SC: U: Towards a Performance-Portable FFT Library for Heterogeneous Computing*

Carlo del Mundo (Student)* and Wu-chun Feng (Advisor)*†
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.

After extensive optimization, our study suggests that there is one unique optimization code sequence that performs optimally across all GPU architectures with global memory data transfer becoming the primary bottleneck. 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 [1, 7, 9, 11]. 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 [2, 3, 5, 6, 8, 10].

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 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. Section 3 presents the optimizations that we have applied to the GPU cores. Section 4 summarizes and discusses our results. Finally, Section 5 presents our conclusions.

2. BACKGROUND

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 synchro-

*This work was supported in part by NSF I/UCRC IIP-0804155 via the NSF Center for High-Performance Reconfigurable Computing.

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.
X-XXXXX-XX-X/XX/XX.

Table 1: 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 2: 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 [10].
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 $\times$ 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 $\times$ 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

nization 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 [6, 10]. 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.

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. For brevity, we only display results for the 256-pt FFT (as the performance between 16-, 64-, and 256- are similar.) We then evaluate shuffle, a poorly understood mechanism endemic to NVIDIA Tesla Kepler GPUs that allows for intra-warp communication without the use of shared memory.

3.1 Optimizations

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.

Table 2 depicts the optimizations applied to the GPU cores. For a detailed explanation of each optimization, we refer the reader to the original paper [4].

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 1 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 configured to utilize four threads on OpenMP with explicit AVX extensions. FFTW was compiled using gcc v4.4.5. We apply optimization listed in Table 2 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 1 shows our results in isolation for each stage of FFT. We applied all optimizations in Table 2 with the exception of shuffle which is evaluated in concert with other optimizations.

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). Even without explicit register preloading and local memory usage, NVIDIA GPUs achieves substantial performance. 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.

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.

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 particular, 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 2 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).

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.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

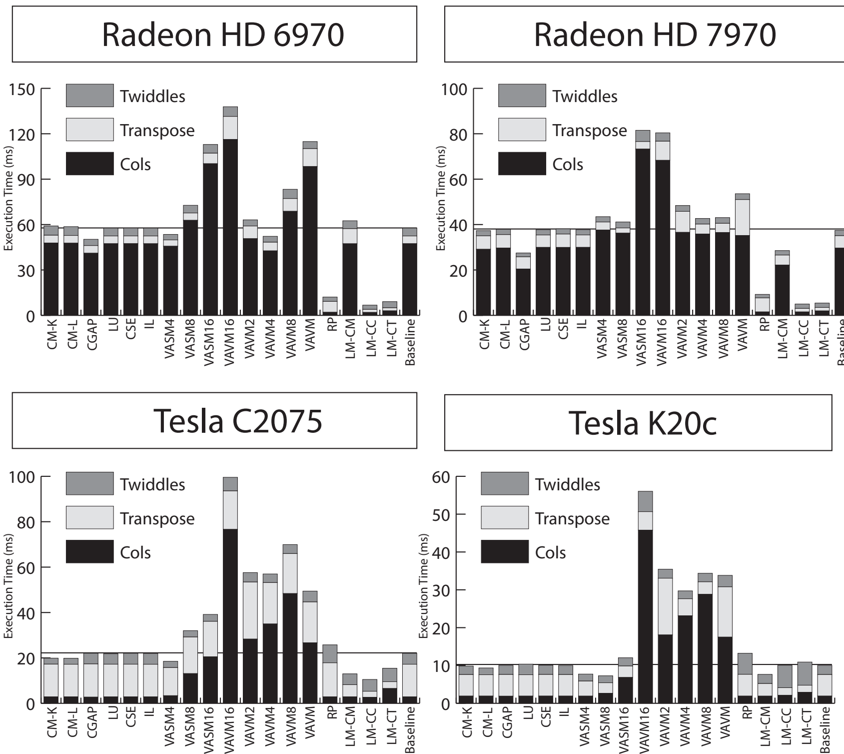

Figure 1: Optimizations applied in isolation to a baseline, unoptimized GPU kernel for 256-pts on Radeon HD 6970, Radeon HD 7970, Tesla C2075, and Tesla K20c.

