Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary
Overall Optimization Strategies

- Maximize parallel execution
  - Exposing data parallelism in algorithms
  - Choosing execution configuration
  - Overlap memory transfer with computation
- Maximize memory bandwidth
  - Keep the hardware busy
- Maximize instruction throughput
  - Get the job done with as few clock cycles as possible

We will talk about how to do those in NVIDIA GPUs.
Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary
Device contains 30 Streaming Multiprocessors (SMs)

Each SM contains
- 8 scalar processors
- 1 double precision unit
- 2 special function units
- shared memory (16 K)
- registers (16,384 32-bit=64 K)
**Execution Model**

**OpenCL**

- Work-item/thread
- Work-group

**Hardware**

- Work-item are executed by scalar processors
- Work-groups are executed on multiprocessors
- Work-groups do not migrate
- Several concurrent work-groups can reside on one SM - limited by SM resources (local and private memory)

- A kernel is launched as a grid of work-groups
- Only one kernel can execute on a device at one time

© NVIDIA Corporation 2009
• Work-groups divide into groups of 32 threads called warps.
• Warps always perform same instruction (SIMT)
• Warps are basic scheduling units
• 4 clock cycles to dispatch an instruction to all the threads in a warp
• A lot of warps can hide memory latency
OpenCL Memory Hierarchy

- Global: R/W per-kernel
- Constant: R per-kernel
- Local memory: R/W per-group
- Private: R/W per-thread
Mapping between OpenCL and CUDA

OpenCL

Compute Unit 1
- Private Memory
  - Work Item 1
- Private Memory
  - Work Item M

Compute Unit N
- Private Memory
  - Work Item 1
- Private Memory
  - Work Item M

Local Memory

Global / Constant Memory Data Cache

Compute Device Memory

CUDA

Multiprocessor
- Registers
  - Thread Processor
- Registers
  - Thread Processor

Shared Memory

Global / Constant Memory Data Cache

Compute Device Memory

© NVIDIA Corporation 2009
Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary
Overview of Memory Optimization

- Minimize host-device data transfer
- Coalesce global memory access
- Use local memory as a cache
Minimizing host-device data transfer

- Host device data transfer has much lower bandwidth than global memory access.
  - 8 GB/s (PCI-e, x16 Gen2) vs 141 GB/s (GTX 280)

- Minimize transfer
  - Intermediate data can be allocated, operated, de-allocated directly on GPU
  - Sometimes it’s even better to recompute on GPU, or call kernels that do not have performance gains

- Group transfer
  - One large transfer much better than many small ones
Coalescing

- Global memory latency: 400-600 cycles. The single most important performance consideration!
- Global memory access by threads of a half warp can be coalesced to one transaction for word of size 8-bit, 16-bit, 32-bit, 64-bit or two transactions for 128-bit.
- Global memory can be viewed as composing aligned segments of 16 and 32 words.

E.g. 32-bit word:
Coalescing in Compute Capability 1.0 and 1.1

K-th thread in a half warp must access the k-th word in a segment; however, not all threads need to participate.

- **Coalesces – 1 transaction**
  - ... ... ...
  - ... ... ...

- **Out of sequence – 16 transactions**
  - ... ... ...
  - ... ... ...
  - ... ... ...

- **Misaligned – 16 transactions**
  - ... ... ...
  - ... ... ...
  - ... ... ...
Coalescing in Compute Capability 1.2 and 1.3

- Coalescing for any pattern of access that fits into a segment size
- \# of transactions = \# of accessed segments

1 transaction - 64B segment

2 transactions - 64B and 32B segments

1 transaction - 128B segment
Example of Misaligned Accesses

__kernel void offsetCopy(__global float *odata,
            __global float* idata,
            int offset)
{
    int xid = get_global_id(0) + offset;
    odata[xid] = idata[xid];
}

