Towards a Performance-Portable FFT Library for Heterogeneous Computing*

Carlo del Mundo* and Wu-chun Feng*†
NSF Center for High-Performance Reconfigurable Computing
*Department of Electrical & Computer Engineering
†Department of Computer Science
Blacksburg, VA, USA.
Virginia Tech
{cdel, wfeng}@vt.edu

ABSTRACT

The fast Fourier transform (FFT), a spectral method that computes the discrete Fourier transform and its inverse, pervades many applications in digital signal processing, such as imaging, tomography, and software-defined radio. Its importance has caused the research community to expend significant resources to accelerate the FFT, of which FFTW is the most prominent example. With the emergence of the graphics processing unit (GPU) as a massively parallel computing device for high performance, we seek to identify architecture-aware optimizations across two different generations of high-end AMD and NVIDIA GPUs, namely the AMD Radeon HD 6970 and HD 7970 and the NVIDIA Tesla C2075 and K20c, respectively, to accelerate FFT performance.

Despite architectural differences across GPU generations and vendors, we identify the following optimizations, when applied individually and in isolation of one another, as being the most effective in accelerating FFT performance: (1) register preloading, (2) transposition via local memory, and (3) 8- or 16-byte vector access and scalar arithmetic. We then demonstrate the efficacy of combining individual optimizations together and find that the most effective combination of optimizations across all architectures encompasses register preloading, transposition via local memory, and use of constant memory. Our study suggests that FFT performance on GPUs is primarily limited by global memory data transfer. Overall, our optimizations deliver speed-ups as high as 31.5 over a baseline GPU implementation and 9.1 over a multithreaded FFTW CPU implementation with AVX vector extensions.

1. INTRODUCTION

The FFT has been identified as a key computational idiom for present and future applications and is central across a wide range of fields such as cognitive radio, digital signal processing, and encryption [2, 13, 18, 20]. Its importance has caused the research community to expend significant resources to accelerate the FFT, of which FFTW is the most prominent example. The constant demands for high performance, however, have caused a shift towards accelerator-based processors such as GPUs to further improve FFT performance. Recent work has demonstrated substantial speedups for FFT on GPUs [3–5, 7, 8, 10–12, 15, 16, 19]. To magnify the gap between CPU and GPU FFT performance, we surveyed several GPU FFT libraries in addition to FFTW. Figure 1 compares the performance across vendor libraries (AMD APPML and NVIDIA CUFFT) and vendor-neutral codes (AppleFFT, FFTW) for commodity CPU and GPU hardware.

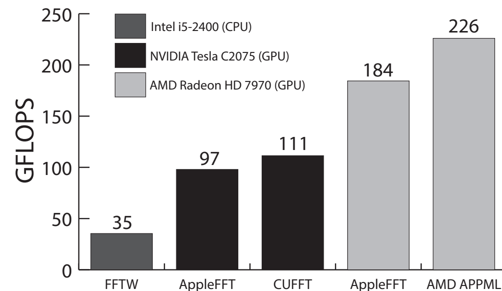

Figure 1: Survey of FFT libraries for state-of-the-art CPU and GPU hardware for a single-precision batched 1D 16-pt FFT for 128 MB. CUFFT and AMD APPML are vendor provided libraries for NVIDIA and AMD GPUs, respectively. AppleFFT is an open-source FFT library designed for NVIDIA GPU architectures exclusively.

Vendor GPU libraries fare well for their respective architectures, while vendor-neutral libraries such as AppleFFT (which was designed for the NVIDIA GPU architecture) provide superior performance on NVIDIA cards and reasonable performance for AMD. In short, GPU-accelerated FFT codes outperform FFTW-based solutions by factors as high as 6.8.

To address the need for high performance while maintaining portability to any device, this work seeks to develop an architecture-agnostic FFT library, similar to FFTW. To

*This work was supported in part by NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CF '14, May 20 - 22 2014, Cagliari, Italy
Copyright 2014 ACM 978-1-4503-2870-8/14/05$15.00.
http://dx.doi.org/10.1145/2597917.2597943.

date, very little work has focused on the portable performance of FFT across heterogeneous processors. In order to develop such a library, (1) a multi-dimensional characterization of optimizations and their interactions is necessary to harness the computational power of a target architecture, and (2) an auto-tuning framework is required to empirically determine optimal cutoff values for sweepable parameters.

Related work has demonstrated the efficacy of auto-tuning FFT on GPUs but lack rigorous characterization of the optimizations and their effects on machine-level behavior. Therefore, we seek to identify optimal optimization sequences for FFT across two generations of AMD and NVIDIA GPUs. The contributions of our work are as follows:

- Optimization principles for FFT on GPUs
- An analysis of GPU optimizations applied in isolation and in concert on AMD and NVIDIA GPU architectures

Due to radical architectural differences across GPU generations and vendors, we expected a diverse set of optimizations per target architecture. However, our results indicate that one unique optimization sequence is most effective in accelerating FFT performance: (1) register preloading, (2) transposition via local memory, and (3) 8 or 16-byte vector access and scalar arithmetic. We then demonstrate the efficacy in combining certain optimizations in concert with register preloading, transpose via local memory, and use of constant memory being the most effective for all architectures. Our study suggests that after extensive optimization, performance of FFTs on graphics processors is primarily limited by global memory data transfer.

The rest of this paper is summarized as follows. Section 2 provides an overview of FFT, OpenCL, and terminology we have adopted in this paper. Section 3 presents the optimizations that we have applied to the GPU cores. Section 4 summarizes and discusses our results. Section 5 discusses related work. Finally, Section 6 presents our conclusions.

2. BACKGROUND

2.1 The Fast Fourier Transform (FFT)

The FFT is part of a family of computations known as spectral methods. A spectral method transforms data from continuous time and space to an equivalent discrete form. Spectral method computations are characterized by multiply-add operations known as butterfly computations. The communication pattern requires local or global all-to-all synchronization between executing units depending on the transform size. The FFT is the canonical spectral method, but many other transforms exist. Improvements in one method will ultimately lead to improvements for the whole family of spectral methods.

Our mapping strategy is based on the Cooley-Tukey framework, where an N-pt FFT is arranged as size $N_1 \times N_2$. We apply a variant of the canonical four-step method [10, 19]. Assuming data is in row-major order, the four steps are: (1) FFT on columns, (2) twiddle multiplication, (3) transpose, and (4) FFT on columns. Figure 2 shows an example of the four-step method applied to a 16-pt FFT.

