langle 64, 64, 32, 64\rangle$; however, radix-64 column kernel does not perform well for large strides on GTX280. The optimal schedule for the GTX280 profile is $\langle 32, 16, 32, 32, 16\rangle$. Interestingly, it uses one more pass over the data, but still delivers higher performance by picking faster kernels.

Figure 5(c) compares the performance of power-of-two DP FFTs for our library, CUFFT and MKL. We outperform CUFFT by up to $25\times$ for very small N due to having better coalescing, $2\times$ for the mid-range due to using specialized row kernels, and 30% for large N. Compared to MKL, our library delivers on average $3.5\times$ higher performance (up to $8\times$).

Mixed Radix FFTs: Figures 5(d) and 5(e) show performance of our library on three GPUs, CUFFT on GTX280 and MKL for batched 1D mixed radix FFTs. On GTX280, our library outperforms CUFFT by up to $12\times$ for large N and by up to $38\times$ for small N. Even on an older-generation 8800GTX GPU, our library outperforms CUFFT3.0 on a GTX280. For CUFFT, mixed radix FFTs are $3\times$ slower than power-of-two FFTs and the performance exhibits large variations. Our auto-tuned mixed radix FFTs perform comparably (within 20%) to power-of-two FFTs and performance curves are mostly flat for large N despite the fact that we approximate kernel performance for a particular stride pair $\langle S_i, S_o\rangle$ by using the closest suitable stride sample. Our library on GTX280 shows $12\times$ higher performance over MKL on average (up to $28\times$).

Figure 5(f) shows performance for mixed radix 1D DP FFTs on a GTX280. We outperform CUFFT by up to $4\times$ for large N and 100 times for small N. The bottleneck for DP FFTs is the performance of double precision arithmetic, especially transcendental functions for twiddle computation. The improvement comes primarily from the twiddle optimizations. We precompute thread block FFT twiddle factors, if any, on the CPU during the FFT schedule creation stage. The twiddle factors are retrieved from constant memory during kernel execution and may be cached in registers. The global twiddle is optimized via strength reduction of sin and cos using trigonometric identities. This improves performance by 2-3 times at the expense of slight decrease in accuracy. Compared to MKL, our library performs on average $3.3\times$ ($4\times$ for large N) times faster (up to $12.5\times$).

Figures 5(g) and 5(h) highlight the accuracy of the FFT algorithms. Our SP accuracy is in the order of 10^{-6} and is up to 6x better than the accuracy of CUFFT FFTs, which behaves erratically. The precision of MKL is in the order of 10^{-7} due to more precise sin and cos. The DP accuracy is in the order of 10^{-16} to 10^{-15}. The DP accuracy of CUFFT is on average 23% higher than ours. We investigated the situation for $N = 900000$ that exhibits the highest difference. The error is introduced solely by the trigonometry strength reduction optimization in the global twiddle computation. If we use exact twiddle factors, the precision is slightly better than that of CUFFT's. However, using CUDA's DP sin and cos decreases performance by a factor of two. We believe that a slight loss of accuracy is an acceptable price to pay for 2-$3\times$ higher performance on GT200-generation GPUs. We expect that the performance of our DP routines will be higher on newer GPUs such as NVIDIA Fermi with better double precision support.

3D FFTs: We measured performance delivered by our auto-tuned library for several 3D FFT sizes to compare with CUFFT v3.0 and numbers published in [10] (AT3dFFT). The performance on an GTX280 is summarized in Figure 6. CUFFT executed correctly for only two sizes. We outperform AT3dFFT by up to 70% due to superior auto-tuning strategy for Y and Z dimensions. AT3dFFT column kernels perform computation entirely in registers, while we present the auto-tuner with an additional option to utilize shared memory. Using shared memory increases the radix and may save extra passes over the data. We investigated the schedule constructed by the library for $256 \times 256 \times 256$ and indeed saw that Y dimension was done in one pass, using radix-256 column kernel. The stride of the Z dimension is too large for radix-256 to perform well, and the library picks 16×16 decomposition. The performance of row kernels for the X dimension is roughly the same. Thus, our library saves one pass over the data resulting in 30% performance improvement.

Limitations: The size of the performance profile may be large because we have to use a lot of specialized precompiled kernels. Reducing the amount of specialization and selecting more frequently used kernels can significantly bring down the profile size at the expense of some performance loss; it is a trade-off. Perhaps, the best way to reduce the profile size is to encode kernel specializations in a few bytes and re-construct the kernel at runtime using a JIT or re-generate the kernel using the GPU compiler.

The performance estimation for a kernel may be inaccurate for the following reasons. First, we auto-tune for a particular data volume so it is possible that the prediction could be inaccurate for a different volumes, though we did not notice performance artifacts for different volumes. If this becomes an issue, it is possible to auto-tune for several representative data volumes. Second, we sparsely sample the stride space and assume that performance changes relatively smoothly. Third, we noticed that sometimes a sequence of kernels runs slower than the sum of runtime estimates for stand-alone kernel runs. Using the CUDA profiler, we found that performance of a kernel may be affected by another kernel executed earlier. This is quite unexpected and we attribute this to a bug in the GPU runtime.

Our FFT library assumes that data resides entirely within GPU memory. Our auto-tuning framework does not account for memory transfers and out-of-core FFT computations.

8. Conclusion and future directions

We presented an auto-tuning framework for automatically generating optimized FFT kernels on GPUs. Our pruning heuristics significantly reduce the optimization search space. Our runtime algorithm is used for efficient schedule execution. We demonstrated the performance improvements of our algorithms on three different GPUs on a wide range of size distributions in both single and double precision, and compared with prior state-of-the-art algorithms on CPUs and GPUs. Our algorithms achieve significant performance improvements while maintaining high accuracy.

There are several avenues for future work. We would like to extend our algorithms to other compute APIs and GPUs including the AMD 5800 series and the NVIDIA Fermi architecture. Another avenue for our work is to extend the auto-tuning framework to support DCT and DST transformations. Our current implementation does not account for data transfers and it would be interesting to extend our algorithms for data sets that do not fit in GPU memory.

References

[1] S. Chellappa, F. Franchetti, and M. Püeschel. Computer generation of fast Fourier transforms for the cell broadband engine. In *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[2] N. Corp. CUDA occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, 2010.

[3] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[4] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006.

[5] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2008.

[6] 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.

[7] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. Technical Report UT-CS-09-635, Massachusetts Institute of Technology, May 2009. LAPACK Working Note 212.

[8] J. L. Mitchell, M. Y. Ansari, and E. Hart. Advanced image processing with DirectX 9 pixel shaders. In W. Engel, editor, *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware Publishing, Inc., 2003.

[9] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2003.

[10] A. Nukada and S. Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[11] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2008.

[12] NVIDIA Corp. *NVIDIA CUDA Programming Guide*, 2009.

[13] M. Pschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, M. Pschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, 18:21–45, 2004.

[14] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the international symposium on Code generation and optimization*, 2008.

[15] J. Spitzer. Implementing a GPU-efficient FFT. *SIGGRAPH Course on Interactive Geometric and Scientific Computations with Graphics Hardware*, 2003.

[16] T. Sumanaweera and D. Liu. Medical image reconstruction with the FFT. In M. Pharr, editor, *GPU Gems 2*, pages 765–784. Addison-Wesley, 2005.

[17] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* Society for Industrial Mathematics, 1992.

[18] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture, 2008. `http:www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf`.