Optimization Techniques: Image Convolution

Udepta D. Bordoloi | December 2010
Contents

- AMD GPU architecture review
- OpenCL mapping on AMD hardware
- Convolution Algorithm
- Optimizations (CPU)
- Optimizations (GPU)
ATI 5800 Series (Cypress) GPU Architecture

- Peak values:
  - 2.72 Teraflops Single Precision
  - 544 Gigaflops Double Precision
  - 153.6 GB/s memory bandwidth
- 20 SIMDS
- Each SIMD has
  - Local (shared) memory
  - Cached (texture) memory
SIMD Engine

Each SIMD:
- Includes 16 VLIW Thread Processing Units, each with 5 scalar stream processing units + 32KB Local Data Share
- Has its own control logic and runs from a shared set of threads
- Has dedicated texture fetch unit w/ 8KB L1 cache
Wavefront

All threads in a “Wavefront” execute the same instruction

- 16 Thread Processing Units in a SIMD * 4 batches of threads
  = 64 threads on same instruction (Cypress)

What if there is a branch?

1. First, full wavefront executes left branch, threads supposed to go to right branch are masked
2. Next, full wavefront executes right branch, left branch threads are masked

OpenCL workgroup = 1 to 4 wavefronts on same SIMD

- Wavefront size less than 64 is inefficient!
OpenCL View of AMD GPU

- **Constants** (cached global)
- **Workgroups**
- **Image cache**
- **Global memory** (uncached)
- **Local memory** (user cache)
- **L2 cache**
OpenCL™ Memory space on AMD GPU

- Registers/LDS
- Thread Processor Unit
- SIMD
- Local Data Share
- Board Mem/Constant Cache
- Board Memory

Compute Unit 1

Private Memory

Work Item 1

Local Memory

Global / Constant Memory Data Cache

Compute Device Memory

Global Memory
Contents

- AMD GPU architecture review
- OpenCL mapping on AMD hardware
- Convolution Algorithm
- Optimizations (CPU)
- Optimizations (GPU)
Convolution algorithm

Input image

3x3 Filter or mask (weights)

Output Image value = Weighted Sum of neighboring Input Image values

Output image
FOR every pixel:

```c
float sum = 0;
for (int r = 0; r < nFilterWidth; r++)
{
    for (int c = 0; c < nFilterWidth; c++)
    {
        const int idxF = r * nFilterWidth + c;
        sum += pFilter[idxF]*pInput[idxInputPixel];
    }
}
```

\[ pOutput[ idxOutputPixel ] = sum; \]

- For a 3x3 filter: 9+9 reads (from input and filter) for every write (to output)
- For large filters such as 16x16, 256+256 reads for every write

- **Notice read overlap between neighboring output pixels!**
OpenCL Convolution on multi-core CPU

- CPU implementation:
  - Automatic multi-threading!
    - One CPU-thread per CPU-core
  - Highly efficient implementation
    - Each CPU-thread runs one or more OpenCL work-groups
    - Use large work-groups (max on CPU is 1024)

- Optimization 1
  - Unroll loops
  - Pass #defines at run-time (compile option for OpenCL kernels)
  - Use vector types to transparently enable SSE in the backend

- Can be faster than simple OpenMP multi-threading!

- **Image Convolution Using OpenCL™ - A Step-by-Step Tutorial**
OpenCL Convolution on multi-core CPU

![Graph showing time (lower is better) vs. filter width for different implementations: OMP(4-core), Float4If, Def_Float4.](image_url)
Contents

- AMD GPU architecture review
- OpenCL mapping on AMD hardware
- Convolution Algorithm
- Optimizations (CPU)
- Optimizations (GPU)
Convolution on GPU (naïve implementation)

```c
__kernel void Convolve(
    const __global float * pInput,
    __global float * pFilter,
    __global float * pOutput,
    const int nInWidth,
    const int nFilterWidth)
```

