CUDA Performance Optimization, Multi-GPU, and Graphics Interop

Mark Harris
NVIDIA Corporation
Outline

- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Graphics Interoperability
Outline

- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Graphics Interoperability
Execution Configuration

• How many threads / thread blocks to launch?

Key to understanding:
• Instructions are issued in order
• A thread blocks when one of its operands is not ready
  • Memory loads don’t block
• Latency is hidden by switching threads
• GMEM latency is 400-800 cycles

Conclusion:
• Need enough threads 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

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently

Limited by resource usage:
- Registers
- Shared memory
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)
Hiding Latency

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

Read-after-write register dependency
- Instruction’s result can be read ~24 cycles later

Scenarios:

CUDA:
\[
x = y + 5;
\]
\[
z = x + 3;
\]
\[
s\_data[0] += 3;
\]

PTX:
\[
\text{add.f32 } f3, f1, f2
\]
\[
\text{add.f32 } f5, f3, f4
\]
\[
\text{ld.shared.f32 } f3, [r31+0]
\]
\[
\text{add.f32 } f3, f3, f4
\]

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
- Threads do not have to belong to the same thread block
Launch Configuration: Summary

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

- Thread block configuration
  - Threads per block should be a multiple of warp size (32)
  - SM can concurrently execute up to 8 threadblocks
    - Really small thread blocks prevent achieving good occupancy
    - Really large thread blocks are less flexible
  - I generally use 128-256 threads/block, but use whatever is best for the application
Register Pressure

Hide latency by using more threads per multiprocessor

Limiting Factors:
- Number of registers per kernel
  - 8K/16K per multiprocessor, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per multiprocessor, partitioned among concurrent threadblocks

Compile with \texttt{--ptxas-options=-v} flag

Use \texttt{--maxrregcount=N} flag to NVCC
- \(N\) = desired maximum registers / kernel
- At some point “spilling” into local memory may occur
  - Reduces performance – local memory is slow
  - Problem: applies to all kernels in a compilation unit
  - Solution: launch bounds
CUDA 3.0 Launch Bounds

- Mechanism to tell compiler desired block size for a kernel
  - Supersedes maxrregcount.

- Can be used with one or two parameters:
  - `maxThreads`: specify max # threads that will ever launch in one block
    - Required for correctness: compiler may use too many registers without it
  - `minBlocks`: specify min # blocks that will ever launch at once
    - Compiler may use this for optimization: can increase or reduce registers used

- Launch bounds is specified per kernel:
  - `__global__ void __launch_bounds__(128) foo() {}`
  - `__global__ void __launch_bounds__(256, 2) foo() {}`
CUDA GPU Occupancy Calculator

1. Just follow steps 1, 2, and 3 below (or click here for help)
2. Enter your resource usage:
   - Threads Per Block:
   - Registers Per Thread:
   - Shared Memory Per Block (optional):
3. (Don't edit anything below this line)
   - 3) GPU Occupancy Data is displayed here and in the graphs
      - Active Threads per Multiprocessor
      - Active Warps per Multiprocessor
      - Active Thread Blocks per Multiprocessor
      - Occupancy of each Multiprocessor
      - Maximum Simultaneous Blocks per GPU
4. (Note: this assumes there are at least this many blocks)
   - Physical Limits for GPU:
     - Multiprocessors per GPU
     - Threads per Warp
     - Warps per Multiprocessor
     - Threads / Multiprocessor
     - Thread Blocks / Multiprocessor
   - Total # of 32-bit registers / Multiprocessor
   - Shared Memory per Multiprocessor (bytes)
5. Allocation Per Thread Block
   - Warp
   - Registers
   - Shared Memory
6. These data are used in computing the occupancy data in the graphs
7. Maximum Thread Blocks Per Multiprocessor
8. Limited by No. Warps / Multiprocessor
9. Limited by Registers / Multiprocessor
10. Limited by Shared Memory / Multiprocessor
11. Thread block limit per multiprocessor is the minimum of those
12. CUDA Occupancy Calculator
13. Version:
14. Copyright & License
15. Help | GPU Data | Copyright & License

© NVIDIA Corporation 2009
Parameterize Your Application

Parameterization helps adaptation to different GPUs

GPUs vary in many ways
- # of multiprocessors
- Memory bandwidth
- Shared memory size
- Register file size
- Max. threads per block

You can even make apps self-tuning (like FFTW and ATLAS)
- “Experiment” mode discovers and saves optimal configuration
Outline

- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Graphics Interoperability
### Run-time Math Library and Intrinsics

**Two types of runtime math library functions**
- **__func():** many map directly to hardware ISA
  - Fast but lower accuracy (see CUDA Programming Guide for full details)
  - Examples: __sinf(x), __expf(x), __powf(x, y)
- **func():** compile to multiple instructions
  - Slower but higher accuracy (5 ulp or less)
  - Examples: sin(x), exp(x), pow(x, y)