GTX280 (compute capability 1.3) drops by a factor of 1.7 while FX 5600 (compute capability 1.0) drops by a factor of 8.
Example of Strided Accesses

```c
__kernel void strideCopy(__global float* odata, __global float* idata, int stride)
{
    int xid = get_global_id(0) * stride;
    odata[xid] = idata[xid];
}
```

Large strides often arise in applications. However, strides can be avoided using local memory.
Local Memory

- Latency ~100x smaller than global memory
- Cache data to reduce global memory access
- Use local memory to avoid non-coalesced global memory access
- Threads can cooperate through local memory
Caching Example 1: Matrix Multiplication

C = AxB

Uncached version:

```c
__kernel void simpleMultiply(__global float* a,
                             __global float* b,
                             __global float* c,
                             int N)
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Every thread corresponds to one entry in C.
Memory Access Pattern of a Half-Warp

A lot of repeated access to the same row of A. Un-coalesced in CC $\leq 1.1$. 
## Matrix Multiplication (cont.)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>NVIDIA GeForce GTX 280</th>
<th>NVIDIA Quadro FX 5600</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimization</td>
<td>8.8 GBps</td>
<td>0.62 GBps</td>
</tr>
<tr>
<td>Coalesced using shared memory to store a tile of A</td>
<td>14.3 GBps</td>
<td>7.34 GBps</td>
</tr>
<tr>
<td>Using shared memory to eliminate redundant reads of a tile of B</td>
<td>29.7 GBps</td>
<td>15.5 GBps</td>
</tr>
</tbody>
</table>
Matrix Multiplication (cont.)

Cashed and coalesced version:

```c
__kernel void coalescedMultiply(__global float* a,
    __global float* b,
    __global float* c,
    int N,
    __local float aTile[TILE_DIM][TILE_DIM])
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0);
    int y = get_local_id(1);
    aTile[y][x] = a[row*TILE_DIM+x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```
## Matrix Multiplication (cont.)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>NVIDIA GeForce GTX 280</th>
<th>NVIDIA Quadro FX 5600</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimization</td>
<td>8.8 GBps</td>
<td>0.62 GBps</td>
</tr>
<tr>
<td>Coalesced using shared memory to store a tile of A</td>
<td>14.3 GBps</td>
<td>7.34 GBps</td>
</tr>
<tr>
<td>Using shared memory to eliminate redundant reads of a tile of B</td>
<td>29.7 GBps</td>
<td>15.5 GBps</td>
</tr>
</tbody>
</table>
Coalescing Example 2: Matrix Transpose

Strided global mem access in naïve implementation, resulting in 16 transactions if stride > 16

Move the strided access into local memory read
## Matrix Transpose Performance

<table>
<thead>
<tr>
<th>Optimization</th>
<th>NVIDIA GeForce GTX 280</th>
<th>NVIDIA Quadro FX 5600</th>
</tr>
</thead>
<tbody>
<tr>
<td>No optimization</td>
<td>1.1 GBps</td>
<td>0.4 GBps</td>
</tr>
<tr>
<td>Using shared memory to coalesce global reads</td>
<td>24.8 GBps</td>
<td>13.3 GBps</td>
</tr>
<tr>
<td>Removing bank conflicts</td>
<td>30.3 GBps</td>
<td>15.6 GBps</td>
</tr>
</tbody>
</table>
A 2\textsuperscript{nd} order effect compared to global memory coalescing

Local memory is divide into banks.
- Successive 32-bit words assigned to successive banks
- Number of banks = 16 for CC 1.x

R/W different banks can be performed simultaneously.

Bank conflict: two R/W fall in the same bank, the access will be serialized.