FFT on Columns. The original N-pt FFT is represented as a $N_1 \times N_2$ matrix in row-major order. N_2 radix-N_1 FFTs are performed on the columns. These subtransforms of radix-N_1 FFTs are calculated directly by each thread using a decimation-in-frequency approach.

Twiddle Multiplication. Twiddle multiplication is characterized by point-wise multiplication of all input elements by a predefined constant, or twiddle factor. The twiddle factor at row i and column j is $e^{\frac{-2\pi ij}{N}}$.

Transpose. The transpose stage represents an all-to-all synchronization between threads in a single batch. Elements must exchange data along the main diagonal. Formally, for each element in matrix X, X_{ij} must swap with element X_{ji}.

2.2 OpenCL Programming Model & Terminology

OpenCL is a high-level API specification for programming heterogeneous processors. OpenCL's strength is functional portability; users simply write code once and execute on a highly diverse set of processors from CPUs to FPGAs. Its generality and applicability as a vendor-agnostic language allows for execution across processors; however, performance is not necessarily portable. Since OpenCL accommodates a broad set of devices, vendor support for special instructions is limited. Most notably, features in CUDA 5.0 such as the shuffle intrinsic, an instruction for intra-wavefront exchange of register data, are not supported in OpenCL. We use the following OpenCL terminology throughout this work. A work item is an autonomous unit of execution on the GPU. Work items are partitioned into work groups. Work groups are coarse units for scheduling into compute units (CU). Architecturally, CUs contain groups of many cores known as processing elements (PE). Finally, a wavefront is the atomic unit of execution for the GPU. In this work, wavefronts from NVIDIA and AMD architectures execute in lockstep.

OpenCL contains several memory spaces: global, texture, constant, local, private. We describe these memory spaces in the context of a GPU. Global memory is a slow DRAM readable and writable by any CU. Texture memory is cache-accelerated global memory for memory textures. The constant memory is a small, cached memory for frequently used data. Local memory[1] is a fast, on-chip scratchpad memory for work items to coordinate. Finally, private memory is on-chip registers.

3. APPROACH

In the context of the FFT, our work seeks to uncover architectural insights on two generations of NVIDIA and AMD GPUs via optimizations applied in isolation and in concert. The FFT was evaluated in three sample sizes: 16-, 64-, and 256-points which naturally fit the capacity of on-chip memories such as registers and local memory. We then evaluated a larger, out-of-core 2D FFT of size 256×256 to validate our optimization strategies for FFTs beyond the capacity of on-chip memories. Finally, we evaluated shuffle, a poorly understood mechanism endemic to NVIDIA Tesla Kepler GPUs that allows for intra-warp communication without the use of shared memory. All implementations are single-precision floating point and validated with a double-precision CPU code.

[1]OpenCL local memory should not be confused with CUDA local memory which is a slower, intermediate memory for register spills in CUDA architectures.

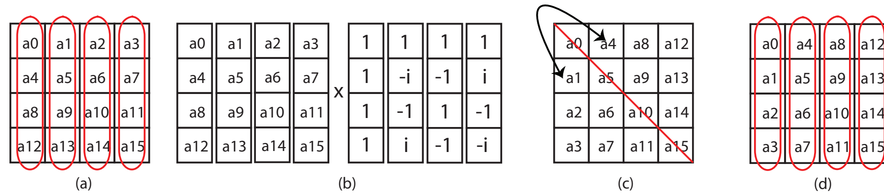

Figure 2: The four-step method applied to a 16-pt FFT. Input is organized in row-major order. In (a), 4 radix-4 FFTs are performed on columns. The twiddle multiplication stage is shown in (b) where a point-wise multiplication is performed on each element of the input array. The transpose step occurs in (c) where elements across the diagonal are exchanged. An arrow depicts one swap between element a1 and a4. Finally, FFT on columns are performed again in (d). The output is the FFT of the original array, organized in row-major order.

3.1 Baseline Implementation

A baseline kernel represents an unoptimized, naive kernel typically implemented as a first resort for evaluating the efficacy of an algorithm on the GPU. Our optimizations are applied relative to the baseline kernel. We systematically apply optimizations one by one to the baseline kernel (e.g., optimizations in isolation) to gain insight on how each optimization interacts with machine-level behavior. We then apply optimizations in combination (e.g., optimizations in concert) based on the insights gained from the results in isolation. It is important to note that the baseline kernel is configured to (1) utilize all GPU cores by computing multiple transforms, (2) performs 8-byte vector access, scalar math (VASM2), and (3) performs all computation and communication operations on global memory. We discuss the optimizations applied to the baseline kernels for 16-, 64-, and 256-points in the 1D case and 256×256 in the 2D case.

3.2 Shuffle (Transpose) Optimization

Architecture-aware optimizations target specific hardware mechanisms of a processor. For instance, NVIDIA introduced a novel mechanism for register-to-register exchange within work items in a wavefront in their latest Tesla K20c GPU. Traditionally, communicating data between work items require a shared, scratchpad memory. This new mechanism, known as shuffle, allows intra-wavefront register exchange without using local memory. Since the FFT requires a communication step between work items akin to matrix transpose, we leveraged shuffle to eliminate the need for scratchpad local memory. Our shuffle algorithm which performs matrix transpose entirely on registers is as follows.

Given a NxN matrix, where each thread, t_i contains one column of the matrix. Then, for each thread t_i for $0 \leq i < N$, perform the following steps:

1. **Horizontal Rotation**. Perform data rotation row-wise via inter-thread shuffle. Data in row k of column i will be assigned to row data from $(i+N-k) \mod N$.

2. **Vertical Rotation**. Perform data rotation column-wise within a thread. Data in row k of column i will be assigned to column data from $(i+k) \mod N$.

3. **Horizontal Rotation**. Perform data rotation row-wise via inter-thread shuffle. Data in row k of column i will be assigned to row data from $(N+k-i) \mod N$.