**A number of additional intrinsics:**
- __sincosf(), __frcp_rz(), ...
- Explicit IEEE rounding modes (rz,rn,ru,rd)
Control Flow

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths (if-else, etc.)
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on perf
- 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
Profiler and Instruction Throughput

Profiler counts per multiprocessor:
- Divergent branches
- Warp serialization
- Instructions issues

Visual Profiler derives:
- Instruction throughput
  - Fraction of fp32 arithmetic instructions that could have been issued in the same amount of time
    - So, not a good metric for code with fp64 arithmetic or transcendental operations
- Extrapolated from one multiprocessor to GPU
Profiler and Instruction Throughput

Visual Profiler derives:

- Instruction throughput
  - Fraction of fp32 arithmetic instructions that could have been issued in the same amount of time
    - So, not a good metric for code with fp64 arithmetic or transcendentals
  - Extrapolated from one multiprocessor to GPU
Tricks with Code Comments

- **Comment out arithmetic**
  - To assess memory-only performance
  - Works if memory access is not data-dependent

- **Comment out gmem accesses**
  - To assess arithmetic-only performance
  - Works if computation is not data dependent
  - Eliminating reads is straightforward, eliminating writes is trickier
    - Compiler will throw away all code it deems as not contributing to output
    - Workaround: precede writes with an if-statement that always fails
      - For example: `if( threadIdx.x == -2 )`
GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings
- CPU operations aren’t strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
- Floating-point arithmetic is not associative!
FP Math is Not Associative!

- In symbolic math, \((x+y)+z == x+(y+z)\)
- This is not necessarily true for floating-point addition
  - Try \(x = 10^{30}, \ y = -10^{30} \) and \(z = 1\) in the above equation

- When you parallelize computations, you potentially change the order of operations

- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution
Outline

- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Graphics Interoperability
Why Multi-GPU Programming?

Many systems contain multiple GPUs:
- Servers (Tesla/Quadro servers and desksides)
- Desktops (2- and 3-way SLI desktops, GX2 boards)
- Laptops (hybrid SLI)

Additional processing power
- Increasing processing throughput

Additional memory
- Some problems do not fit within a single GPU memory
Multi-GPU Memory

- **GPUs do not share global memory**
  - One GPU cannot access another GPUs memory directly

- **Inter-GPU communication**
  - Application code is responsible for moving data between GPUs
  - Data travels across the PCIe bus
    - Even when GPUs are connected to the same PCIe switch
CPU-GPU Context

- A CPU-GPU context must be established before calls are issued to the GPU.
- CUDA resources are allocated per context.
- A context is established by the first CUDA call that changes state:
  - `cudaMalloc`, `cudaMemcpy`, `cudaFree`, kernel launch, ...
- A context is destroyed by one of:
  - Explicit `cudaThreadExit()` call
  - Host thread terminating
Run-Time API Device Management:

- A host thread can maintain one context at a time
  - GPU is part of the context and cannot be changed once a context is established
  - Need as many host threads as GPUs
  - Note that multiple host threads can establish contexts with the same GPU
  - Driver handles time-sharing and resource partitioning

- GPUs have consecutive integer IDs, starting with 0

- Device management calls:
  - `cudaGetDeviceCount(int *num_devices)`
  - `cudaSetDevice(int device_id)`
  - `cudaGetDevice(int *current_device_id)`
  - `cudaThreadExit()`
Choosing a Device

Properties for a given device can be queried
- No context is necessary or is created
- `cudaGetDeviceProperties(cudaDeviceProp *properties, int device_id)`
- This is useful when a system contains different GPUs

Explicit device set:
- Select the device for the context by calling `cudaSetDevice()` with the chosen device ID
  - Must be called prior to context creation
  - Fails if a context has already been established
  - One can force context creation with `cudaFree(0)`

Default behavior:
- Device 0 is chosen when no explicit `cudaSetDevice` is called
  - Note this will cause multiple contexts with the same GPU
  - Except when driver is in the `exclusive mode` (details later)
Ensuring One Context Per GPU

Two ways to achieve:
- Application-control
- Driver-control

Application-control:
- Host threads negotiate which GPUs to use
  - For example, OpenMP threads set device based on OpenMP thread ID
  - Pitfall: different applications are not aware of each other’s GPU usage
- Call `cudaSetDevice()` with the chosen device ID
Driver-control (Exclusive Mode)

**To use exclusive mode:**
- Administrator sets the GPU to exclusive mode using **SMI**
  - **SMI** (System Management Tool) is provided with Linux drivers
- Application: do not explicitly set the GPU in the application