Thus, accessing should be designed to avoid bank conflict
Outline

- Overview
- The CUDA architecture
- Memory optimization
- **Execution configuration optimization**
- Instruction optimization
- Summary
Work-group Heuristics

- # of work-groups > # of SM
  - Each SM has at least one work-group to execute
- # of work-groups / # of SM > 2
  - Multi work-groups can run concurrently on a SM
  - Work on another work-group if one work-group is waiting on barrier
- # of work-groups / # of SM > 100 to scale well to future device
Work-item Heuristics

- The number of work-items per work-group should be a multiple of 32 (warp size)
- Want as many warps running as possible to hide latencies
- Minimum: 64
- Larger, e.g. 256 may be better
- Depends on the problem, do experiments!
**Occupancy**

**Hide latency**: thread instructions are executed sequentially. So executing other warps when one warp is paused is the only way to hide latencies and keep the hardware busy.

**Occupancy**: ratio of active warps per SM to the maximum number of allowed warps

- 32 in GT 200, 24 in GeForce 8 and 9-series.
Global Memory Latency Hiding

- Enough warps can hide the latency of global memory access

- We need $400/4 = 100$ arithmetic instructions to hide the latency. For example, assume the code has 8 arithmetic instructions (4 cycle) for every one global memory access ($\sim 400$ cycles). Thus $100/8 \sim 13$ warps would be enough. This corresponds to 54% occupancy.
Register Dependency Latency Hiding

- If an instruction uses a result stored in a register written by an instruction before it, this is ~ 24 cycles latency.

- So, we need $24/4=6$ warps to hide register dependency latency. This corresponds to 25% occupancy.
Occupancy Considerations

- Increase occupancy to achieve latency hiding
- After some point (e.g. 50%), further increase in occupancy won’t lead to performance increase
- Occupancy is limited by resource usage:
  - Registers
  - Local memory
  - Scheduling hardware
Work-groups on a SM partition registers and local memory
If every thread uses 10 registers and every work-group has 256 work-items, then 3 work-groups use 256*10*3 < 8192. A 100% occupancy can be achieved.
However, if every thread uses 11 registers, since 256*11*3 > 8192, only 2 work-groups are allowed. So occupancy is reduced to 66%!
But, if work-group has 128 work-items, since 128*11*5 < 8192, occupancy can be 83%.
Other Resource Limitations on Occupancy

- Maximum number of warps.
- Maximum number of work-groups per SM: 8
- So occupancy calculation in realistic case is complicated, thus...
Outline

- Overview
- The CUDA architecture
- Memory optimization
- Execution configuration optimization
- Instruction optimization
- Summary
Instruction Throughput

Throughput: # of instructions per cycle

In SIMT architecture, if T is the number of operations per clock cycle

SM Throughtput = T/WarpSize

Maximizing throughput: using smaller number of cycles to get the job done
Arithmetic Instruction Throughput

- Int, and float add, shift, min, max, and float mul, mad: $T = 8$

- Int divide and modulo are expensive

- Avoid automatic conversion of double to float
  - Adding “f” to floating literals (e.g. 1.0f) because the default is double
Memory Instructions

- Use local memory to reduce global memory access

- Increase algorithm’s arithmetic intensity (the ratio of arithmetic to global memory access instructions). The higher of this ratio, the fewer of warps are required to hide global memory latency.
NVIDIA GPUs have a scalar architecture
- Use vector types in OpenCL for convenience, not performance
- Generally want more work-items rather than large vectors per work-item

Use the -cl-mad-enable compiler option
- Permits use of FMADs, which can lead to large performance gains

Investigate using the -cl-fast-relaxed-math compiler option
- Enables many aggressive compiler optimizations
There are two types of runtime math libraries
- Native\_function() map directly to the hardware level: faster but lower accuracy
- Function(): slower but higher accuracy

Use native math library whenever speed is more important than precision
Control Flow

- If branching happens within a warp, different execution paths must be serialized, increasing the total number of instructions.

- No penalty if different warps diverge
  - No divergence if controlling condition depends only on local_id/warp_size
OpenCL programs run on GPU can achieve great performance if one can:
- Maximize parallel execution
- Maximize memory bandwidth
- Maximize instruction throughput

Thank you and enjoy OpenCL!
Additional Topics

• Async transfer
• Zero copy
• Texture memory
• OpenCL extensions
• Interoperability
• Multi-GPU