PERFORMANCE OPTIMIZATION ON FUSION PLATFORMS
Performance Analysis and Optimization Techniques

Srikanth Gollapudi
AMD

Udepta Bordoloi
AMD
AGENDA

- Typical application in heterogeneous computing
- Mapping of different components of the application on a heterogeneous system
- Load distribution between CPU and GPU
- Typical implementation and performance analysis
- Techniques to speed up the implementation
  - Wait for event vs. clFinish
  - Kernel launch overhead
  - Efficient way of moving data from CPU to GPU
  - Usage of zero copy buffers
  - Double buffering
- Kernel optimization techniques
APPLICATION

- JPEG Decoder
  - Huffman decoding
  - Conditional code
  - Not suitable for parallel computing
  - Bitwise computations involved
- IQT and IDCT
  - Math-intensive operations
  - Lot of scope for parallelization
MAPPING ON A HETEROGENEOUS SYSTEM

- Entropy Decode
  - No data parallelism
  - Highly conditional code
  - Not heavily math intensive
  - Suitable for CPU

- IQT and IDCT
  - Data parallelism possible
  - Highly math intensive
  - Suitable for GPU

Diagram:
- Bitstream Read
- Header Extraction
- Entropy Decode
- Inverse Quant
- Inverse DCT

CPU

GPU
DATA TRANSFER ISSUE

- Entropy decode produces uncompressed coefficients
  - For 4000x3000 YUV 420 image, this is ~36MBytes per frame
- Data generated by CPU needs to be transferred to GPU for further processing

- Data transfer time higher than GPU processing time
- System performance will be gated by the data transfer time
- Perform lossless compression after entropy decoding on CPU
  - Simple run-length encoding
- Re-compressed coefficients are transferred from CPU to GPU
  - This reduces data transfer overhead
- GPU should be able to decompress coefficients efficiently
  - Design the lossless compression scheme to have data parallelism
- Gain from this compression should be significant enough to reduce transfer load
  - At the extra cost of CPU load for lossless compression
TYPICAL IMPLEMENTATION IN OPENCL™

- Typical Implementation
  - Initializations
    - Create command queue
    - Build and load kernels
  - Memory/Buffer Allocations
    - `malloc RLE_coeff` – to store run-length encoded coefficients
    - `CreateBuffer GPU_RLE_coeff_buf` – GPU side buffer to read encoded coefficients and process
  - CPU Code
    - Read JPEG bitstream
    - Entropy decode, run-length encode
    - Write into `RLE_coeff`
      - `clEnqueueWriteBuffer` – `GPU_RLE_coeff_buf, RLE_coeff`
      - `clEnqueueNDRangeKernel`
      - `clFinish`
PPA INSTRUMENTATION TO PROFILE CPU CODE

- PPA is a tool to profile the code running on CPU and GPU
  - Events can be created and used in the CPU code to point start and end of an event

```c
PPAStartCpuEventFunc(JPEG_Dec_Buffer_alloc);
if ( ocl_JPEGdecomp_initcontrol(oclCoef, (cinfo->first_time != 0), cinfo->ping_pong ))
{
    exit(-1);
}
PPAStopCpuEventFunc(JPEG_Dec_Buffer_alloc);
```

- PPA uses sprofile to collect GPU events
- PPA collects CPU events and GPU events and puts together under same time scale.
ANALYSIS USING PARALLEL PATH ANALYZER (PPA)
**PROFILE ANALYSIS**

- Software overhead when kernels are launched for the first time:
  - When JPEG application is called for only one image, this load is significant
  - Try to average this load by multiple kernel launches, or
  - Launch a dummy kernel and hide this overhead behind some other code, if possible

- **Polling vs. clFinish**
  - Usually all the commands are enqueued without checking for finish
  - In cases where explicit wait is needed for a command to finish:
    - Usage of `clFinish` is observed to take up more time
    - Registering an event for the command and waiting with `clWaitforEvent` is a better approach
    - Active polling with `clGetEventInfo` for CL_COMPLETE is also preferred approach.
PPA ANALYSIS – KERNEL LAUNCH OVERHEAD
PPA ANALYSIS – KERNEL LAUNCH OVERHEAD
PPA ANALYSIS – CLFINISH AT THE END OF KERNEL

![Parallel Path Analyzer screenshot]

Frame: GPU_RLU_DCT Event ID: 4 Group ID: 255 Thread ID: 2520 Thread Priority: 11 Core ID: 0-3 Start time: 9479.92 ms End time: 9498.90 ms Duration: 16.98 ms
PPA ANALYSIS – POLLING AT THE END OF THE KERNEL
EFFICIENT METHOD FOR COPYING FROM CPU TO GPU

- Pinning Cost
  - `clEnqueueWriteBuffer` has a pinning cost along with DMA transfer time

