OpenCL on NVIDIA GPUs

Timo Stich, NVIDIA
Outline

- OpenCL and the CUDA architecture
- Optimizations
  - Memory
  - Launch Configurations
  - Math instructions
- Tools
  - Visual Profiler

OpenCL on NVIDIA GPUs
OpenCL and the CUDA Architecture

- Application Innovation
- Development Environment
- Leading Edge GPU Hardware

GPU Computing Applications

CUDA C

OpenCL™

DirectCompute

CUDA Fortran

NVIDIA GPU
with the CUDA Parallel Computing Architecture

OpenCL on NVIDIA GPUs
OpenCL Platform Model on the CUDA Architecture
Kernel Execution on Platform Model

- Each work-item is executed by a compute element
- Each work-group is executed on a compute unit
- Several concurrent work-groups can reside on one compute unit depending on work-group’s memory requirements and compute unit’s memory resources
- Each kernel is executed on a compute device
- On Tesla architecture, only one kernel can execute on a device at one time
Memory Spaces

Scope and Lifetime

OpenCL Terminology

CUDA Architecture

• Registers
  • 16 K (Tesla arch)
  • 32 K (Fermi arch)
  of 32-bit registers
  per compute unit

• On-chip
  • CUDA shared memory
  • 16 KB (Tesla arch)
  • 48 KB (Fermi arch)
  per compute unit

• Off-chip, cached
  • CUDA constant
  memory
  • 64 KB

• Off-chip
  • CUDA global memory
  • Up to 4 GB
Extensions are optional features exposed through OpenCL

Supported Khronos extensions:
- Image support
- OpenGL interop/sharing
- Double precision
- Atomic functions (base & extended)
- Byte addressable stores (write to pointers with types < 32-bits)

NVIDIA extensions:
- Advanced Compiler options
- Loop unroll hinting
- HW specific device attribute queries
Memory Hierarchy Review

- **Private storage**
  - Each thread has own private storage
  - Mostly registers (managed by the compiler)

- **Local memory**
  - Each work-group has its own local memory
  - Very low latency (a few cycles)
  - Very high throughput: 38-44 GB/s per multiprocessor
  - 30 multiprocessors per GPU -> over 1.1-1.4 TB/s

- **Global memory**
  - Accessible by all work-items as well as host (CPU)
  - High latency (400-800 cycles)
  - Throughput: 140 GB/s (1GB boards), 102 GB/s (4GB boards)
GMEM Coalescing: Compute Capability 1.2, 1.3

Possible GPU memory bus transaction sizes:
- 32B, 64B, or 128B
- Transaction segment must be aligned
  - First address = multiple of segment size

Hardware coalescing for each half-warp (16 threads):
- Memory accesses are handled per half-warsps
- Carry out the smallest possible number of transactions
- Reduce transaction size when possible

OpenCL on NVIDIA GPUs
HW Coalescing Steps

- Find the memory segment that contains the address requested by the lowest numbered active thread:
  - 32B segment for 8-bit data
  - 64B segment for 16-bit data
  - 128B segment for 32, 64 and 128-bit data.
- Find all other active threads whose requested address lies in the same segment
- Reduce the transaction size, if possible:
  - If size == 128B and only the lower or upper half is used, reduce transaction to 64B
  - If size == 64B and only the lower or upper half is used, reduce transaction to 32B
    - Applied even if 64B was a reduction from 128B
- Carry out the transaction, mark serviced threads as inactive
- Repeat until all threads in the half-warp are serviced
OpenCL on NVIDIA GPUs
Comparing Compute Capabilities

- **Compute capability < 1.2**
  - Requires threads in a half-warp to:
    - Access a single aligned 64B, 128B, or 256B segment
    - Threads must issue addresses in sequence
  - If requirements are not satisfied:
    - Separate 32B transaction for each thread

- **Compute capability 1.2 and 1.3**
  - Does not require sequential addressing by threads
  - Perf degrades gracefully when a half-warp addresses multiple segments
Experiment: Impact of Address Alignment

Assume half-warp accesses a contiguous region

Throughput is maximized when region is aligned on its size boundary

100% of bytes in a bus transaction are useful

Impact of misaligned addressing:

- 32-bit words, streaming code, Quadro FX5800 (102 GB/s)
- 0 word offset: 76 GB/s (perfect alignment, typical perf)
- 8 word offset: 57 GB/s (75% of aligned case)
- All others: 46 GB/s (61% of aligned case)
8-word (32B) offset from perfect alignment:
- Observed 75% of the perfectly aligned perf
- Segments starting at multiple of 32B
  - One 128B transaction (50% efficiency)
- Segments starting at multiple of 96B
  - Two 32B transactions (100% efficiency)
Address Alignment, 32-bit words

