CUDA is a programming system for utilizing the G80 processor for compute
- CUDA follows the architecture very closely
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor

Matches architecture features
Specific parameters not exposed

CUDA Device Memory Space Overview
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories

CUDA Optimization Priorities
- Memory coalescing is #1 priority
  - Optimize for locality
- Take advantage of shared memory
  - Very high bandwidth
  - Threads can cooperate to save work
- Use parallelism efficiently
  - Keep the GPU busy at all times
  - High arithmetic / bandwidth ratio
  - Many threads & thread blocks
- Leave bank conflicts and divergence for last!
  - 4-way and smaller conflicts are not usually worth avoiding if avoiding them will cost more instructions
CUDA Optimization Strategies

- Optimize Algorithms for the GPU
- Optimize Memory Access Pattern
- Take Advantage of On-Chip Shared Memory
- Use Parallelism Efficiently
- Use appropriate mechanisms

Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Modern DRAMs are Sensitive to Pattern

![Graph showing DRAM sensitivity to pattern](image)

Optimize Memory Pattern (“Coherence”)

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/local device memory
  - Sequential access by threads in a half-warp gets coalesced
- Optimize for spatial locality in cached texture memory
- Constant memory provides broadcast within SM
- In shared memory, avoid high-degree bank conflicts

Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
  - Use one / a few threads to load / compute data shared by all threads
  - Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalescable addressing
  - See the transpose SDK sample for an example

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing
  - Stride = 2

- 8-way Bank Conflicts
  - Linear addressing
  - Stride = 8

Data types and bank conflicts

- Structs
  - This has no conflicts if type of shared is 32 bits:

```cpp
foo = shared[baseIndex + threadIdx.x];
```

- But not if the data type is smaller
  - 4-way bank conflicts:

```cpp
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```

- 2-way bank conflicts:

```cpp
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks

- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Maximizing Instruction Throughput

- Minimize use of low-throughput instructions

- Maximize use of high-bandwidth memory
  - Maximize coherence of cached accesses
  - Minimize accesses to (uncached) global and local memory
  - Maximize coalescing of global memory accesses

- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
  - i.e. high ratio of math to memory transactions
  - Many concurrent threads

Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/speak (PCI-e x16) vs. 80 GB/speak (Quadro FX 5600)

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

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

Page-Locked Memory Transfers

- cuMemAllocHost() allows allocation of page-locked host memory

- Enables highest cudaMemcpy performance
  - ~3.2 GB/s common on PCI-e x16
  - ~4 GB/s measured on nForce 680i motherboards

- See the “bandwidthTest” CUDA SDK sample

- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Device memory to host memory bandwidth much lower than device memory to device bandwidth

- 4GB/speak (PCI-e x16) vs. 80 GB/speak (Quadro FX 5600)

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

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

Page-Locked Memory Transfers

- cuMemAllocHost() allows allocation of page-locked host memory

- Enables highest cudaMemcpy performance
  - ~3.2 GB/s common on PCI-e x16
  - ~4 GB/s measured on nForce 680i motherboards

- See the “bandwidthTest” CUDA SDK sample

- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
Optimizing threads per block

- Choose total threads in a grid
  - Choose block size and number of blocks to maximize occupancy.

Occupancy = # of warps running concurrently on a multiprocessor divided by maximum # of warps that can run concurrently

(Demonstrate CUDA Occupancy Calculator)

Grid/Block Size Heuristics

- # of blocks / # of multiprocessors > 1
  - Ensures all multiprocessors have at least a block to execute
  - Per-block resources at most half of total available
    - Shared memory and registers
    - Multiple blocks can run concurrently in a multiprocessor
    - If multiple blocks exist that aren’t all waiting at a __syncthreads() function, machine can stay busy

- # of blocks / # of multiprocessors > 2
  - So multiple blocks run concurrently in a multiprocessor
  - # of blocks > 100 to scale to future devices
    - Blocks stream through machine in pipeline fashion
    - 1000 blocks per grid will scale across multiple generations

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance
  - BUT...
  - Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
    - It all comes down to arithmetic intensity and available parallelism

Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps

- More threads per block = better memory latency hiding
  - But, more threads per block = fewer regs per thread
    - Kernel invocations can fail if too many registers are used

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation!

Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only accesses registers assigned to itself

Communication

- How do threads communicate?
  - Remember the execution model:
    - Data parallel streams that represent independent vertices, triangles, fragments, and pixels in the graphics world
    - These never communicate

- Some communication allowed in compute mode:
  - Shared memory for threads in a thread block
    - No special communication within warp or using registers
  - No communication between thread blocks
  - Kernels communicate through global device memory

  - Mechanisms designed to ensure portability
Synchronization

• Do threads need to synchronize?
  - Basically no communication allowed

• Threads in a block share memory – need sync
  - Warps scheduled OoO, can't rely on warp order
  - Barrier command for all threads in a block
  - __syncthreads()

• Blocks cannot synchronize
  - Implicit synchronization at end of kernel
  - Can build some sync with atomic operations

Atomic Operations

• Exception to communication between blocks

• Atomic read-modify-write
  - Shared memory
  - Global memory

• Simple ALU operations
  - Add, subtract, AND, OR, min, max, inc, dec

• Exchange operations
  - Compare-and-swap, exchange