Lecture 16 - CUDA Optimization Strategies

Mattan Erez

The University of Texas at Austin
Outline

- CUDA
  - Development process
  - Performance Optimization
  - Syntax

- Most slides courtesy Massimiliano Fatica (NVIDIA)
Compute Unified Device Architecture

- 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 Programming Model: A Highly Multithreaded Coprocessor

• The GPU is viewed as a compute **device** that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

• Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads

• **Differences between GPU and CPU threads**
  - **GPU threads** are extremely lightweight
    - Very little creation overhead
  - **GPU needs 1000s of threads for full efficiency**
    - Multi-core CPU needs only a few
CUDA: C on the GPU

• A simple, explicit programming language solution

• Extend only where necessary

```c
__global__ void KernelFunc(...);
__shared__ int SharedVar;

KernelFunc<<< 500, 128 >>>(...);
```

• Explicit GPU memory allocation
  - `cudaMalloc()`, `cudaFree()`

• Memory copy from host to device, etc.
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
Compilation

• Any source file containing CUDA language extensions must be compiled with \texttt{nvcc}

• \texttt{NVCC} is a \textit{compiler driver}
  – Works by invoking all the necessary tools and compilers like \texttt{cudacc}, \texttt{g++}, \texttt{cl}, ...

• \texttt{NVCC} can output:
  – Either \texttt{C} code (CPU Code)
    • That must then be compiled with the rest of the application using another tool
  – Or PTX object code directly

• Any executable with CUDA code requires two dynamic libraries:
  – The CUDA runtime library (\texttt{cudart})
  – The CUDA core library (\texttt{cuda})
Compiling CUDA

- C/C++ CUDA Application
- NVCC
- PTX Code
- PTX to Target Compiler
- G80, ..., GPU

Target code
Compiling CUDA

C/C++ CUDA Application

NVCC

PTX Code

PTX to Target Compiler

G80...GPU

Target code

Virtual

Physical
Debugging Using the Device Emulation Mode

• An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

• When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of __syncthreads
Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs
  - Different instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host.
Outline

• CUDA
  – Development process
  – Performance Optimization
    • Optimize Algorithms for the GPU
    • Optimize Memory Access Pattern
    • Take Advantage of On-Chip Shared Memory
    • Use Parallelism Efficiently
  – Syntax

• Most slides courtesy Massimiliano Fatica (NVIDIA)
CUDA Optimization Priorities

• Memory coalescing is #1 priority
  - Highest I/\$ optimization
  - 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
Parameterize Your Application

• Parameterization helps adaptation to different GPUs

• GPUs vary in many ways
  – # of multiprocessors
  – Shared memory size
  – Register file size
  – Threads per block
  – Memory bandwidth

• You can even make apps self-tuning (like FFTW)
  – “Experiment” mode discovers and saves optimal config
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

Inorder  Row  Row+Col

%peak BW

1 1x1rd 1x1rdcf 1x1rw 1x1nwcf 1x40rd 48x48rd  cr1rd  cr1rw  r1rd  r1nw  r4rd  r4nw
Optimize Memory Pattern (“Coherence”)

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
  - Sequential access by threads in a half-warp get 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-coalesceable addressing
  - See the transpose SDK sample for an example
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 use of shared 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/s peak (PCI-e x16) vs. 80 GB/s peak (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

• Given: 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
  - So 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 coexist that aren’t all waiting at a __syncthreads(), 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 $\Rightarrow$ better memory latency hiding
• But, more threads per block $\Rightarrow$ 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
  – 192 or 256 threads a better choice
    • Usually still enough regs to compile and invoke successfully
  – This all depends on your computation!
    • Experiment!