OpenCL™ Programming Webinar Series

Optimization Techniques: Image Convolution

Udeepta D. Bordoloi| December 2010

AMD
The future is fusion

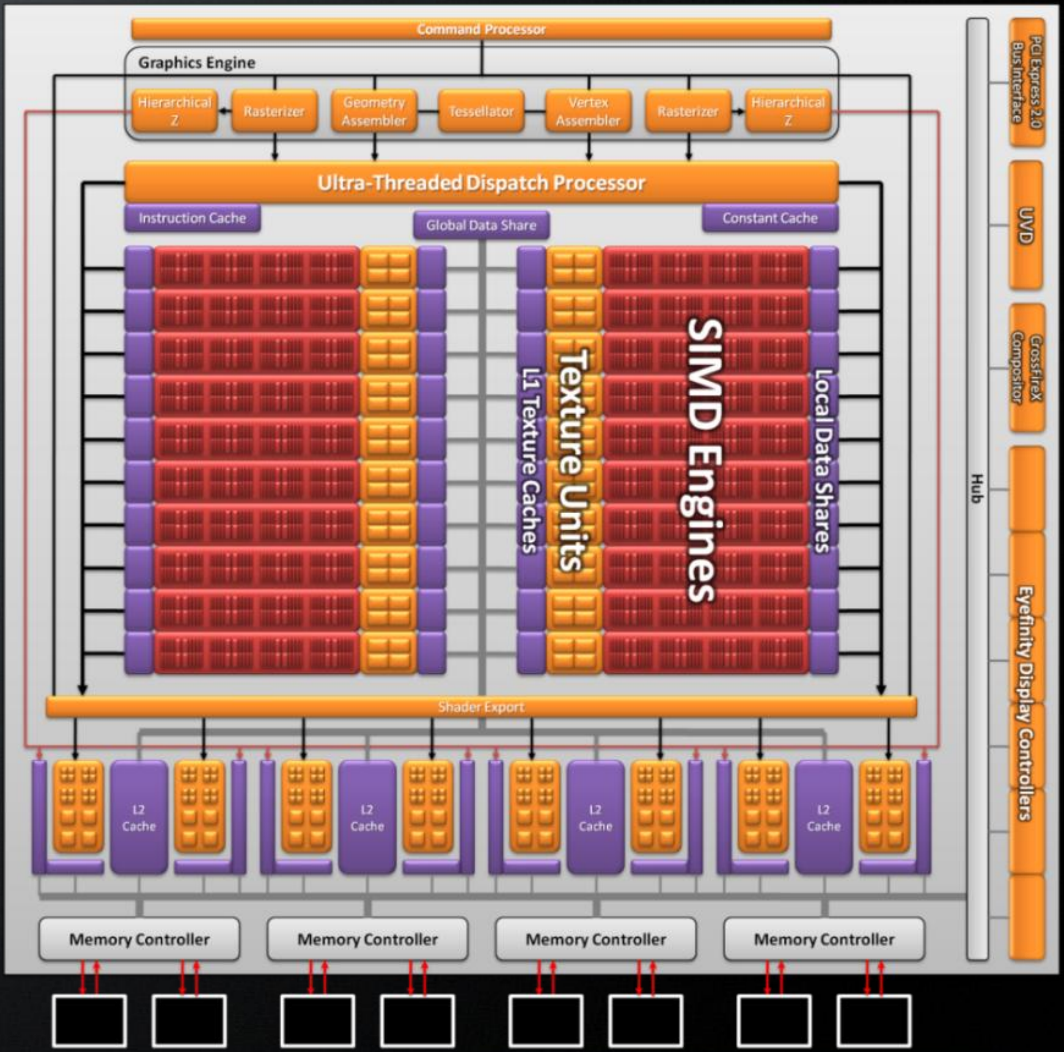
Contents

- 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

AMD The future is fusion

Wavefront

All threads in a "Wavefront" execute the same instruction