Figure 3 illustrates our shuffle algorithm. Steps 1 and 3 (horizontal rotation) require inter-thread communication using the shuffle mechanism, and step 2 (vertical rotation) requires movement within a thread's own register file. We will demonstrate later that effective use of shuffle hinges on the treatment for the vertical rotation stage. The primary issue is *a priori* indexing [6]. The compiler must know the source and destination indexing at compile time; if not, the compiler is forced to allocate a region of memory known as CUDA local memory which is significantly slower than GPU registers. We will further address this issue in the results.

3.3 System-level Optimizations

A system-level optimization is applicable to any application. The following is a list of system-level optimizations that have been applied.

Run Many Work Items. The GPU is a massively multi-threaded throughput machine. A large number of work items must be launched in order to hide long-latency global memory operations. Wavefronts can context switch during a stall for global memory transactions. An abundance of wavefronts can effectively hide memory latency. We apply this optimization through batching. All implementations process 128 MB of single-precision transforms over application execution. This size is enough to saturate all compute units in our experimental testbed.

Use On-Chip Memory. The aggregate bandwidth for on-chip memory is roughly two orders of magnitude higher than global memory. Table 1 lists the read bandwidths for on-chip and global memories for Radeon HD 6970. Furthermore, effectively staging computation and communication operations in on-chip memory significantly improves application performance because it avoids multiple passes in long-latency global memory. Additional on-chip memories such as the limited capacity local data share provide scratchpad memory visible to all work items in a work group. Our optimized versions make use of the register file (RP) and local memory (LM) for computation, communication, or both.

Table 1: Memory Read Bandwidth for Radeon 6970

Functional Unit Read	Bandwidth (TB/s)
Registers	16.2
Constant Memory	5.4
Local Data Share	2.7
Global Memory	0.17

Organize Data In Memory. Efficient data layouts are essential for optimized memory transactions. A reorganiza-

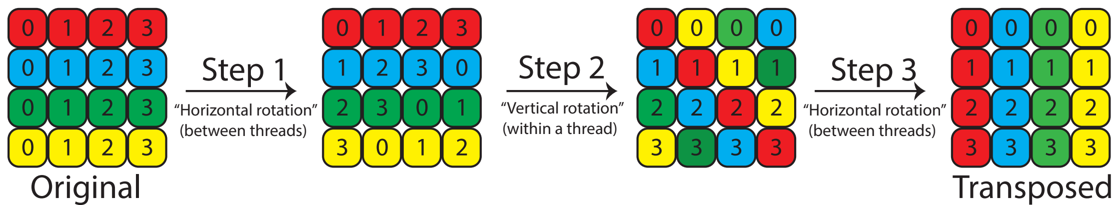

Figure 3: Our shuffle algorithm applied to a 16-pt FFT. The FFT's data communication stage (matrix transpose) is performed entirely in registers

.

tion in memory can yield more efficient transaction transfers in hardware. A worst-case scenario occurs when irregular access patterns result in many small incongruous transactions. In our unoptimized implementation of sample size N, sets of $\sqrt{N}$ threads access memory contiguously. However, the stride between sets of threads is N which results in poor accesses. Our optimized version (CGAP) reorganizes memory for coalesced accesses. The kth thread accesses memory element k in a wavefront.

Reduce Dynamic Instruction Count. While it is traditionally important to reduce dynamic instruction count, there is no guarantee that the GPU compiler will automatically apply these optimizations. Some examples include common subexpression elimination, loop unrolling, and function inlining. We take advantage of these optimizations by explicitly programming them in our kernels. We performed loop unrolling (LU) for memory loads and stores, inlined (IL) kernel helper functions, and reordered computation to eliminate common subexpressions (CSE).

Use Constant Memory. Constant memory provides cached performance and lower latency relative to device global memory. For commonly accessed data elements, the cached constant memory is an excellent candidate for performance improvement. Constant memory is limited in capacity, so it is best suited for a small set of frequently accessed data. We implemented both kernel argument (CM-K) and kernel literal (CM-L) optimizations for constant memory [1].

Use Vector Types. Data layout in FFT is represented in interleaved or planar format. Elements are ordered as a float2 {real, imaginary} pair in the interleaved format, while real and imaginary parts are allocated in separate float arrays for the planar format. The interleaved format allows for vector access, scalar math (VASM), while the planar format allows for vector access, vector math (VAVM). Ideally, the planar format should be used in AMD VLIW architectures as this significantly improves the co-issue packing size at compile time by providing increased instruction-level parallelism.

Our baseline implementation uses an interleaved float2 format (VASM2). We vary the vector size in powers of two for both interleaved and planar. Optimized implementations use vector access/scalar math (VASM) or vector access/vector math (VAVM). Table 2 depicts the local memory usage as a function of on-chip memory implementation and vector size. The increased local memory pressure for larger sample sizes decreases the effectiveness of larger vector types (VAVM4, VAVM2, and VAVM4). Local memory implementations (LM-CC and LM-CT) exhaust the local memory file thereby reducing the number of active threads in flight.

Table 2: Local Memory Usage per Transform for Each On-chip Optimization for Sample Size, N

	RP+LM-CM	LM-CC	LM-CT
VASM2	N sz(float)	N sz(float2)	N sz(float2)
VASM4	N sz(float)	N sz(float4)	N sz(float4)
VAVM2	N sz(float)	$2N$ sz(float2)	$2N$ sz(float2)
VAVM4	N sz(float)	**2N sz(float4)**	**2N sz(float4)**

3.4 Algorithm-level Optimizations

Problem decomposition, assignment, mapping, and orchestration is algorithm dependent. Our decomposition, assignment, and mapping have been detailed previously (§ 2.1). However, orchestration of computation and communication is staged in on-chip memory in a variety of ways.

We split our on-chip memory implementations in three distinct categories: (1) computation on register file, communication on local memory {RP + LM-CM}, (2) computation on local memory, communication in register file {LM-CC}, and (3) computation on local memory, no communication {LM-CT}. The third implements a technique mentioned in the work of Volkov and Kazian where a property of the Cooley-Tukey framework is exploited [19]. The transpose step is completely eliminated by rearranging the data in column-major order. Radix-$\sqrt{N}$ FFTs and twiddle multiplication stages are first applied on the rows followed by radix-$\sqrt{N}$ FFTs on columns. The transformed output is in column-major order.

4. RESULTS AND DISCUSSION

Here, we provide results and analytical insights for FFT. Although there are many points of discussion with the results, we focus only on the salient aspects of FFT. Whenever possible, we derive metrics to highlight aspects of machine-level behavior.