4-word (16B) offset (other offsets have the same perf):
- Observed 61% of the perfectly aligned perf
- Two types of segments, based on starting address
  - One 128B transaction (50% efficiency)
  - One 64B and one 32B transaction (67% efficiency)
Address Alignment, 64-bit words

Can be analyzed similarly to 32-bit case:

- **0B offset**: 80 GB/s (perfectly aligned)
- **8B offset**: 62 GB/s (78% of perfectly aligned)
- **16B offset**: 62 GB/s (78% of perfectly aligned)
- **32B offset**: 68 GB/s (85% of perfectly aligned)
- **64B offset**: 76 GB/s (95% of perfectly aligned)

Compare 0 and 64B offset performance:

- Both consume **100%** of the bytes
  - **64B**: two 64B transactions
  - **0B**: a single 128B transaction, slightly faster
GMEM Optimization Guidelines

- **Strive for perfect coalescing**
  - Align starting address (may require padding)
  - Warp should access within contiguous region

- **Process several elements per work-item**
  - Multiple loads get pipelined
  - Indexing calculations can often be reused

- **Launch enough threads to cover access latency**
  - GMEM accesses are not cached (changed with Fermi)
  - Latency is hidden by switching threads (warps)
Launch Configuration
Launch Configuration

How many work-items/work-groups to launch?

Key to understanding:

- Instructions are issued in order, one work-item is addressed by one thread
- A thread is blocked when one of the operands isn’t ready:
  - Memory read doesn’t block
- Latency is hidden by switching threads
- GMEM latency is 400-800 cycles

Conclusion:

- Need enough work-items to hide latency
Hiding Latency

**Arithmetic:**
- Need at least 6 warps (192 threads per SM)

**Memory:**
- Depends on the access pattern
- For GT200, 50% occupancy (512 threads per SM) is often sufficient
  - Occupancy = fraction of the maximum number of threads per multiprocessor

OpenCL on NVIDIA GPUs
Hiding Latency

**Arithmetic:**
- Need at least 6 warps (192) threads per SM

**Memory:**
- Depends on the access pattern
- For GT200, 50% occupancy (512 threads per SM) is often sufficient
  - Occupancy = fraction of the maximum number of threads per multiprocessor

Streaming 16M words: each thread reads, increments, writes 1 element

Throughput, 32-bit words

Throughput, 64-bit words
Launch Configuration: Summary

- Need enough total work-items to keep GPU busy
  - Currently (GT200), **512+** threads per SM is ideal
  - Fewer than **192** threads per SM **WILL NOT** hide arithmetic latency

- Work-group configuration
  - Work-items per group should be multiple of warp size (**32**)
  - SM can concurrently execute up to **8** groups
    - Really small groups prevent achieving good occupancy
    - Really large groups are less flexible
  - rule of thumb is **128-256 threads/block**, but use whatever is best for the application
Memory Throughput as Performance Metric
Many applications are memory throughput bound.

When coding from scratch:
- Start with memory operations first, achieve good throughput.
- Add the arithmetic, measuring perf as you go.

When optimizing:
- Measure effective memory throughput.
- Compare to the theoretical bandwidth.
  - 70-80% is very good, ~50% is good if arithmetic is nontrivial.

Measuring throughput:
- From the app point of view ("useful" bytes).
- From the hw point of view (actual bytes moved across the bus).
- The two are likely to be different.
  - Due to coalescing, discrete bus transaction sizes.
Measuring Memory Throughput

- **Visual Profiler** reports memory throughput
  - From **HW** point of view
  - Based on counters for one **TPC** (3 multiprocessors)
  - Need compute capability 1.2 or higher GPU

OpenCL on NVIDIA GPUs
Measuring Memory Throughput

- Latest Visual Profiler reports memory throughput
- From HW point of view
- Based on counters for one TPC (3 multiprocessors)
- Need compute capability 1.2 or higher GPU

<table>
<thead>
<tr>
<th>Profiler Output</th>
<th>Summary Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>GPU usec</td>
</tr>
<tr>
<td>fwd_3D_16x16_order</td>
<td>3.09382e+06</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>503094</td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td>168906</td>
</tr>
</tbody>
</table>
Measuring Memory Throughput

