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 of 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-
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- point is similar.) We then evaluate shuffling data elements solely in local memory. This optimization requires only \( N \times sz(\text{float}) \) local memory by transposing each dimension of a floatn vector one dimension at a time.

### 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 6970 GPU
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 targeted 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 achieve 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.
We briefly summarize our results in Table 3 calculated via the following model flop count for FFT:

\[
\text{GFLOPS} = 5 \times 10^{-9} \times N \times \log_2(N) \times \text{batches} \quad \text{seconds}
\]

5. CONCLUSIONS

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

<table>
<thead>
<tr>
<th>Device</th>
<th>Sample Size</th>
<th>Baseline (GFLOPS)</th>
<th>Optimal (GFLOPS)</th>
<th>Speedup</th>
<th>Speedup over FFT (X%)</th>
<th>Optimal Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radeon HD 6970</td>
<td>16</td>
<td>12</td>
<td>174</td>
<td>14.5x</td>
<td>4.8x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>14</td>
<td>257</td>
<td>18.4x</td>
<td>6.0x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>11</td>
<td>346</td>
<td>31.5x</td>
<td>7.2x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
<tr>
<td>Radeon HD 7970</td>
<td>16</td>
<td>36</td>
<td>240</td>
<td>6.7x</td>
<td>6.7x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>23</td>
<td>366</td>
<td>15.9x</td>
<td>8.5x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>24</td>
<td>437</td>
<td>18.2x</td>
<td>9.1x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>16</td>
<td>37</td>
<td>139</td>
<td>3.7x</td>
<td>3.7x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>69</td>
<td>200</td>
<td>2.9x</td>
<td>4.7x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>60</td>
<td>177</td>
<td>3.0x</td>
<td>3.7x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
<tr>
<td>Tesla K20c</td>
<td>16</td>
<td>54</td>
<td>183</td>
<td>3.4x</td>
<td>5.1x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>99</td>
<td>265</td>
<td>2.7x</td>
<td>6.2x</td>
<td>RP + LM-CM + CGAP + VASM4 + CM-K</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>95</td>
<td>280</td>
<td>2.9x</td>
<td>5.8x</td>
<td>RP + LM-CM + CGAP + VASM2 + CM-K</td>
</tr>
</tbody>
</table>

6. REFERENCES