4.1 Experimental Testbed and Optimizations

Table 3 depicts the devices used in this study. The Radeon HD 7970 and the Tesla K20c are the latest GPUs from AMD and NVIDIA, respectively, and the Radeon HD 6970 and Tesla C2075 are previous generations, respectively. One notable exception in architecture is the VLIW pipeline present in Radeon HD 6970. In VLIW processors, the burden is on the compiler to find co-issue opportunities within kernel code. In Radeon HD 6970, a VLIW instruction is comprised of four independent microinstructions. If there is insufficient instruction-level parallelism for an application, execution units for VLIW processors go idle. For comparison with a multi-core CPU, we have included a quad-core Intel i5-2400 CPU running FFTW version v3.3.2. FFTW was con-

Table 3: Experimental Testbed

Device	Cores	Peak Performance (GFLOPS)	Global Memory Bandwidth (GB/s)	Register Size per CU (kB)	LDS Size per CU (kB)	Core Clock (MHz)	Memory Clock (MHz)	Max TDP (Watts)
Radeon HD 6970 GPU	1536	2703	176	256	32	880	1375	250
Radeon HD 7970 GPU	2048	3788	264	256	64	925	1375	250
Tesla C2075 GPU	448	1288	144	32	48	1147	1666	225
Tesla K20c GPU	2496	4106	208	64	48	705	2600	225

Table 4: List of Optimizations Applied to FFT

Codename	Name	Description
LM-CT	Local Memory (Compute, No Transpose)	Data elements are loaded into local memory for computation. The communication step is avoided by algorithm reorganization [19].
LM-CC	Local Memory (Compute, Communicate)	All data elements are preloaded into local memory. Computation is performed in local memory, while registers are used for scratchpad communication.
LM-CM	Local Memory (Communicate Only)	Data elements are loaded into local memory only for communication. Threads swap data elements solely in local memory. This optimization requires only $N\times$ sz(float) local memory by transposing each dimension of a floatn vector one dimension at a time.
CM-{K,L}	Constant Memory {Kernel, Literal}	The twiddle multiplication stage can be pre-computed on the host and stored in constant memory for fast look up. This saves two transcendental single-precision operations at the cost of a cached memory access. CM-K refers to the usage of constant memory as a kernel argument, while CM-L refers to a static global declaration in the OpenCL kernel.
RP	Register Preloading	All data elements are first preloaded onto the register file of the respective GPU. Computation is facilitated solely on registers.
CGAP	Coalesced Global Access Pattern	Threads in a wavefront access memory contiguously, e.g. the kth thread accesses memory element, k.
CSE	Common Subexpression Elimination	A traditional optimization that collapses identical expressions in order to save computation. This optimization may increase register live time, therefore, increasing register pressure.
IL	(Function) Inlining	A function's code body is inserted in place of a function call. It is used primarily for functions that are frequently called.
LU	Loop Unrolling	A loop is explicitly rewritten as an identical sequence of statements without the overhead of loop variable comparisons.
VASM {2,4,8,16}	Vector Access Scalar Math float{2,4,8,16}	Data elements are loaded as the listed vector type. Arithmetic operations are scalar (float × float).
VAVM {2,4,8,16}	Vector Access Vector Math float{2,4,8,16}	Data elements are loaded as the listed vector type. The arithmetic operations are vectorized with the sizes listed, (floatn × floatn).
SHFL	Shuffle	The transpose stage in FFT is performed entirely in registers eliminating the use of local memory. This optimization is only specific to NVIDIA Tesla K20c

figured to utilize four threads on OpenMP with explicit AVX extensions. FFTW was compiled using gcc v4.4.5. We apply optimization listed in Table 4 both in isolation and in concert. All results were collected using OpenCL kernel event timers. For our AMD GPUs, we used Radeon driver v12.10 on a 64-bit Windows 7 machine using AMD APP SDK v2.7. For NVIDIA GPUs, we used NVIDIA driver 304.54 on a Debian Linux machine with kernel 2.6.37.2. Each implementation processes 128 MB of data, and the average of 1000 kernel iterations was collected.

4.2 Optimizations in Isolation

Figure 4 shows our results in isolation for each stage of FFT. We applied all optimizations in Table 4 with the exception of shuffle which is evaluated in concert with other optimizations.

Timing Methodology. Each data point in Figure 4 is the summation of each stage, e.g. $t_{isolation} = t_{cols} + t_{twiddles} + t_{transpose}$. Each stage is a separate kernel invocation and the sum of all kernel execution time was included.

Trends across FFT stages (in isolation). In general, the execution time (from greatest to least) is columns, transpose, and twiddles. Optimizations that targetted each stage specifically was LM-CM and CM-K/-L. Both optimizations were not effective in isolation.

Trends across GPU architectures (in isolation). NVIDIA GPUs achieves substantial performance even without explicit register preloading and local memory usage; in contrast, AMD GPUs are critically dependent on applying these optimizations for high-performance. In general, the efficacy of vector math operations (VAVM) are improved with the VLIW architecture of the Radeon HD 6970, but these improvements are meager compared to scalar math operations. Vector math operations are detrimental to the scalar GPU architectures (Radeon HD 7970, NVIDIA Tesla C2050, NVIDIA Tesla K20c).

Global memory bus traffic is the largest performance limiter for AMD GPUs. We define *global memory bus traffic* as the number of bytes transferred from off-chip device memory to on-chip memory. We will refer to this as "bus traffic". The optimal bus traffic is the minimum number of memory load and store operations issued for a kernel. Factors such as uncoalesced memory accesses, register spills to device memory, and CUDA local memory allocation may increase bus traffic. Bus traffic of each kernel was calculated using the following formula.