- All data is in global (uncached) buffers
- Filter (float * pFilter) is 16x16
- Output image (float * pOutput) is 4096x4096
- Input image (float * pInput) is (4096+15)x(4096+15)
- Work-group size is 8x8 to correspond to wavefront size of 64 on AMD GPUs
- Convolution time: 1511 ms on Radeon 5870
Convolution on GPU (Optimization 1)

- Previously, all data was in global (uncached) buffers
  - Did not reuse common data between neighboring pixels
  - Input items fetched per output pixel = 16x16 = 256

- Can share input data within each work-group (SIMD)
- Preload input data into local memory (LDS), and then access it

- For a work-group of 8x8, if you pre-load input data into LDS
  - Filter is 16x16
  - Output image (per work-group) is 8x8 = 64
  - Input image that is loaded onto LDS is (8+15)x(8+15) = 529
  - Input items fetched per output pixel = 529/64 = 8.3 !

- Convolution time: 359 ms !
You may have deduced by now that if you have a larger work-group, there is more data reuse.

- Largest work-group size on CPU = 1024 = 32x32
- Largest work-group size on GPU = 256 = 16x16

For a work-group of 16x16, if you pre-load input data into LDS

- Filter is 16x16
- Output image (per work-group) is 16x16 = 256
- Input image that is loaded onto LDS is (16+15)x(16+15) = 961
- Input items fetched per output pixel = 961/256 = 3.7 !!

Convolution time: 182 ms !!

**Be aware**: Increasing work-group size and increasing LDS memory usage will reduce the number of concurrent wavefronts running on a SIMD, which can lead to slower performance. There is a trade-off that may nullify the advantages, depending on the kernel.
- Previously, we used local memory (LDS)
  - You can imagine that to be a user-managed cache
  - What if the developer does not want to manage the cache
  - Use the hardware texture cache that is attached to each SIMD
- Why use texture cache instead of LDS?
  - Easier and cleaner code
  - Sometimes faster than LDS
- How to use the cache?
  - OpenCL image buffers = cached
  - OpenCL buffers = uncached
- For the previous example

<table>
<thead>
<tr>
<th>Workgroup size</th>
<th>LDS</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
<td>359 ms</td>
<td>346 ms</td>
</tr>
<tr>
<td>16x16</td>
<td>182 ms</td>
<td>207 ms</td>
</tr>
</tbody>
</table>
Let us go back and start from the naïve implementation to check other possible optimizations.

What about the filter array? (uncached in the naïve kernel)
- It is usually a small array that remains constant
- All work-items (threads) in the work-group (SIMD) access the same element of the array at the same instruction

