Tuning GPGPU Applications for Performance

Justin Hensley and Jason Yang
Graphics Product Group
AMD

GPGPU
Overview

• Goals:
  - How to approach the optimization process
  - Taking advantage of hardware features

• GPGPU from real world applications
  - Decoding H.264 Video
  - Radial Distortion Correction

• Cap ‘N’ Stream Demo

• Conclusion and Questions
H.264 Decoding: A Performance Case Study

When fast is not good enough!
Where to begin?

- You’re done writing code, now what?
- Does it work?
- Is it fast?
- What does fast mean?
  - 60x faster than the CPU is pretty good
  - What are you leaving on the table?

- How close is it to theoretical?
Breaking down GPGPU performance

- GPGPU applications generally progress through the hardware in a predictable fashion, unlike rendering.

- Theoretical performance can be calculated.

- What we know...
  - # of ALU, TEX, and Write operations
  - GPUShaderAnalyzer

- What do we need to find out?
Calculating Theoretical Performance

• **Radeon X1900XT (R580)**
  - 256 bit memory bus
  - 625/750 MHz Engine/Memory Clocks

• **ALU Performance**
  - 1 ALU Shader

\[
\frac{(# \text{pixels}) \times (# \text{alu instructions})}{(alu/clk) \times (3D \text{ engine speed})} = \frac{(1920 \times 1088) \times (1)}{(48) \times (625 \text{Mhz})} = 0.07ms
\]
Calculating Theoretical Performance (2)

• Texture Performance
  - 1 ALU + TEX Shader
  - ALU and TEX operations occur in parallel

\[
\frac{(#\text{pixels}) \times (#\text{tex instructions})}{(\text{tex/clk}) \times (3D\ \text{engine\ speed})} = \frac{(1920 \times 1088) \times (1)}{(16) \times (625\ \text{Mhz})} = 0.21\text{ms}
\]

• Memory Performance
  - 1 Byte in 1 Byte out (Copy)

\[
\frac{(#\text{pixels}) \times (\text{input} + \text{output\ bits\ per\ pixel})}{(\text{bus\ width}) \times (\text{memory\ speed})} = \frac{(1920 \times 1088) \times (16)}{(256) \times (750\ \text{Mhz} \times 2\text{DDR})} = 0.085\text{ms}
\]
Calculating Theoretical Performance (3)

- Overall Theoretical Performance
  - $\max(\text{ALU,TEX,Memory})$
  - $= \max(0.07, 0.21, 0.085) = 0.21 \text{ ms}$

- Remember, this is only a starting point!
  - ALU and TEX calculation is reasonable
  - Memory is peak
Optimization Example: H.264

- Interprediction - filter data from previously decoded frames
- Deblocking - filter out block edges
- Today: Loosely based on DX9 HW Implementation
Problem Domain

- Max Resolution 1080p = 1920x1088 “RGBA” pixels (3Bpp)
  - Luma (Y) = 1920x1088 pels (1Bpp)
  - Chroma (U and V) = 960x544 pels each
Interprediction

- Decode each pel by interpolating a subregion of previously decoded frames
- Example case: 6x6 fetches per pel
  - Kernel not aligned to x4 boundary
Do more work per pixel

- Problem: Shader has a lot of texture fetches
- Share already fetched data by processing four pels in one pixel (thread)
  - Consolidate texture fetches & Consolidate ALU instructions
  - Consolidate writes (four bytes per pixel)
  - Increase number of threads

Saved 96 fetches!
Optimizing the algorithm

• Theoretical Model for 4 pels per pixel
  - 115 ALU, 56 TEX, 66 Bytes In/Out
  - 480x1088 pixels processed
  - From above equations:
    • ALU time = 2 ms
    • TEX time = 3 ms
    • Mem time = 0.67 ms
    • Theoretical time = 3 ms

• Using the theoretical model we can find the problems and fix it
Do even more work per pixel!

- Use multiple render targets
  - 16 pels per pixel

Saved 522 fetches!
What happens to the model?

• Theoretical Model for 16 pels per pixel
  - 304 ALU, 85 TEX, 107 Bytes In/Out
  - 480x272 pixels processed
  - From above equations:
    • ALU time = 1.32 ms
    • TEX time = 1.12 ms
    • Mem time = 0.29 ms
    • Theoretical time = 1.32 ms

• Shader goes from TEX bound to ALU bound
Oops...

- Render targets split the image

Could do a copy shader at the end, or...
Interleave Render Targets

- CAL API supports interleaved render targets
- Four surfaces appear as one

- Scatter is slow. MRTs are fast.
Back to the real world...