- 16 Thread Processing Units in a SIMD * 4 batches of threads

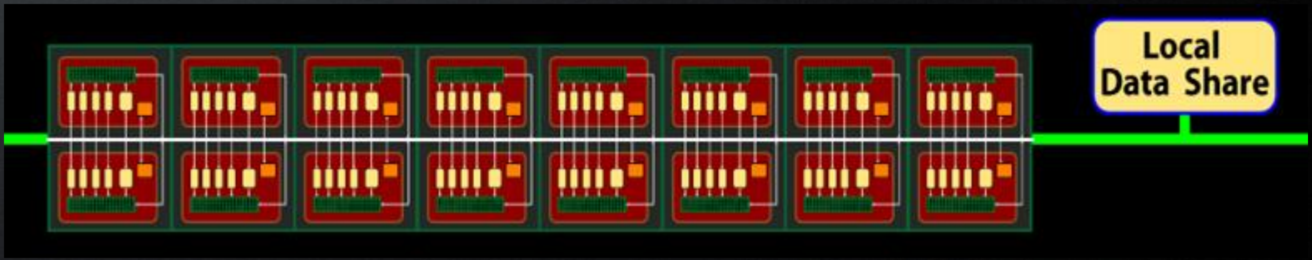
  = 64 threads on same instruction (Cypress)

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!



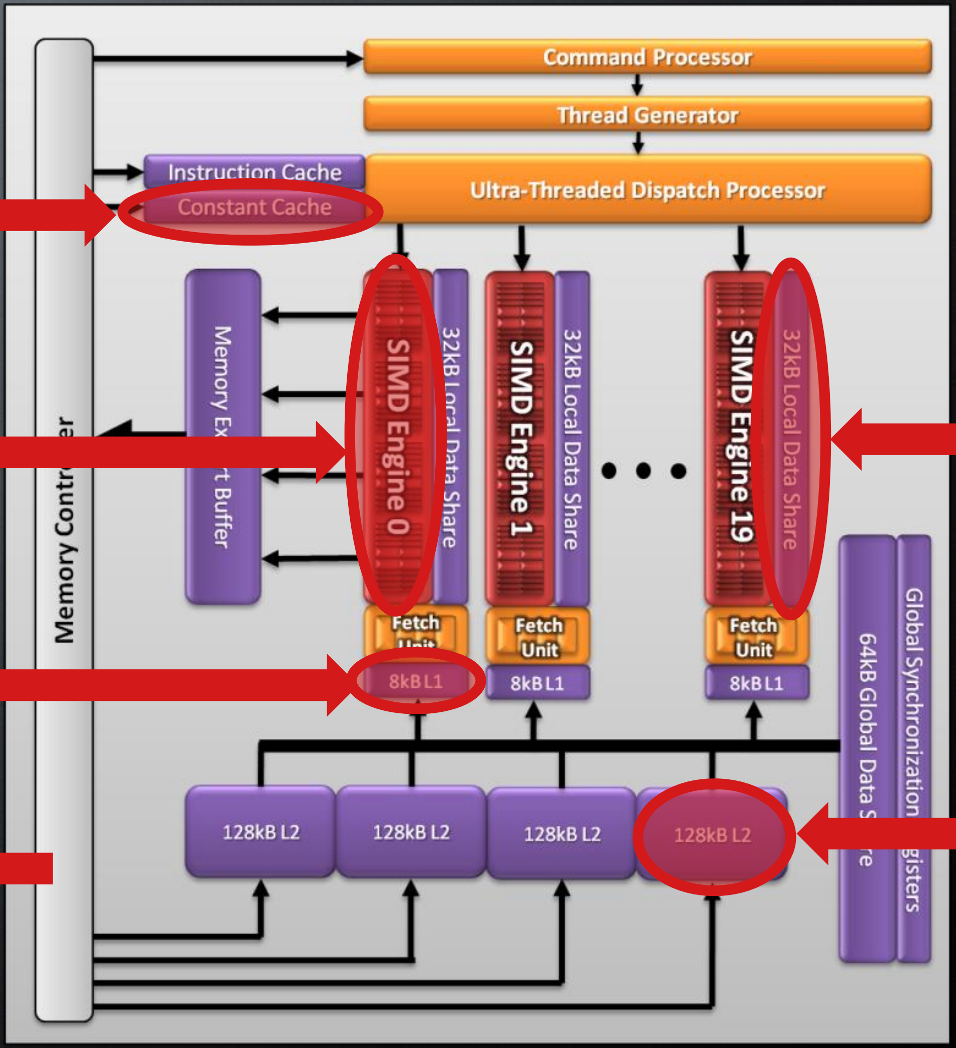
OpenCL View of AMD GPU

Constants (cached global)