- Latest Visual Profiler reports memory throughput
- From HW point of view
- Based on counters for one TPC (3 multiprocessors)
- Need compute capability 1.2 or higher GPU

<table>
<thead>
<tr>
<th>Method</th>
<th>GPU usec</th>
<th>%GPU time</th>
<th>glob mem read throughput (GB/s)</th>
<th>glob mem write throughput (GB/s)</th>
<th>glob mem overall throughput (GB/s)</th>
<th>instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>fwd_3D_16x16_order8</td>
<td>3.09382e+06</td>
<td>82.15</td>
<td>46.9465</td>
<td>11.6771</td>
<td>58.6236</td>
<td>0.763973</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>503094</td>
<td>13.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td>168906</td>
<td>4.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Local Memory
Local Memory

**Uses:**
- Inter-thread communication within a group
- Cache data to reduce global memory accesses
- Use it to avoid non-coalesced access

**Organization:**
- 16 banks, 32-bit wide banks
- Successive 32-bit words belong to different banks

**Performance:**
- 32 bits per bank per 2 clocks per multiprocessor
- Imem accesses are per 16-threads (half-warp)
- **Serialization:** if \( n \) threads (out of 16) access the same bank, \( n \) accesses are executed serially
- **Broadcast:** \( n \) threads access the same word in one fetch
Bank Addressing Examples

No Bank Conflicts

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15

No Bank Conflicts

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 15

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15

OpenCL on NVIDIA GPUs
Bank Addressing Examples

2-way Bank Conflicts
- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 8
- Thread 9
- Thread 10
- Thread 11
- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15

8-way Bank Conflicts
- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7
- Thread 15
- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 8
- Bank 9
- Bank 15
Trick to Assess Impact On Performance

- Change all LMEM reads to the same value
  - All broadcasts = no conflicts
  - Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for LMEM writes
  - So, replace LMEM array indices with `threadIdx.x`
  - Can also be done to the reads
Bank Addressing Examples

- Banks are 32-bit, layout for different types

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Short</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Int</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Float</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Beware: 4x/2x Bank conflict when accessing contiguous chars/shorts
Additional Memories

- Constant (read-only) and Images
- Data resides in global memory
- Different read path:
  - includes caches (unlike current global memory access)
Constant Memory

- Data stored in global memory, read through a constant-cache path
  - `__constant` qualifier in declarations
  - Can only be read by GPU kernels
  - Limited to **64KB**
- To be used when all threads in a warp read the same address
  - Serializes otherwise
- **Throughput:**
  - **32 bits** per warp per clock per multiprocessor
Instruction Throughput / Control Flow
Two types of runtime math library functions

- **native_func()**: many map directly to hardware ISA
  - Fast but lower accuracy (see Programming Guide for full details)
  - Examples: `native_sinf(x), native_expf(x), native_powf(x, y)`

- **func()**: compile to multiple instructions
  - Slower but higher accuracy (5 ulp or less)
  - Examples: `sin(x), exp(x), pow(x, y)`

Use **-cl-mad-enable** compiler option

- Enables use of FMADs, which can lead to large performance gains

Investigate using the **-cl-fast-relaxed-math** compiler option

- Enables many aggressive compiler optimizations
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance
- Avoid diverging within a warp
  - Example with divergence:
    - if (threadIdx.x > 2) { ... } else { ... }
    - Branch granularity < warp size
  - Example without divergence:
    - if (threadIdx.x / WARP_SIZE > 2) { ... } else { ... }
    - Branch granularity is a whole multiple of warp size
CPU-GPU Interaction
Pinned (non-pageable) memory

- **Pinned memory enables:**
  - faster PCIe copies (~2x throughput)
  - memcopies asynchronous with CPU
  - memcopies asynchronous with GPU

- **Usage**
  - Allocate buffer with CL_MEM_ALLOC_HOST_PTR
  - Copy with buffer as source

- **Implication:**
  - pinned memory is essentially removed from host virtual memory
OpenCL Visual Profiler

- Analyze GPU HW performance signals, kernel occupancy, instruction throughput, and more
- Highly configurable tables and graphical views
- Save/load profiler sessions or export to CSV for later analysis
- Compare results visually across multiple sessions to see improvements
- Supported on Windows and Linux
- Included in the CUDA Toolkit
**OpenCL Information and Resources**

- **NVIDIA OpenCL Web Page:**
- **NVIDIA OpenCL Forum:**
- **NVIDIA driver, profiler, code samples for Windows and Linux:**
- **Khronos (current specification):**
  - [http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf](http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf)
- **Khronos OpenCL Forum:**