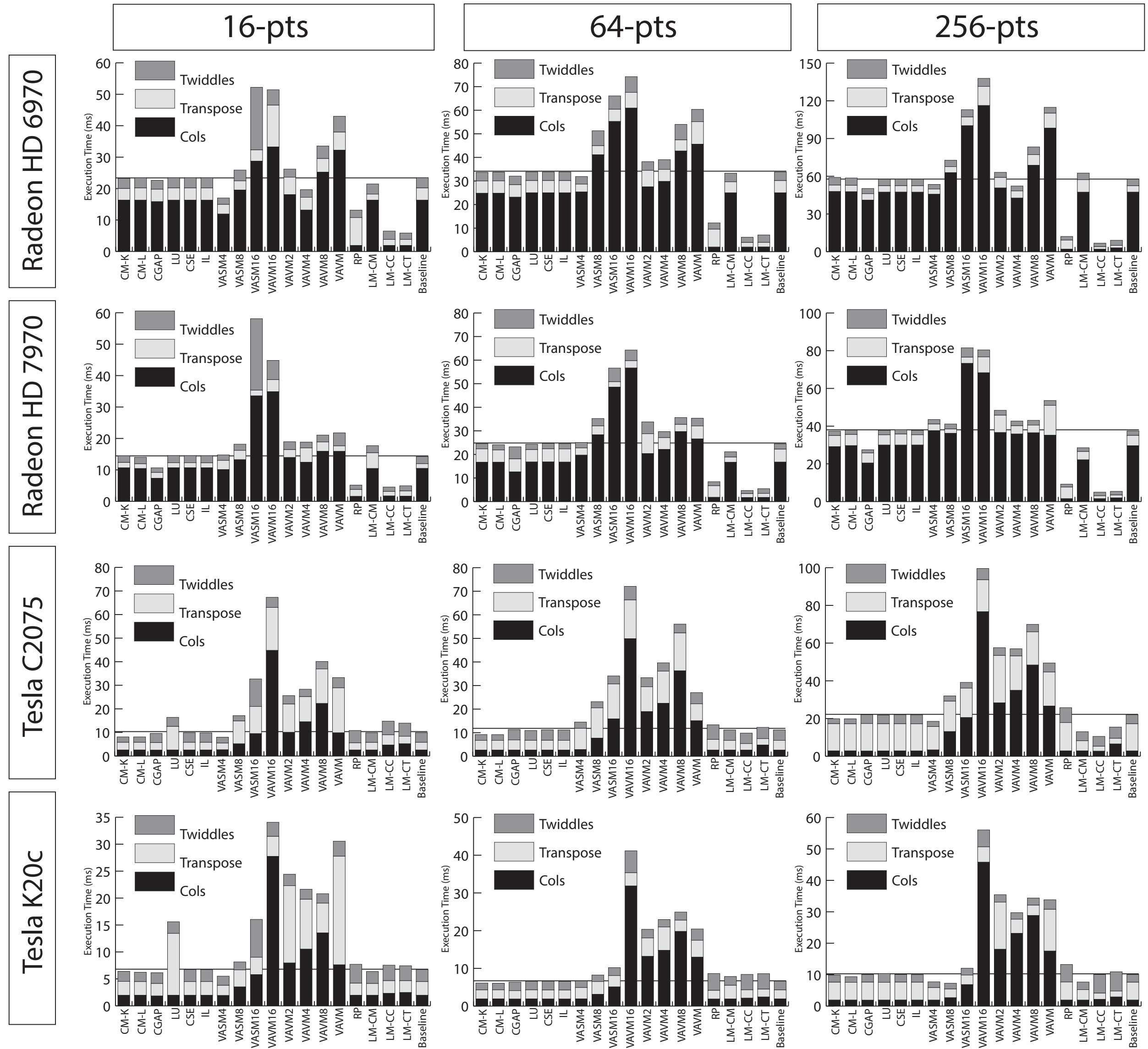

Figure 4: Optimizations applied in isolation to a baseline, unoptimized GPU kernel for for 16-, 64-, and 256-pts on Radeon HD 6970, Radeon HD 7970, Tesla C2075, and Tesla K20c. Comparisons should only be made within a sample size within an architecture.

$$\text{Bus Traffic (MB)} \quad = 2^{-20} \times (bytes_{loaded} + bytes_{stored})$$

where $bytes_{loaded}$ and $bytes_{stored}$ refer to the minimum number of bytes required to load and store, respectively.[2] This number can be determined statically. For FFT with a problem size of 128 MB, the minimum bytes loaded and stored for the columns, transpose, and the twiddles stage are 256 MB, 256 MB, and 240 MB, respectively.[3]

[2]In AMD architectures, the profiler counters FetchSize and FastPath are the total number of kilobytes fetched and stored in global memory, respectively.

[3]240 MB for the twiddles stage since elements in the first row and column are multiplied by unity.

Table 5 depicts the bus traffic of optimizations in isolation for a 256-pt FFT on AMD Radeon HD 7970. Performance of AMD architectures is directly related to the total bus traffic. Minimizing bus traffic was achieved through three on-chip memory optimizations (RP, LM-CC, LM-CT). These optimizations reduce bus traffic by prefetching off-chip device memory to on-chip resource with all computation and communication operations computed entirely on-chip. A linear regression (not shown) for this data comparing performance and bus traffic yield a strong correlation ($R^2 = -0.99$).

Trends across optimizations (in isolation). VASM2 (baseline) and VASM4 are the optimal vector implementations with potential improvements in all architectures. We do not consider vector sizes larger than 16 bytes. In par-

Table 5: Global memory bus traffic for results in isolation for FFT256 in AMD Radeon HD 7970. This table shows the number of bytes transferred (in megabytes) and the relative traffic (x) compared with the ideal bus traffic. The bolded cells depict optimal bus traffic.

FFT256	CM-K	CM-L	CGAP	LU,CSE,IL	VASM4	VASM8	VASM16	VAVM16	VAVM2	VAVM4	VAVM8	VAVM	RP	LM-CM	LM-CC	LM-CT	Baseline
Cols	2744	2748	2795	2745	3444	4320	7615	6496	3290	3380	3877	3344	**255**	2262	**255**	**255**	2744
(MB / x)	11x	11x	11x	11x	13x	17x	30x	25x	13x	13x	15x	13x	**1x**	9x	**1x**	**1x**	11x
Transpose	623	653	642	623	392	277	305	490	658	429	337	1189	783	747	**255**	**255**	661
(MB / x)	2x	3x	3x	2x	2x	1.08x	1.19x	2x	3x	2x	1.31x	5x	3x	3x	**1x**	**1x**	3x
Twiddles	**239**	**239**	**239**	**239**	**239**	285	382	337	**240**	**239**	274	**239**	255	**239**	255	255	**239**
(MB / x)	**1x**	**1x**	**1x**	**1x**	**1x**	1.19x	1.59x	1.40x	**1x**	**1x**	1.14x	**1x**	1.06x	**1x**	1.06x	1.06x	**1x**

ticular, VAVM16 is the worst vector access and arithmetic type.

