Implementing a GPU-Efficient FFT

John Spitzer
NVIDIA Corporation

Why Fast Fourier Transform?
“Classic” algorithm
Computationally intensive
Useful
Imaging
Signal analysis
Procedural texturing

What is a FFT?
Fourier transform
Transform function from spatial- to frequency-domain
\[ H(f) = \int h(t) e^{2\pi i ft} dt \]

Inverse Fourier transform
\[ h(t) = \frac{1}{\sqrt{N}} \sum_{n=-N/2}^{N/2-1} H(n) e^{2\pi i nt/N} \]

Discrete Forms for Series of Samples
Discrete Fourier transform
\[ H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i kn/N} \]

Inverse discrete Fourier transform
\[ h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i nk/N} \]
Solving Fourier Transforms

- As matrix equation:
  \[ H_n = \sum_{k=0}^{N-1} W_{nk} h_k \]
  \[ \hat{H} = W \hat{h} \]
  \[ O(N^2) \] operations

- Recursive (Fast Fourier Transform):
  \[ F_k = \sum_{j=0}^{N-1} e^{2\pi i j k/N} f_j \]
  \[ F_k = F_k^* + W F_k^o \]
  \[ O(N \log N) \] operations

Fast Fourier Transform Implementations

- [Numerical Recipes in C]
  - Loop over elements for bit-reversal
  - Loop log N times to recombine neighbors
  - Weights are computed iteratively

- Fastest Fourier Transform in the West
  - http://fftw.org
  - Optimized for current CPU architectures
  - Adapts itself to current CPU cache sizes

Application Example: SETI@home

- SETI@home Pulse Search
  - Search for dispersed pulses of intrinsically short duration, e.g., pulsars

- Computation task at hand:
  - Have ~2.5 years of data
  - Need to examine every .8ms of that data
  - Each examination requires ~0.34 GFlops
  - mostly in the form of FFTs
  - ~33,507,000,000 GFlops computation

  - Needs every help it can get

GPU FFT Feasibility

- 2048 element FFT requires
  - ~8 * 2048 * log(2048) = ~180 KFlops
  - 2048 * 8 = 16KB of data

- Computational limits for GeForceFX 5900 (NV35)
  - Vertex: .450 GHz * 3 units * 4 FLOPS/unit = 5.4 GFLOPS
  - Pixel: .450 GHz * 4 units * 12 FLOPS/unit = 21.6 GFLOPS
  - Total: 27 GFLOPS

- Theoretical times for GPU
  - Download: 16k @ 2.0 GB/s = 8 us (AGP 8X)
  - Computation: 180KGFlop @ 27 GFLOPS = ~7 us
  - Upload: 16k @ 0.18 GB/s = 90 us (PCI)
FFT Algorithm Overview

Pass 0: Bit Reversal
Pass 1: combine 1-neighbors
Pass 2: combine 2-neighbors
... (multiple passes)
Pass log N: combine N/2-neighbors

Mapping Data-Structures to GPU

1D texture (from AGP)
1D float texture (render target)
1D float texture (render target)
1D float texture (render target)
1D float texture (render target)
1D float texture (render target to be read back to system memory)

GPU Algorithm Overview

- Download FFT data to GPU as a 1D texture
  - 2k by 1 texels big
- Render quad into float texture render-target
  - Quad is 2k pixels wide and 1 pixel high
  - Use x pixel position to index texture
- Bit-Reversal done as:
  - Pass address of pixel as texture coordinate
  - Fragment(x) = tex(bitreversal(x))
  - Bitreversal() is simply texture look-up

GPU Algorithm Overview (cont.)

- Log N combination passes
  - Fragment(x) = tex(index0(x)) * w(index1(x)) * tex(index1(x))
  - w(), index0(), and index1() are textures
  - Different for every pass
  - Pre-computed
- Read final render-target back into system memory
Red Flags for GPU Performance

- $1 + \log N$ passes
- All data stays on GPU (good)
- Per-vertex computations trivial (good)
- Lots of API calls for CPU to instruct GPU what to do
- GPU has to finish each pass before next one starts

- Only 1D textures
  - GPUs highly optimized for 2D textures
- Complex number computations
  - Complex numbers are 2D
  - But hardware is optimized for 4-vectors

Batching Many FFT Transforms

- Download 2D texture of coefficients
- Compute hundreds of FFTs per pass
- Cuts driver calls by hundreds of times
- Fully utilizes multi-pipe fragment processing hardware

- Basically uses the same fragment programs
  - Only differ in needing a 2nd texture coordinate

Using Vector Operations

- Store 2 complex numbers per texture
  - $(t0.r, t0.g)$ is first number
  - $(t0.b, t0.a)$ is second number

- Store 4 complex numbers in 2 textures
  - $(t0.r, t0.g, t0.b, t0.a)$ are real parts
  - $(t1.r, t1.g, t1.b, t1.a)$ are imaginary parts
  - Code is more symmetric
  - But more temporaries are used

Real World Performance

**CPU**
- FFTW algorithm
- 3.0 GHz Intel Pentium 4
- 2048 FFT takes 12 us
- 1.5 GLOPS

**GPU**
- Algorithm outlined here
- NVIDIA GeForceFX 5900 Ultra (NV35 @ 450 MHz)
- 2048 FFT takes 16 us (32 us with readback over PCI)
- 1.1 GLOPS (.6 GFLOPS with readback)
Optimization Possibilities

- Range and precision of computation and results
  - Is 16-bit floating point sufficient for registers?
  - Conversion to lower precision has double benefit:
    - Faster to compute
    - Faster to transfer back to CPU
- If range and precision of input is limited
  - Don’t compute results, but rather...
  - Replace N passes with table look-up
- Tap into over 5 GLOPS of unused vertex processing

Conclusions

- GPU useful now as co-processor to CPU
- Keep the faith!
  - Faster access to (and particularly from) graphics subsystem is critical, but coming soon
  - GPU parallelism outstripping that of CPUs
  - GPUs will continue to enjoy an advantage over CPUs in dedicated memory bandwidth

Future Work

- Integrate more of the Pulse Search problem
- Straightforward power computations and thresholding after FFT
- Thresholding translates to rejecting a fragment
  - Potentially saves memory bandwidth
  - Use occlusion queries to determine if read-back is unnecessary

Thanks to...

- Dinesh Manocha for organizing this course
- Matthias Wloka for preparing this material
- Jeremy Zelsnack for implementing the GPU FFT
Questions, Comments, Feedback?

- John Spitzer, spit@nvidia.com