Benchmarking the NVIDIA 8800GTX with the CUDA Development Platform

Michael McGraw-Herdeg, MIT
Douglas P. Enright, The Aerospace Corporation
B. Scott Michel, The Aerospace Corporation

©2007 The Aerospace Corporation
Outline

• Introduction

• The G8X GPU

• Time-Domain Finite Input Response Filter (TD-FIR)

• Frequency-Domain Finite Input Response Filter (FD-FIR)

• Complex QR Decomposition

• CUDA Programmability

• Conclusions and Acknowledgments
Introduction

• Wish to examine performance characteristics of newly available data-parallel architectures

• HPEC Challenge Benchmark Suite
  – Finite impulse response filter
    – Time-Domain: 15x speedup
    – Frequency-Domain (1D FFT): 35x speedup
  – QR decomposition
    – Data-interdependent matrix factorization
    – 2.5x speedup

• CUDA platform: C on a GPU + Runtime Library for
  – Compute Unified Device Architecture
  – Intuitive, thread-level parallelization with SIMD operations
  – GeForce 8 Series/Quadro FX/Tesla

CUDA Software Stack

Fig. 1-3 NVIDIA CUDA Programming Guide
Outline

- Introduction

- The G8X GPU

- Time-Domain Finite Input Response Filter (TD-FIR)

- Frequency-Domain Finite Input Response Filter (FD-FIR)

- Complex QR Decomposition

- CUDA Programmability

- Conclusions and Acknowledgments
The G8X GPU: Architecture

- Sixteen SIMD 1350 MHz “multiprocessors”
  - 16KB fast shared memory
  - 64KB constant memory
  - 8KB texture memory
  - 8192 total registers
  - 8 chained SIMD processors
  - Single precision floating point
    - Tesla to have double precision
- 768MB of GDDR3 global device memory
- PCIx16 bus adapter to host system
The G8X GPU: What Software People See

• Developers code in a C-extended language and call “kernels” (inlined device functions)
  – 32 blocks of 256 threads each: kernel<<32, 256>>(args);
  – Thread scheduling is tightly interleaved and invisible to developers
  – Reading global memory is slow (hundreds of cycles), so the challenge is interleaving data accesses appropriately

• On-card computations are standard single-precision floating point arithmetic
  – Addition, multiplication, division, square root, trig functions

• Mathematical library support
  – Vector & Matrix Linear Algebra: CUBLAS
  – Fast Fourier Transform: CUFFT
Outline

• Introduction

• The G8X GPU

• Time-Domain Finite Input Response Filter (TD-FIR)

• Frequency-Domain Finite Input Response Filter (FD-FIR)

• Complex QR Decomposition

• CUDA Programmability

• Conclusions and Acknowledgments
Time-Domain Filter: Approach

- Convolve signal and filter: for each filter element, multiply by entire signal and add to an element of the result vector

- In CUBLAS, the inner loop is just:
  
  ```c
  cublasCaxpy(signalsize, (cuComplex) filterdata, (cuComplex*) signalptr, 1, (cuComplex*) resultptr, 1);
  ```

Diagram from “Exploring the Cell with HPEC Challenge Benchmarks”, S. Sacco, G. Schrader, J. Kepner, HPEC 2006

- Number of Operations:
  - K – Filter size
  - N – Input size
  - M – Number of filters
  - Total FOPs: \( \sim 8 \times M \times N \times K \)

- Output Size:
  - \( L = N + K - 1 \)
Time Domain Results

- Signal length performance analysis
  - GPU performance strongly dependent on signal length
    - 4x length, 4x perf. (ratio 1,3,4 to 2), 8x length, 12x perf. (5 to 2)
- Filter size performance analysis
  - GPU performance relatively invariant to filter size (1,3,4)
- Reference CPU system is a dual-core Athlon x64 4200+ (2210MHz)
Outline

- Introduction
- The G8X GPU
- Time-Domain Finite Input Response Filter (TD-FIR)
- Frequency-Domain Finite Input Response Filter (FD-FIR)
- Complex QR Decomposition
- CUDA Programmability
- Conclusions and Acknowledgments
Frequency-Domain Filter: Approach

• Time-domain convolution is frequency-domain multiplication:
  – Convolution Theorem: FFT -> multiply -> FFT
  – M: number of filter/signal pairs, N: signal length, K: filter length
  – Operation count:
    – Time-domain: 8*M*N*K flops
    – Frequency domain: M*(10*N*log2(N)+8*N)) flops.
  – Frequency-domain approach optimal for large filters

• CUFFT library does all the work!