CGAP generally improves performance across all architectures. The efficacy of this optimization is clearer when combined later with on-chip implementations. CSE/IL/LU optimizations have little to no effect to the baseline for each architecture, and we no longer consider these optimizations in concert. There is little difference between CM-K and CM-L. Constant memory is not as effective in isolation due to computation on the global memory. In concert with on-chip memory implementations, using constant memory for the twiddle calculation is helpful by saving two transcendental operations and a floating point multiplication for a cached global memory access

4.3 Optimizations in Concert

Figures 5 depict optimizations applied in concert for Radeon HD 6970, Radeon HD 7970, NVIDIA Tesla C2075, and NVIDIA Tesla K20c. We varied on-chip optimizations, and vector types. All implementations are coalesced (CGAP) and make use of constant memory (CM-K).

Timing Methodology. We timed both the kernel compute time and the kernel memory load and store time. Each data point is calculated using the following scheme:

$$t_{\text{concert}} = t_{\text{compute}} + t_{\text{mem}}$$

$$t_{\text{compute}} = \begin{cases} t_{\text{overall}} - t_{\text{mem}}, & \text{if } t_{\text{overall}} - t_{\text{mem}} > 0 \\ 0, & \text{if } t_{\text{overall}} - t_{\text{mem}} < 0 \end{cases}$$

where t_{compute} is the time it takes to perform the computation only (excluding memory transfers), t_{mem} represents the time it takes to transfer from off-chip global memory to on-chip resources and back, t_{concert} is the actual plotted value in Figure 5, and t_{overall} is the time it takes to perform the computation and global memory loads to on-chip resources and back. We acknowledge that register allocations will fluctuate between t_{mem} and t_{overall} which could potentially decrease occupancy (and skew execution time). Nevertheless, this timing methodology is here to illustrate the relative differences between kernel execution and global memory load and store time.

Trends across GPU architectures (in concert). First, the VLIW pipeline of Radeon HD 6970 handles vector access vector math (VAVM) computations more efficiently than the scalar pipelines present in the rest of the GPUs. The VAVM optimization showed an increase in the VLIW packing ratio of Radeon HD 6970 compared to VASM optimizations.

We note that global memory loads and stores contribute a majority of the overall execution time for all sample sizes on all architectures. In the most optimal implementations, the computation is completely overlapped with memory transfers. Therefore, the algorithm becomes memory bound and architectures with higher global memory bandwidth determines the overall performance of the FFT. The average achieved global memory bandwidth for implementations in concert for the Radeon HD 6970, HD 7970, Tesla C2075, and Tesla K20c are 136 ± 5, 183 ± 11, 94 ± 13, and 139 ± 20 GB/s, respectively.

Trends across optimizations (in concert). RP + LM-CM consistently provided best performance improvement across all devices in all vector types. RP + LM-CM is the most efficient on-chip implementation in terms of local memory usage as shown in Table 2. The transpose step is unrolled for each component of the vector saving precious local memory space by a factor related to the vector size.

In isolation, constant memory provided little to no performance improvement, however, our analysis in concert indicates that computing the twiddle stage on-the-fly (via explicit transcendental calculations) introduced kernel execution overhead. Applying constant memory optimizations (CM) eliminated this overhead and improved performance for all sample sizes.

The best combination makes use of the following optimizations: register preloading and transpose via local memory (RP+LM-CM), coalescing global access (CGAP), and use of constant memory. The optimal vector size for all architectures is vector access, scalar math (VASM) with 8 or 16-byte words. Finally, computing values on local memory (LM-CC, LM-CT) introduce overheads relative to RP and the differences between the two optimizations are negligible.

4.4 Shuffle Optimization on K20c

Figure 6 depicts our results for the shuffle mechanism for a 256-pt FFT on NVIDIA Tesla K20c. Communication (shown in gray) refers to the local transpose stage within an FFT, while computation refers to stages in the FFT that require arithmetic (e.g., twiddles and FFT on columns). **LM-CM** represents a point of reference in which local memory is used to communicate data between threads. **Naive** represents our initial shuffle code. We discovered that the **naive** implementation suffers from heavy usage of CUDA local memory, a slower memory region typically allocated in the L1 cache or global memory. To mitigate this is-