Workgroups

Image cache

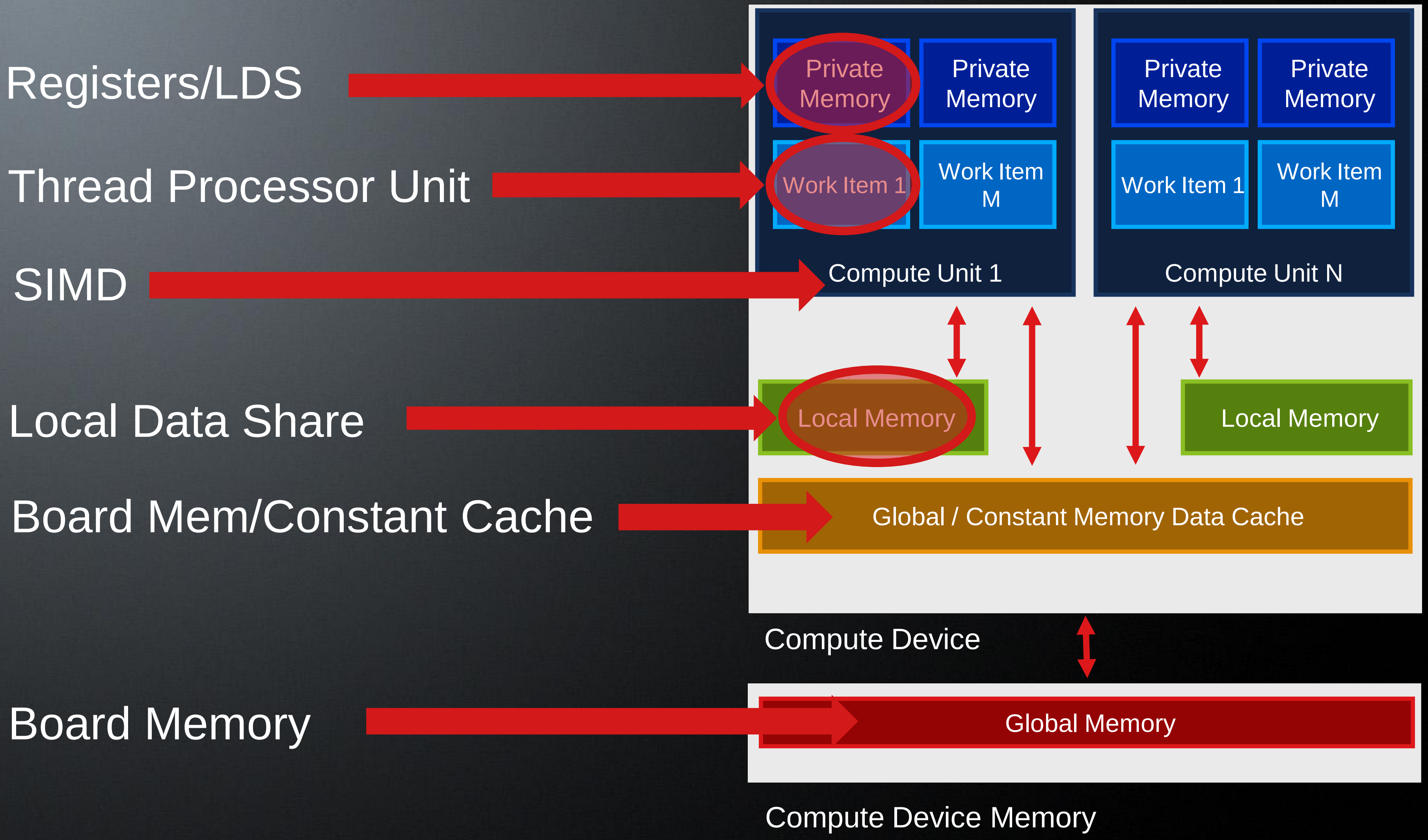
Global memory (uncached)



Local memory (user cache)