• Series approach: convolve one signal at a time, attacking them sequentially

• Parallel approach: convolve all the signals at once using batching
Frequency Domain Results: Series Computations

- **Signal length performance analysis**
  - Long signals (tests 1,3,4): GPU calculation 1.5-16x faster than CPU
  - Short signal (test 2): CPU is faster by a factor of 2

- **Filter size performance analysis**
  - GPU performance is fairly invariant to filter size (tests 1,3,4)
Frequency Domain Results: Parallel Computations

• Parallel FFT Performance
  – GPU-only calculation 35x faster than CPU (test 3)
  – Large FFTs are serialized (test 5)

• PCI bus hampers overall performance
  – One-half of GPU-only performance lost to PCI bus latency (tests 1,3,4,5)
Outline

• Introduction

• The G8X GPU

• Time-Domain Finite Input Response Filter (TD-FIR)

• Frequency-Domain Finite Input Response Filter (FD-FIR)

• Complex QR Decomposition

• CUDA Programmability

• Conclusions and Acknowledgments
Complex QR Decomposition: Approach

\[ A = QR \]

- A = QR via Fast Givens QR
  - Givens rotations to eliminate one element of A at a time
  - \( R \): computed from A by eliminations
  - \( Q \): computed as a by product of eliminating A

- Each Givens rotation modifies two rows; some parallelization possible

- The Sameh-Kuck pattern (top right) allows up to n concurrent rotations

Complex QR Decomposition: Results

- CPU performance is constant across test sizes
- GPU performs much better on large tests (about 2.5x faster)
- Data interdependence is less problematic for large matrices
An Alternative QR Approach

- A pipelined Givens pattern theoretically twice as fast as Sameh-Kuck:

<table>
<thead>
<tr>
<th>cycle</th>
<th>steps</th>
<th>set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3, 4, 5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6, 7, 8, 9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10, 11, 12, 13, 14</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15, 16, 17, 18, 19, 20</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>21, 22, 23, 24, 25, 26, 27</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>28, 29, 30, 31, 32, 33, 34</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>35, 36, 37, 38, 39, 40</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>41, 42, 43, 44</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>45, 46</td>
<td></td>
</tr>
</tbody>
</table>


- But it won't work with CUDA.
  - Pipelining requires either fast thread synchronization or fast fine-grained operations between kernels
  - The computation at each step is too small
Outline

• Introduction

• The G8X GPU

• Time-Domain Finite Input Response Filter (TD-FIR)

• Frequency-Domain Finite Input Response Filter (FD-FIR)

• Complex QR Decomposition

• CUDA Programmability

• Conclusions and Acknowledgments
CUDA Programmability

• Solid NVIDIA code base solves many programming issues
  – “Starter code” examples in SDK demonstrate common GPU computation patterns
  – CUDA includes “malloc” and “memcpy” clones for on-card memory; developers can easily transfer between card & host memory
  – CUFFT, CUBLAS libraries accelerate computation without device code

• Source line of code counts:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>CUDA C</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDFIR</td>
<td>122</td>
<td>350 [TDFIR + FDFIR]</td>
</tr>
<tr>
<td>FDFIR</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>QR</td>
<td>238</td>
<td>369</td>
</tr>
</tbody>
</table>

• Benchmark coding time equivalent in C, CUDA: O(days)
How to Get High Performance

• Parallelizing at the thread level takes only a little practice; optimizing thread utilization requires more thought
  – In CUDA's SIMD architecture, thread execution is hidden
    – high-level control over number of threads and “thread blocks”
  – Partitioning data is the real challenge
    – coalesced memory reads and writes are much faster than random ones

• GPU data-parallel architecture designed for high (computations/memory operations) ratio

• Asynchronous library calls are useful – CUBLAS, CUFFT

• Work on large data sets, but in small bites that can fit in shared memory

• Avoid need for synchronization
  – A __syncthreads() primitive exists, but is severely limited
  – Threads are too tightly interwoven to be managed by developers
Future Directions

• Expanded libraries expected
  – CUBLAS is missing many complex operations
  – Handrolled Givens operations used in QR are probably not optimal

• Atomic operations are coming
  – Atomic integer operations available in Compute Capability 1.1, which currently runs on newer, slower cards
  – Atomic floating-point operations in the future?

• Double precision floating-point by end of '07

• Transparent multi-GPU computation with Tesla

• More support for asynchronous actions expected
  – Goals: send data to a running kernel, multiple concurrent kernels
  – The GPU array as a mature multicore platform
Outline

• Introduction

• The G8X GPU

• Time-Domain Finite Input Response Filter (TD-FIR)

• Frequency-Domain Finite Input Response Filter (FD-FIR)

• Complex QR Decomposition

• CUDA Programmability

• Conclusions and Acknowledgments
Conclusions

• CUDA brings C to a multiprocessor architecture
• Pros:
  – It’s easy to use and program
  – Extensive, responsive support base, including developers
  – NVIDIA is actively supporting the project
  – CUBLAS and CUFFT are remarkably successful
  – Handmade SIMD code yields impressive results
• Cons:
  – Performance depends heavily on the algorithm
  – Handmaking SIMD code requires learning a “new” style
  – Mostly exploiting capabilities is easy
    – Fully exploiting capabilities is difficult
Acknowledgments

• The Aerospace Corporation Summer Internship Program
• The Aerospace Corporation IR&D Program
• Computer Systems Research Department
  – Director Stuart Kerr
• Computers and Software Division
• NVIDIA Corporation

All trademarks, service marks, and trade names are the property of their respective owners.