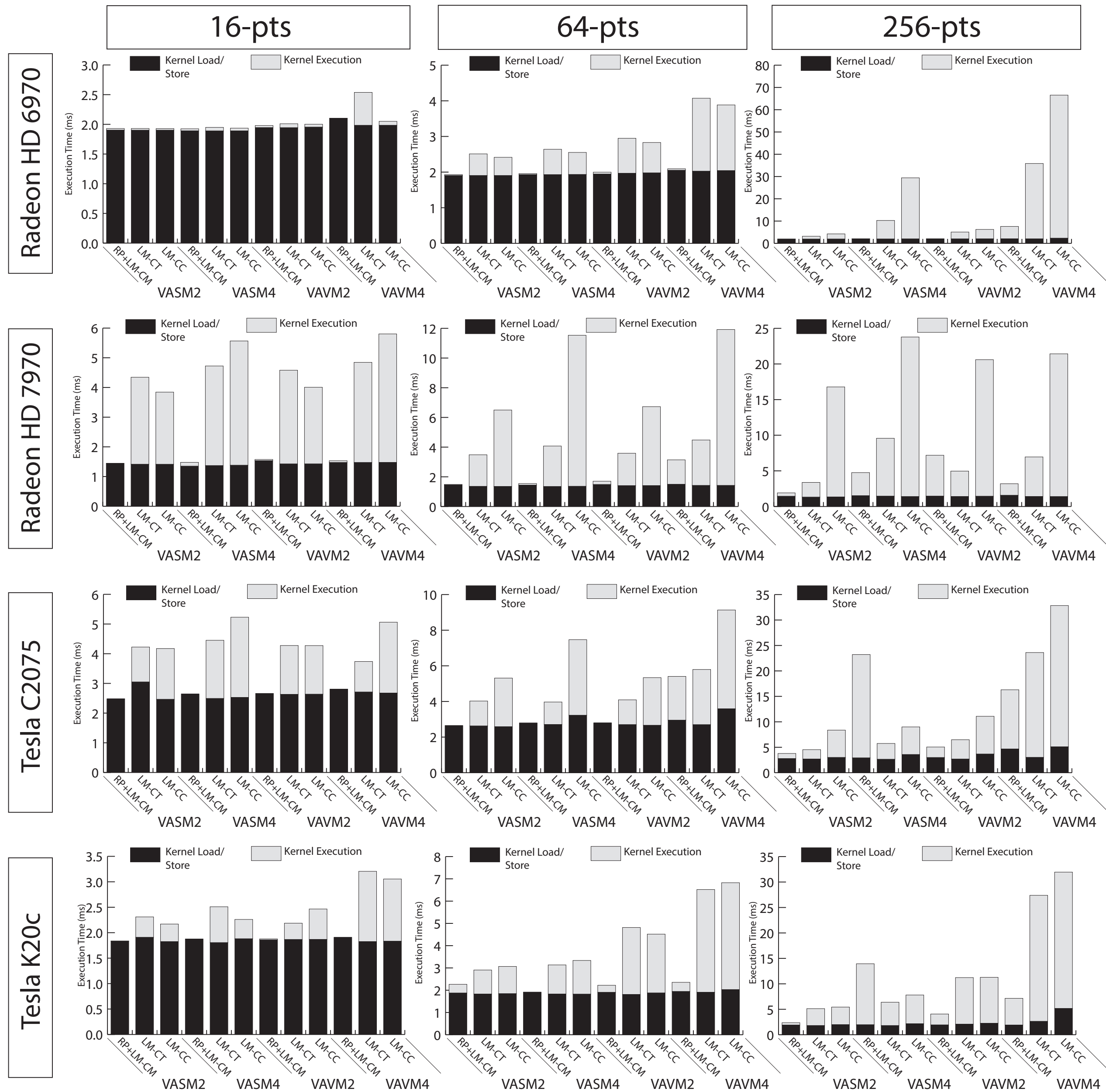

Figure 5: Optimizations applied in concert to a baseline, unoptimized GPU kernel for for 16-, 64-, and 256-pts on Radeon HD 6970, Radeon HD 7970, Tesla C2075, and Tesla K20c. Note: coalesced global access pattern (CGAP) and constant memory kernel literal (CM-K) was applied to the data points listed. Comparisons should only be made within a sample size within an architecture.

sue, we introduced code divergence (e.g., **DIV**) to ensure *a priori* indexing (see § 3.2). Divergence is typically considered bad programming practice, but the cost of divergence is subsumed by the greater cost of CUDA local memory allocation and usage. Application of the CUDA selection and comparison PTX instruction, an instruction that allows for predicated stores of data values, in **SELP** reduced the divergence associated with **DIV**. SELP was applied entirely in-place (e.g., **IP**) or out-of-place (e.g., **OOP**). **SELP IP** uses the least number of registers than **LM-CM** allowing it to execute at a much higher occupancy rate yielding an aggregate improvement in both computation and communication. Overall, our shuffle optimization improves the performance of FFT by an additional 1.17-fold.

4.5 2D FFT

To demonstrate the portability of our optimizations for larger FFT sizes, we evaluated an out-of-core FFT that exceeds on-chip memory capacity. We applied the best combination of optimizations from the 1D case and applied it

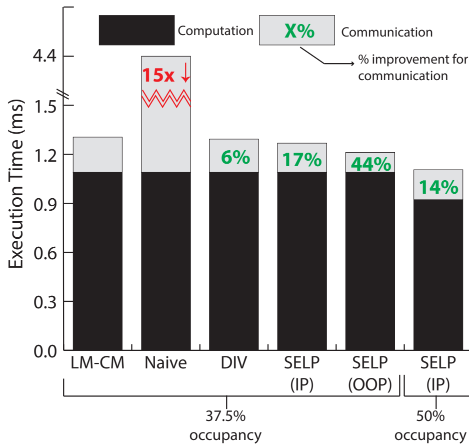

Figure 6: Results for shuffle mechanism applied to a 256-pt FFT.

to a 2D FFT of size 256×256 and juxtaposed its unoptimized, baseline counterpart. *Optimized* refers to (1) register preloading, (2) transposition via local memory, (3) coalesced global memory accesses, (4) 8-byte vector access and scalar arithmetic, and (5) constant memory usage, while *unoptimized* means usage of 8-byte vector access and scalar arithmetic only. Figure 7 demonstrates these results.

The performance gap between highly optimized, hand-tuned codes to an unoptimized version is known as the ninja gap [17]. This gap is much more pronounced for AMD GPUs suggesting that architecture-aware optimization is critical to fully harness the computational potential of AMD hardware. This gap is lesser for NVIDIA GPUs retaining much of its performance characteristics without explicit optimization.

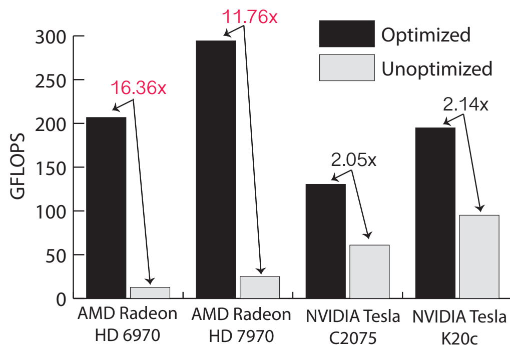

Figure 7: Optimizations applied to a larger 2D FFT of size 256×256

5. RELATED WORK

Optimized FFT libraries started with FFTW in the late 1990s with single-core CPUs as the standard computer architecture of the time [9]. FFTW was developed to address the growing pains of novel CPU features such as deep pipeline schemes, superscalar execution, and speculation. FFTW succeeded in addressing this diversity within CPUs by providing portable performance via auto-tuning.

With the advent of accelerator-based computing, coupled with architecture support for gather and scatter operations, researchers have tapped into massively parallel architectures such as the GPU to further accelerate the FFT. The work of Govindaraju et al. set the stage for high-performance discrete Fourier transforms on NVIDIA GPUs [10]. This seminal work accelerated FFT through the CUDA architecture rather than through low-level graphics APIs such as DirectX or OpenGL. Concurrent with Govindaraju's work is the work of Volkov and Kazian [19]. Both of these works focus primarily on the algorithmic design and mapping of FFT onto the GPU and are limited to the NVIDIA GPU architecture.