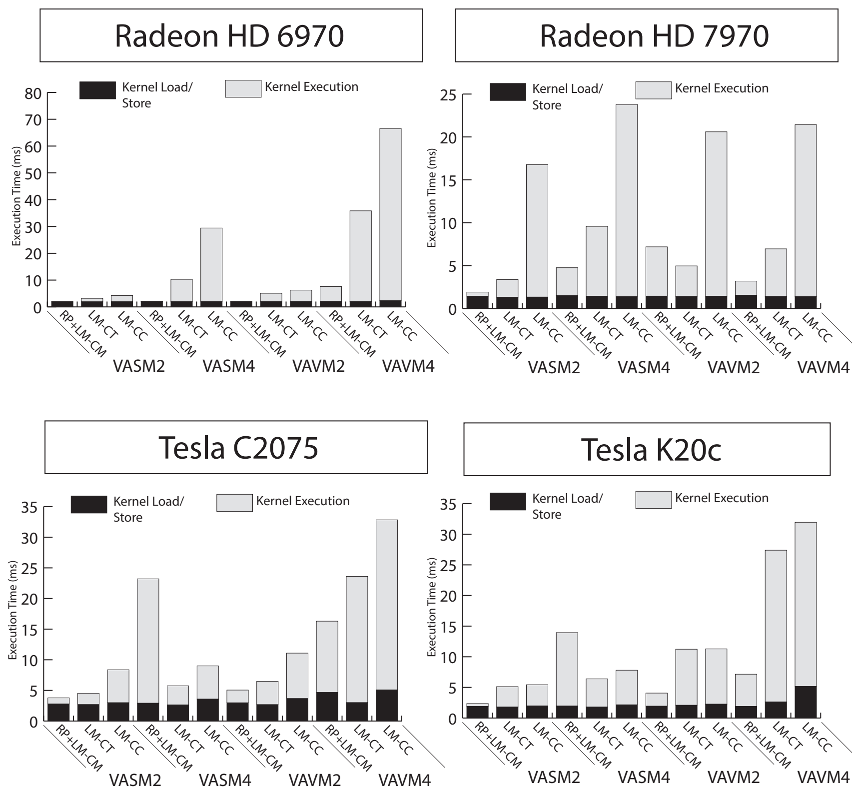

Figure 2: Optimizations applied in concert to a baseline, unoptimized GPU kernel for 256-pts on Radeon HD 6970, Radeon HD 7970, Tesla C2075, and Tesla K20c. Note: coalesced global access pattern (CGAP) and constant memory kernel literal (CM-K) was applied to the data points listed.

Table 3: Summary of Experiment (Number of elements = 128 MB)

Device	Sample Size	Baseline (GFLOPS)	Optimal (GFLOPS)	Speedup	Speedup over FFTW	Optimal Set
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

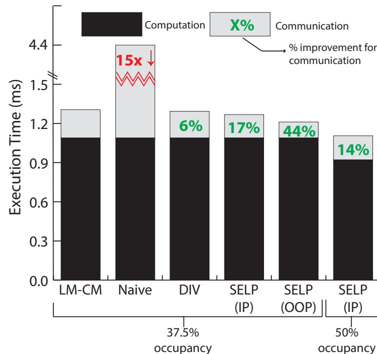

Figure 3: Results for shuffle mechanism applied to a 256-pt FFT.

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 3 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 issue, we introduced code divergence (e.g., **DIV**) to ensure *a priori* indexing. 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.

5. CONCLUSIONS

We briefly summarize our results in Table 3 calculated via the following model flop count for FFT.

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

6. REFERENCES

[1] K. Asanovic. A View of the Parallel Computing Landscape. *Commun. ACM.*

[2] 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.

[3] K. Czechowski 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.

[4] C. del Mundo and W.-c. Feng. Towards a Performance-Portable FFT Library for Heterogeneous Computing. In *2014 ACM International Conference on Computing Frontiers*, CF '14, 2014.

[5] Y. Dotsenko. Auto-Tuning of Fast Fourier Transform on Graphics Processors. In *PPoPP '11*, pages 257–266, 2011.

[6] N. K. Govindaraju. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, 2008.

[7] V. Lyubashevsky. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption*, Lecture Notes in Computer Science.

[8] 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.

[9] 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.

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

[11] 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.