Options: Image (Texture) buffer or constant buffer

Constant buffer: cached reads as all threads access same element

```__kernel void Convolve(const __global  float * pInput,
__constant float * pFilter __attribute__((max_constant_size(4096))),
__global  float * pOutput, ...)```

Naïve implementation time: 1511 ms
__constant buffer optimization: 1375 ms
Convolution on GPU (Optimization 5)

- Let us again go back to the naïve implementation to check other possible optimizations.
- This time, we will try unrolling the inner loop.
  - Unroll by 4
    - Reduces control flow overhead
    - Fetch 4 floats at a time instead of a single float
- Since we are accessing uncached data (in the naïve kernel), fetching float4 instead of float will give us faster read performance.
  - In general, accessing 128-bit data (float4) is faster than accessing 32-bit data (float).

- Naïve implementation time: 1511 ms
- Unroll-by-4 and float4 input buffer fetch: 401 ms
- Unroll-by-4 and float4 input + float4 filter fetch: 389 ms
Convolution on GPU (Optimization 6)

- What if we combine optimizations 4 and 5 to the naïve kernel?
- Mark the filter as __constant float* buffer
- Unroll by 4 and float4 input buffer fetch

Unroll-4, float4 input fetch + __constant float* filter: 680 ms!!
- Why did the time increase?!
  - Be aware: Using __constant float* increases the ALU usage in the shader as the compiler has to add instructions to extract a 32-bit data from a 128-bit structure.

- Instead, use a __constant float4* buffer
- Naïve implementation time: 1511 ms
- Unroll-by-4 and float4 input buffer fetch: 401 ms
- Unroll-by-4 and float4 input + float4 filter fetch: 389 ms
- Unroll-4, float4 input fetch + __constant float4* filter: 346 ms
We can now combine the previous optimizations to the caching optimizations (LDS and textures).

For a 16x16 work-group, same input and filter sizes as before:

<table>
<thead>
<tr>
<th>Optimization</th>
<th>LDS</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve implementation</td>
<td>1511 ms</td>
<td>1511 ms</td>
</tr>
<tr>
<td>Data reuse (#1,2,3)</td>
<td>182 ms</td>
<td>207 ms</td>
</tr>
<tr>
<td>__constant float* filter(#4)</td>
<td>190 ms</td>
<td>160 ms</td>
</tr>
<tr>
<td>Unroll4, float4 input (#5)</td>
<td>90 ms</td>
<td>130 ms</td>
</tr>
<tr>
<td>Unroll4, float4 input, float4 filter (#5)</td>
<td>83 ms</td>
<td>127 ms</td>
</tr>
<tr>
<td>All above, __constant float* (#6 bad)</td>
<td>88 ms</td>
<td>158 ms</td>
</tr>
<tr>
<td>All above, __constant float4* (#6 good)</td>
<td>71 ms!</td>
<td>93 ms!</td>
</tr>
</tbody>
</table>
Convolution on GPU (Optimization 7)

- Pass #defines to the kernel at runtime:

- When an OpenCL application runs, it can
  - Load binary kernels, or, compile kernels from source at runtime

- When compiling at runtime, runtime parameters (such as filter sizes, work-group sizes etc.) may be available.

- When possible, pass these values to the OpenCL compiler when compiling the kernel using the “-D” option.

- The GPU compiler is able to plug these known parameter values and produce highly optimized code for the GPU.
For a 16x16 work-group, same input and filter sizes as before:

At runtime, if we pass the filter-width, work-group size etc values to the kernel compilation:

<table>
<thead>
<tr>
<th>Optimization</th>
<th>LDS</th>
<th>Texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve implementation</td>
<td>1511 ms</td>
<td>1511 ms</td>
</tr>
<tr>
<td>Data reuse (#1,2,3)</td>
<td>69 ms</td>
<td>128 ms</td>
</tr>
<tr>
<td>__constant float* filter(#4)</td>
<td>25 ms</td>
<td>127 ms</td>
</tr>
<tr>
<td>Unroll4, float4 input (#5)</td>
<td>68 ms</td>
<td>127 ms</td>
</tr>
<tr>
<td>Unroll4, float4 input, float4 filter (#5)</td>
<td>66 ms</td>
<td>127 ms</td>
</tr>
<tr>
<td>All above, __constant float* (#6 bad)</td>
<td>26 ms</td>
<td>127 ms</td>
</tr>
<tr>
<td>All above, __constant float4* (#6 good)</td>
<td>25 ms !!</td>
<td>63 ms !!</td>
</tr>
</tbody>
</table>
Visit the OpenCL Zone on developer.amd.com

http://developer.amd.com/zones/OpenCLZone/

- Tutorials, developer guides, and more
- OpenCL Programming Webinars page includes:
  - Schedule of upcoming webinars
  - On-demand versions of this and past webinars
  - Slide decks of this and past webinars
DISCLAIMER
The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION
© 2010 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ATI, the ATI logo, AMD Opteron, Radeon, FirePro, FireStream and combinations thereof are trademarks of Advanced Micro Devices, Inc. Microsoft, Windows, Windows Vista, and DirectX™ are registered trademarks of Microsoft Corporation in the United States and/or other jurisdictions. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. Other names are for informational purposes only and may be trademarks of their respective owners.