L2 cache

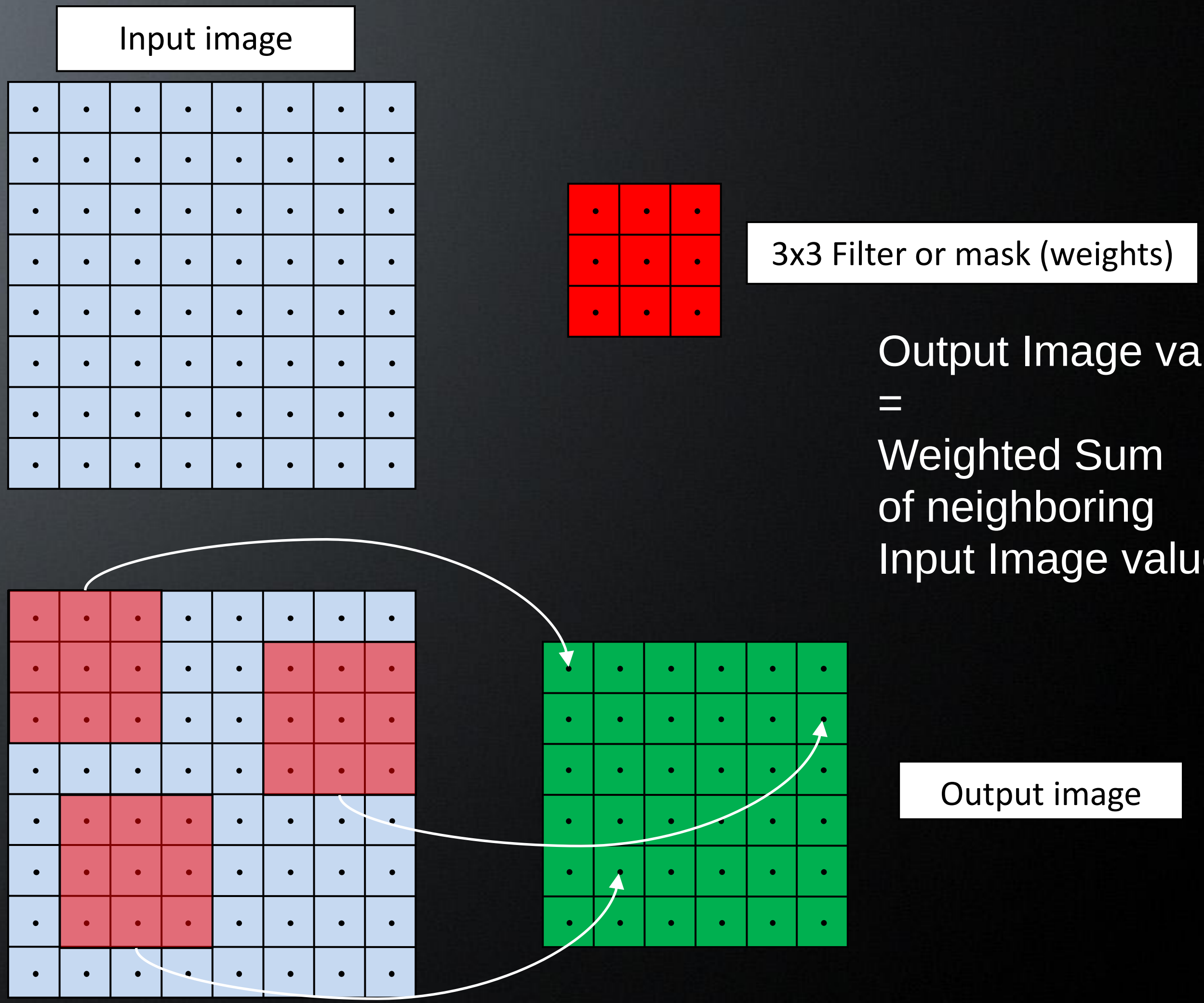
OpenCL™ Memory space on AMD GPU



Contents

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

Convolution algorithm



Output Image value
=
Weighted Sum
of neighboring
Input Image values

Convolution algorithm

```
FOR every pixel:
            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];
                }
            } //for (int r = 0...
            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

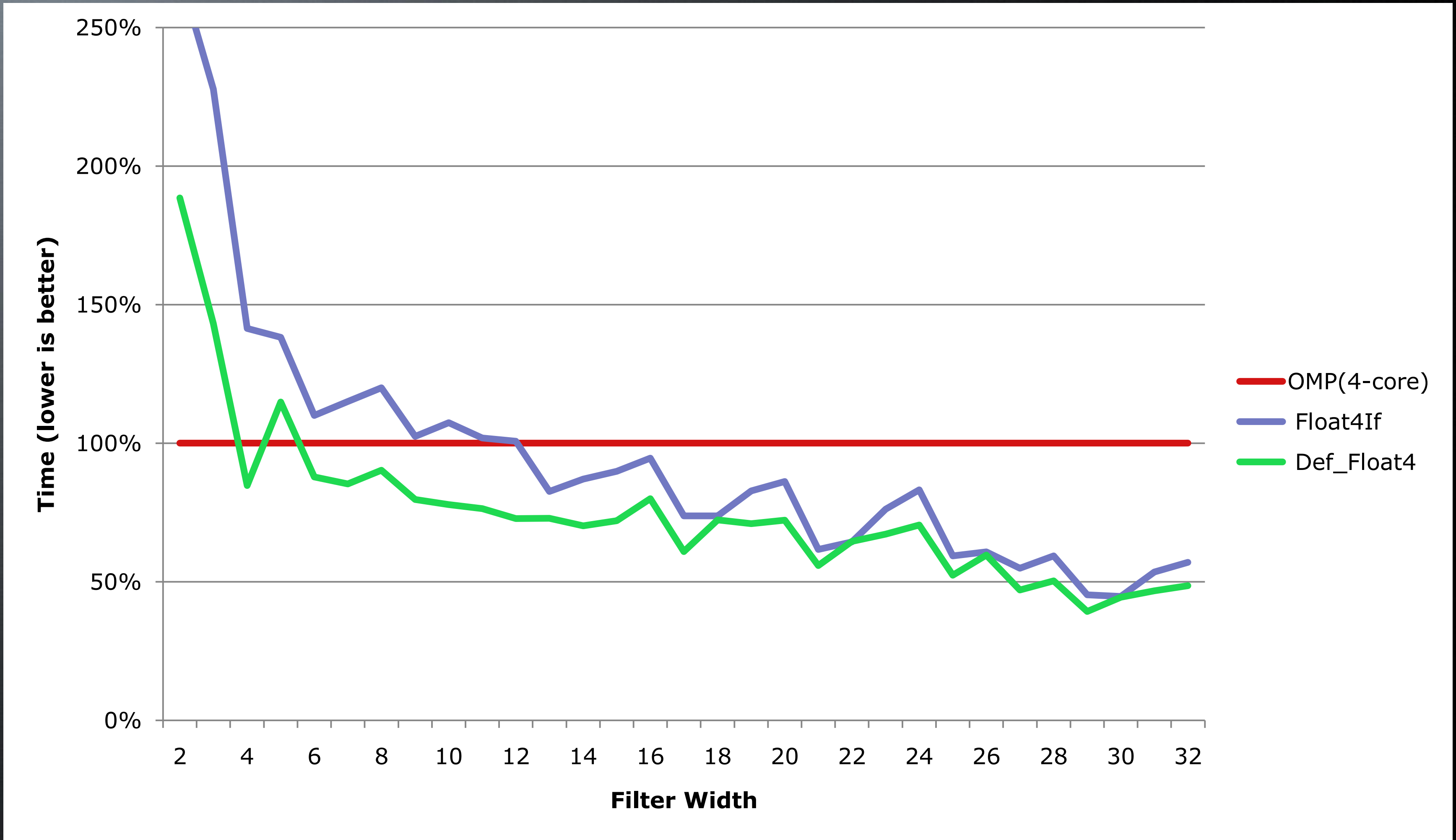
- CPU implementation:
  - Automatic multi-threading!
    - One CPU-thread per CPU-core
  - Highly efficient implementation
    - Each CPU-thread runs one or more OpenCL work-groups
    - Use large work-groups (max on CPU is 1024)

- 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



Contents

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

Convolution on GPU (naïve implementation)

```
__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 !

Convolution on GPU (Optimization 2)

- 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.

Convolution on GPU (Optimization 3)

- 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

Workgroup size	LDS	Texture
8x8	359 ms	346 ms
16x16	182 ms	207 ms

Convolution on GPU (Optimization 4)

- 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

Convolution on GPU (All combined)

- 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:

Optimization	LDS	Texture
Naïve implementaion	1511 ms	1511 ms
Data reuse (#1,2,3)	182 ms	207 ms
__constant float* filter(#4)	190 ms	160 ms
Unroll4, float4 input (#5)	90 ms	130 ms
Unroll4, float4 input, float4 filter (#5)	83 ms	127 ms
All above, __constant float* (#6 bad)	88 ms	158 ms
All above, __constant float4* (#6 good)	71 ms !	93 ms !

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

Convolution on GPU (Optimization 7)

- 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:

Optimization	LDS	Texture
Naïve implementaion	1511 ms	1511 ms
Data reuse (#1,2,3)	69 ms	128 ms
__constant float* filter(#4)	25 ms	127 ms
Unroll4, float4 input (#5)	68 ms	127 ms
Unroll4, float4 input, float4 filter (#5)	66 ms	127 ms
All above, __constant float* (#6 bad)	26 ms	127 ms
All above, __constant float4* (#6 good)	25 ms !!	63 ms !!

Questions and Answers

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 and Attribution