- Compare actual performance with theoretical
- Easy way to determine next step
  - If you’re close, you probably have to redesign if you’ve missed your performance target
  - If you’re off, time to look at the hardware
Interprediction Filtering

• Each block of 4x4 pels are processed differently
  - Up to 6 filter cases
  - Up to 2 prediction directions
  - Up to 2 block types (frame/field)
  - Up to 16 input textures (max 4 for 1080p)

• With up to up to 384 possible cases, that’s a pretty big shader!
Branching isn’t perfect

• Shader length
• Branch overhead
• Divergent paths between pixel blocks
  - One pixel branch can stall the others
  - “Branch Granularity”
Our old friend the Z-Buffer

- Use a fast z-pass to initialize the buffer
- Render each “case” separately

- Fast because of top-of-the-pipe pixel kills
- Very little branching overhead
- Potentially shorter shaders
- More threads
Deblocking

- Filters block edges
- Performance can suffer due to render dependencies
  - Input to next pixel is output from previous pixel
  - Frame divided into MB of 16x16 pels
  - 4 vertical and 4 horizontal passes per MB
  - For 1080p frame, up to 748 passes

- Example...
Vertical Deblocking

- Edge 0
- Edge 1
- Edge 2
- Edge 3

8 pels

Previous macroblock
Do more work in a pixel revisited

- **Save on number of passes**
  - If I have the data, why not use it?
  - Filter two edges in one pixel

![Diagram](image)

- Edge 0 and 1
- Render as 1x16 rectangle to 3 MRT
Hmmm....

- Performance improves, but is still off
  - Hardware units arranged in 2x2 pattern
  - 1x16 strip only uses half the pipes
  - Rearrange the rendering
Read/Write to one texture

- Because of dependencies the output is the input to the next pass.
- CAL allows reading and writing from the same texture
- No need to reset states
Getting more info

• Look at performance counters
  - HW Utilization
  - Texture cache miss
  - Z-buffer statistics
Cache and Memory Tips

- Rearrange the data
- Play with the memory layout
  - CAL API allows more fine grain control
- Memory performance is hard to predict
  - Access pattern
  - Memory subsystem is unknown
- Tip: Try replacing input textures with a very small texture (1x1)
- Doesn’t help with dependent texture fetches
Radial Distortion Correction
Radial Correction

System Requirements:
- Capture video from one or more cameras.
- Transfer images to GPU
- Convert bayer-pattern images to RGB images
- Remove lens distortions
- Return to host for further processing

• System needs to use limited power
  - Mobile GPU

• Want to minimize correction time
  - Images further processed in a large real-time system
Naive Implementation

Input images (system memory) -> copy -> Input images (GPU memory) -> Debayer kernel -> Intermediate (GPU memory)

Intermediate (GPU memory) -> Undistort kernel -> Output images (GPU memory) -> copy -> Output images (system memory)

???
Faster Implementation

Input images (system memory) → ??? → Output images (system memory)

Debayer kernel

Accessing system memory directly is 1.8x faster than naive implementation! (2.74 ms vs. 4.96 ms)

Undistort kernel

GPGPU AMD

32
Caveat: System Memory

- Performance is chipset dependent
- Rasterizers optimized for texture cache performance when rendering graphics
Caveat: System Memory

- Performance is chipset dependent
- Rasterizers optimized for texture cache performance when rendering graphics

Not a system memory “friendly” traversal
“Forcing” Raster pattern

• “Force” the rasterizer to be more friendly with system memory traversal
  - Use strips of geometry (CTM can directly stamp quads)
Effect of “Forcing” Raster pattern

- **2048x2048 float32x4 “copy” shader**
  - Reads input in local GPU memory, writes to system memory
  - RD580 with an R580

- **Full screen quad time = 45.51 ms**
  - ~1.5 GB/sec readback

- **“Raster-Blocks” time = 26.53 ms**
  - ~2.5 GB/sec readback

- Technique could also be used to optimize shaders with non-standard to local memory accesses

1.7x faster
GPU HD Encoding Demo

- Cap ‘N’ Stream
- DX9 game displaying at 1280x768
- 720p MPEG2 encoding on the GPU
  - Motion estimation
  - Multi-GPU capable (not required)
  - Quality scales with HW performance
Conclusion and Questions

• Extremely useful to estimate performance
• Direct access to system memory
• “Forcing” raster pattern can be helpful

• Useful tools
  - GPU Shader Analyzer, GPU Perf Studio Tool
  - http://ati.amd.com/developer/

• Information Contact:
  streamcomputing@amd.com