FFTs on AMD GPUs have been less explored, despite its higher peak performance and memory bandwidth compared to NVIDIA GPUs. Of the few works that are out there, we note Nukada's tutorial presentation on double-precision 1D FFT targetted for the AMD Radeon HD 6970 GPU [14]. His work provides a cursory overview of achieved performance on OpenCL codes.

In the taxonomy of the aforementioned work, this work focuses primarily on optimizations and its interactions with machine-level behavior. Our end goal is to determine a set of optimizations amenable for a vendor's GPU architectures in order to promote performance portability. While FFTW is performance-portable across any CPU regardless of instruction set, architecture, or organization, our work seeks performance portability across graphics processors, and later, to any heterogeneous processor.

6. CONCLUSIONS

We briefly summarize our results in Table 6 calculated via the following model flop count for FFT.

$$\text{GFLOPS} = \frac{5 \times 10^{-9} \times N \times log_2(N) \times \text{batches}}{\text{seconds}}$$

The model flop count equation is derived from the average number of floating point operations to calculate the FFT. The $N \times log_2(N)$ portion is the algorithmic complexity of the Cooley-Tukey formulation and the constant, 5, is the average number of floating point operations in an FFT's butterfly computation.

We identify the following list of optimization principles for FFT on GPUs.

- **On-chip Resource Usage**. Prefetching memory to on-chip resources reduces the overall global bus traffic. In particular, perform computation on the register file and communication (transpose) operations in scratch-pad local memory (RP+LM-CM). Unrolling the transpose step saves local memory space by a factor of the vector size, thus improving thread occupancy in all implementations.

- **Scalar and Vector Operations**. 8 or 16-byte scalar arithmetic operations are best suited for both scalar and VLIW GPU architectures. In contrast, vectorized math (VAVM) degrades performance due to an increase in register and local memory usage.

- **Use Constant Memory**. In isolation, constant memory provided little improvement as the computation was performed in global memory. In concert, however, constant memory produced a significant performance improvement by trading transcendental floating-point computation for cached memory accesses.

Table 6: Summary of Experiment (Number of elements = 128 MB)

Device	Sample Size	Baseline (GFLOPS)	Optimal (GFLOPS)	Speedup	Speedup over FFTW	Optimal Set
Intel i5-2400 (FFTW)	16		36			
	64		43			
	256		48			
Radeon HD 6970	16	12	174	14.5x	4.8x	RP + LM-CM + CGAP + **VASM4** + CM-K
	64	14	257	18.4x	6.0x	RP + LM-CM + CGAP + **VASM4** + CM-K
	256	11	346	**31.5x**	7.2x	RP + LM-CM + CGAP + **VASM2** + CM-K
Radeon HD 7970	16	36	240	6.7x	6.7x	RP + LM-CM + CGAP + **VASM4** + CM-K
	64	23	366	15.9x	8.5x	RP + LM-CM + CGAP + **VASM2** + CM-K
	256	24	437	18.2x	**9.1x**	RP + LM-CM + CGAP + **VASM2** + CM-K
Tesla C2075	16	37	139	3.7x	3.9x	RP + LM-CM + CGAP + **VASM2** + CM-K
	64	69	200	2.9x	4.7x	RP + LM-CM + CGAP + **VASM4** + CM-K
	256	60	177	3.0x	3.7x	RP + LM-CM + CGAP + **VASM2** + CM-K
Tesla K20c	16	54	183	3.4x	5.1x	RP + LM-CM + CGAP + **VASM2** + CM-K
	64	99	265	2.7x	6.2x	RP + LM-CM + CGAP + **VASM4** + CM-K
	256	95	280	2.9x	5.8x	RP + LM-CM + CGAP + **VASM2** + CM-K

7. REFERENCES

[1] AMD Accelerated Parallel Processing OpenCL Programming Guide, July 2012.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*.

[3] L. Brandon, C. Boyd, and N. Govindaraju. Fast Computation of General Fourier Transforms on GPUs. In *Multimedia and Expo, 2008 IEEE International Conference on*, 23 2008-april 26 2008.

[4] Y. Chen, X. Cui, and H. Mei. Large-scale fft on gpu clusters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 315–324, New York, NY, USA, 2010. ACM.

[5] K. Czechowski, C. Battaglino, C. McClanahan, K. Iyer, P.-K. Yeung, and R. Vuduc. On The Communication Complexity of 3D FFTs and Its Implications for Exascale. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, 2012.

[6] C. del Mundo and W.-c. Feng. Enabling Efficient Intra-Warp Communication for Fourier Transforms in a Many-Core Architecture. In *Supercomputing, 2013. Proceedings of the 2013 ACM/IEEE International Conference on*, 2013. (Poster Publication).

[7] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-Tuning of Fast Fourier Transform on Graphics Processors. In *PPoPP '11*, pages 257–266, 2011.

[8] O. Fialka and M. Cadik. FFT and Convolution Performance in Image Filtering on GPU. In *Information Visualization, 2006. IV 2006. Tenth International Conference on*, July 2006.

[9] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, 1998.

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

[11] L. Gu, X. Li, and J. Siegel. An Empirically Tuned 2D and 3D FFT Library on CUDA GPU. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10.

[12] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[13] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption*, Lecture Notes in Computer Science.

[14] A. Nukada. Fast Fourier Transform on AMD GPUs, 2011.

[15] 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*, SC '09, 2009.

[16] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth Intensive 3-D FFT Kernel for GPUs using CUDA. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, nov. 2008.

[17] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *SIGARCH Comput. Archit. News*.

[18] I. Uzun, A. Amira, and A. Bouridane. FPGA Implementations of Fast Fourier Transforms for Real-time Signal and Image Processing. *Vision, Image and Signal Processing, IEE Proceedings* -, june 2005.

[19] V. Volkov and B. Kazian. Fitting FFT Onto the G80 Architecture. May 2008.

[20] Q. Zhang, A. Kokkeler, and G. Smit. An Efficient FFT For OFDM Based Cognitive Radio On A Reconfigurable Architecture. In *Communications, 2007. ICC '07. IEEE International Conference on*, June 2007.