**Behavior:**
- Driver will implicitly set a GPU with no contexts
- Implicit context creation will fail if all GPUs have contexts
- The first state-changing CUDA call will fail and return an error

**Device mode can be checked by querying its properties**
Inter-GPU Communication

- Application is responsible for moving data between GPUs:
  - Copy data from GPU to host thread A
  - Copy data from host thread A to host thread B
    - Use any CPU library (MPI, ...)
  - Copy data from host thread B to its GPU
- Use asynchronous memcopies to overlap kernel execution with data copies
- Lightweight host threads (OpenMP, pthreads) can reduce host-side copies by sharing pinned memory
  - Allocate with `cudaHostAlloc(...)`
Example: Multi-GPU 3DFD

- **3DFD Discretization of the Seismic Wave Equation**
  - $8^{th}$ order in space, $2^{nd}$ order in time, regular grid
- **Fixed $x$ and $y$ dimensions, varying $z$**
- **Data is partitioned among GPUs along $z$**
  - Computation increases with $z$, communication (per node) stays constant
  - A GPU has to exchange 4 $xy$-planes (ghost nodes) with each of its neighbors
- **Cluster:**
  - 2 GPUs per node
  - Infiniband SDR network
Linear scaling is achieved when computation time exceeds communication time
- Single GPU performance is ~3.0 Gpoints/s
- OpenMP case requires no copies on the host side (shared pinned memory)
  - Communication time includes only PCIe transactions
- MPI version uses MPI_Sendrecv, which invokes copies on the host side
  - Communication time includes PCIe transactions and host memcopies
3 or more cluster nodes

- Times are per cluster node
- At least one cluster node needs two MPI communications, one with each of the neighbors
Performance Example: 3DFD

- Single GPU performance is \(~3,000\) MPoints/s
- Note that 8x scaling is sustained at \(z > 1,300\)
  - Exactly where computation exceeds communication
Outline

- Execution Configuration Optimizations
- Instruction Optimizations
- Multi-GPU
- Graphics Interoperability
OpenGL Interoperability

- OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  - Vertex buffer objects
  - Pixel buffer objects
- Direct3D vertex and pixel objects can also be mapped
- Data can be accessed like any other global data in the device code
- Image data can be displayed from pixel buffer objects using `glDrawPixels` / `glTexImage2D`
  - Requires copy in video memory, but still fast
OpenGL Interop Steps

- Register a buffer object with CUDA
  - `cudaGLRegisterBufferObject(GLuint buffObj);`
  - OpenGL can use a registered buffer only as a source
  - Unregister the buffer prior to rendering to it by OpenGL

- Map the buffer object to CUDA memory
  - `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
  - Returns an address in global memory
  - Buffer must registered prior to mapping

- Launch a CUDA kernel to process the buffer

- Unmap the buffer object prior to use by OpenGL
  - `cudaGLUnmapBufferObject(GLuint buffObj);`

- Unregister the buffer object
  - `cudaGLUnregisterBufferObject(GLuint buffObj);`
  - Optional: needed if the buffer is a render target

- Use the buffer object in OpenGL code
Interop Scenario: Dynamic CUDA-generated texture

- Register the texture PBO with CUDA
- For each frame:
  - Map the buffer
  - Generate the texture in a CUDA kernel
  - Unmap the buffer
  - Update the texture
  - Render the textured object

```c
unsigned char *p_d = 0;
cudaGLMapBufferObject((void**)&p_d, pbo);
prepTexture<<<height, width>>>(p_d, time);
cudaGLUnmapBufferObject(pbo);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, texID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_BGRA, GL_UNSIGNED_BYTE, 0);
```
Interop Scenario:
Frame Post-processing by CUDA

For each frame:
- Render to PBO with OpenGL
- Register the PBO with CUDA
- Map the buffer
- Process the buffer with a CUDA kernel
- Unmap the buffer
- Unregister the PBO from CUDA

```
unsigned char *p_d=0;
cudaGLRegisterBufferObject(pbo);
cudaGLMapBufferObject((void**)&p_d, pbo);
postProcess<<<blocks,threads>>>(p_d);
cudaGLUnmapBufferObject(pbo);
cudaGLUnregisterBufferObject(pbo);
...```
Some Changes For Fermi

- Memory operations are done per warp (32 threads)
  - Global memory, Shared memory (per half-warp on GPUs before Fermi)
- Shared memory:
  - 16KB or 48KB
  - Now 32 banks, each 32 bits wide
  - No bank conflicts when accessing 8-byte words
- L1 cache per multiprocessor
  - Should help with misaligned access, strides access, some register spilling
- Much improved dual issue:
  - Can dual issue fp32 pairs, fp32-mem, fp64-mem, etc.
- Others...
CUDA Performance
Optimization

Questions?