- Pre-pinned Buffers
  - Memory/Buffer Allocations
    - `CreateBuffer CPU_RLE_coeff_buf` – CPU side buffer (Alloc Host Ptr) to store run-length encoded coefficients
    - `RLE_coeff = clEnqueueMapBuffer(CPU_RLE_coeff_buf, ...)`
    - `CreateBuffer GPU_RLE_coeff_buf` – GPU side buffer to read encoded coefficients and process

- CPU Code
  - Read JPEG bitstream
  - Entropy decode, run-length encode
  - Write into `RLE_coeff`
  - `clEnqueueWriteBuffer – GPU_RLE_coeff_buf, RLE_coeff`
  - `clEnqueueNDRangeKernel`
  - `clWaitForEvent`
PPA ANALYSIS – CONVENTIONAL DATA TRANSFER
PPA ANALYSIS – EFFICIENT DATA TRANSFER WITH PRE-PINNED BUFFER
**ZERO COPY BUFFERS**

- **Usage of Zero Copy Buffers**
  - Usage of special buffer types avoid explicit copy of the data from CPU to GPU memory

- **Types of Zero Copy Buffers**
  - **Alloc Host PTR**
    - Buffer resides on host memory and GPU can access this memory directly
  - **Persistent Buffer**
    - Buffer resides on device memory and host can access this data directly
  - **Alloc Host pointer with READ_ONLY attribute**
    - Create a buffer with `CL_MEM_READ_ONLY` and `CL_MEM_ALLOC_HOST_PTR`
    - This kind of buffer can be written from CPU and read by GPU at highest possible data rate
PPA ANALYSIS – ZERO COPY ALLOC HOST PTR
PPA ANALYSIS – ZERO COPY PERSISTENT BUFFER
PPA ANALYSIS – ZERO COPY ALLOC HOST PTR WITH READ ONLY
DOUBLE BUFFERING

- Technique to run CPU and GPU code in parallel:
  - Memory/Buffer Allocations
    - CreateBuffer RLE_coeff_buf1 – CPU write and GPU read buffer (USWC) for run-length encoded coefficients
    - CreateBuffer RLE_coeff_buf2 – CPU write and GPU read buffer (USWC) for run-length encoded coefficients
  - RLE_coeff1 = clEnqueueMapBuffer(CPU_RLE_coeff_buf1, ...
  - CPU code – entropy decode and RLE, use RLE_coeff1
  - Loop
    - clEnqueueUnmapMemObject(CPU_RLE_coeff_buf1, RLE_coeff1,...)
    - RLE_coeff2 = clEnqueueMapBuffer(CPU_RLE_coeff_buf2,...)
    - clEnqueueNDRangeKernel || CPU code – entropy decode and RLE, use RLE_coeff2
    - clEnqueueUnmapMemObject(CPU_RLE_coeff_buf2, RLE_coeff2,...)
    - RLE_coeff1 = clEnqueueMapBuffer(CPU_RLE_coeff_buf1,...)
    - clEnqueueNDRangeKernel || CPU code – entropy decode and RLE, use RLE_coeff2
  - clWaitforEvent
PPA ANALYSIS – DOUBLE BUFFERING WITH ZERO COPY BUFFER
KERNEL OPTIMIZATION
FACTORS TO CONSIDER

- Compute capability
  - High end discrete GPUs are faster
- Memory Bandwidth
  - High end discrete GPUs have larger bandwidth
- Total memory footprint
  - Fusion systems can have much larger memory capacity
- Data transfer from host to GPU
  - PCIe bandwidth

- What is the bottleneck?
GETTING PERFORMANCE OUT OF THE GPU (KERNEL)

- Keep the compute hardware busy
  - GPUs have a lot of ALU capacity

- Minimum number of threads
  - Each SIMD executes two wavefronts simultaneously

- Latency hiding
  - Need more wavefronts in flight to hide data access latency

- Number of wavefronts in flight is impacted by
  - Register pressure
  - Local memory usage

- Control flow
  - Branching effects
  - Clause structure and latency

- Loop unrolling (can use pragma)

- Check whether the kernel is ALU limited or bandwidth limited
  - Profiler
GETTING PERFORMANCE OUT OF THE GPU (BANDWIDTH)

- 128-bit accesses are preferable
  - Use float4, int4 etc.
- Coalesced access pattern will help
  - Workgroup accesses contiguous region, and work-item access over iterations is interleaved
  - As opposed to one work-item accessing a contiguous region over some iterations
- Use Cache whenever possible
  - Local memory (LDS)
  - Images (Texture path)
  - Read-only buffers (no aliasing)
- Writes
  - Fast path
  - Complete path (used with atomics etc.)
QUESTIONS
Disclaimer & Attribution
The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

OpenCL is a trademark of Apple Inc. used with permission by Khronos.

© 2011 Advanced Micro Devices, Inc. All rights reserved.