Optimizing CUDA – Part II

Outline

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

Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

- 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

Blocks per Grid Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Register Dependency

- Read-after-write register dependency
  - Instruction's result can be read ~24 cycles later
  - Scenarios: CUDA: PTX:

```
x = y + 5;
z = x + 3;
```

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

```
s_data[0] += 3;
```

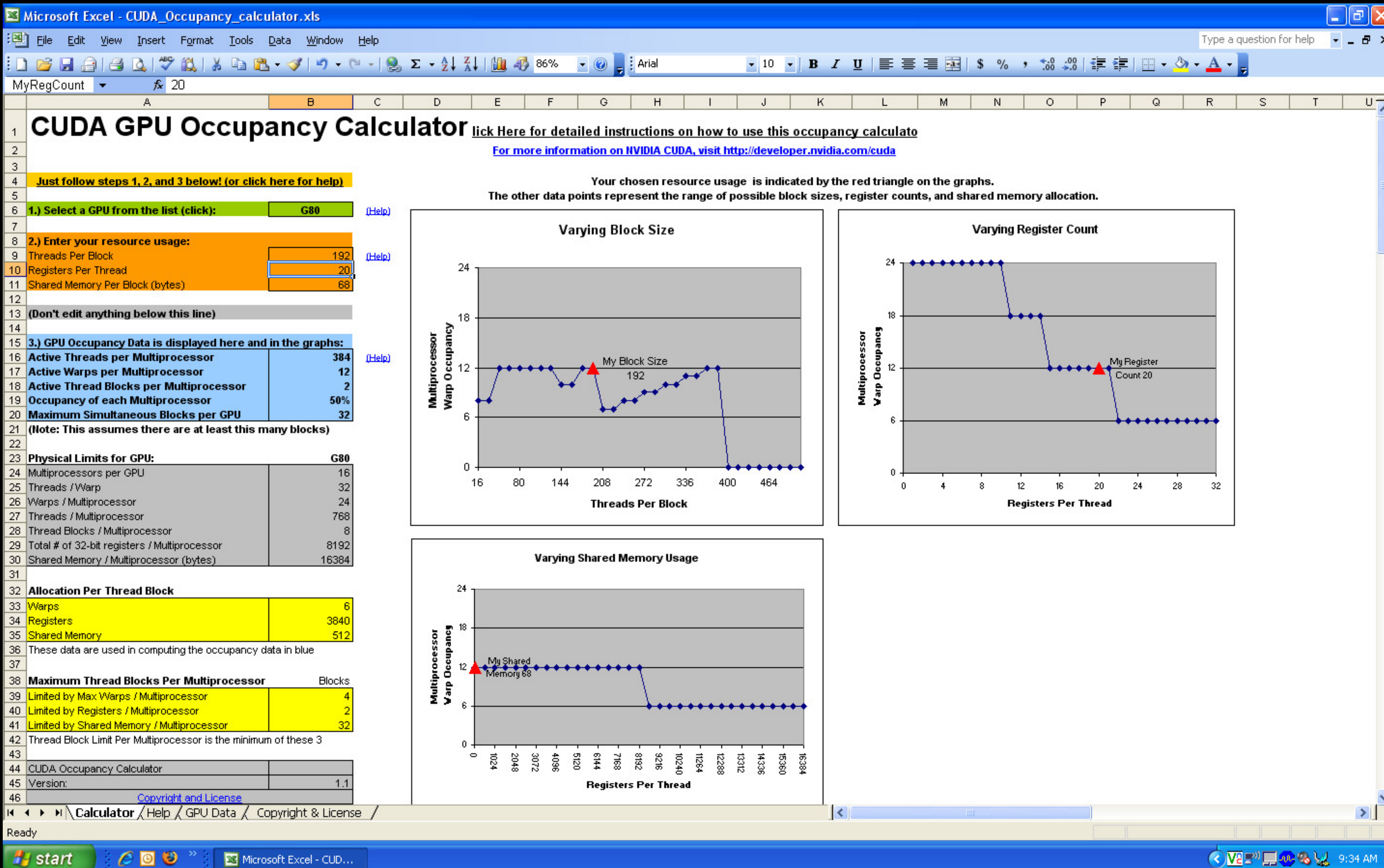
```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- 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

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 `–ptxas-options=-v` flag
- Use `–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

Occupancy Calculator



Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing
- Want to run as many warps as possible per multiprocessor (hide latency)
- Multiprocessor can run up to 8 blocks at a time

- 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, so experiment!

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)**

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

CUDA Instruction Performance

- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles
- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth
- "Cycle" refers to the multiprocessor clock rate
  - 1.3 GHz on the Tesla C1060, for example

Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global 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

Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use `__mul24()/__umul24()` intrinsics for 4-cycle 24-bit int multiply

- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: `foo%n==foo&(n-1)` if `n` is a power of 2

Runtime Math Library

- **There are two types of runtime math operations in single precision**
  - `__funcf()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: `__sinf(x)`, `__expf(x)`, `__powf(x,y)`
  - `funcf()` : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: `sinf(x)`, `expf(x)`, `powf(x,y)`
- **The `-use_fast_math` compiler option forces every `funcf()` to compile to `__funcf()`**

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

Control Flow Instructions

- **Main performance concern with branching is divergence**
  - **Threads within a single warp take different paths**
  - **Different execution paths must be serialized**

- **Avoid divergence when branch condition is a function of thread ID**
  - **Example with divergence:**
    - `if (threadIdx.x > 2) { }`
    - **Branch granularity < warp size**
  - **Example without divergence:**
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - **Branch granularity is a whole multiple of warp size**

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 OpenMPI 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

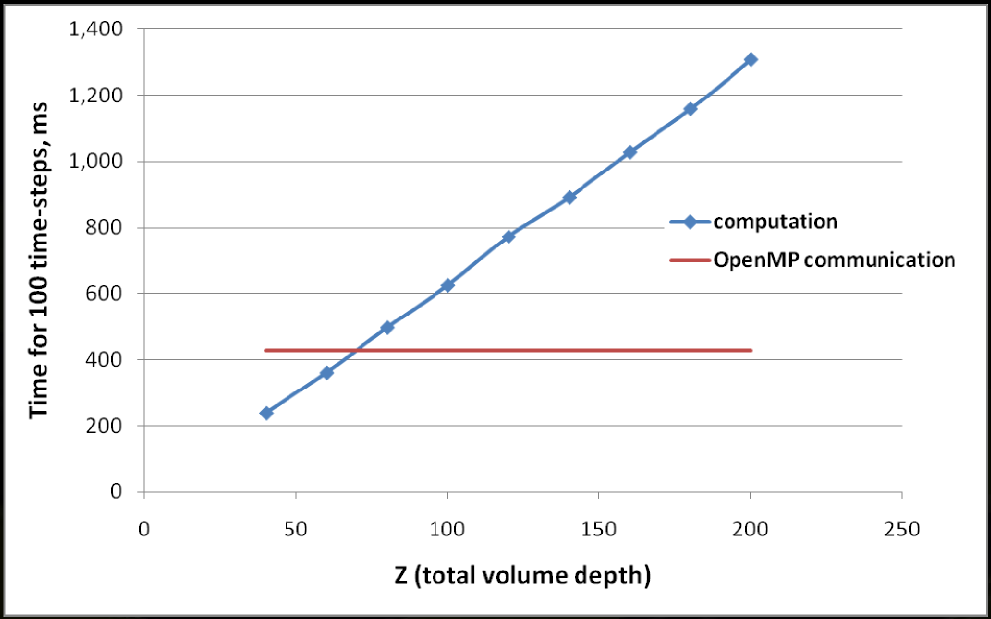
Inter-GPU Communication

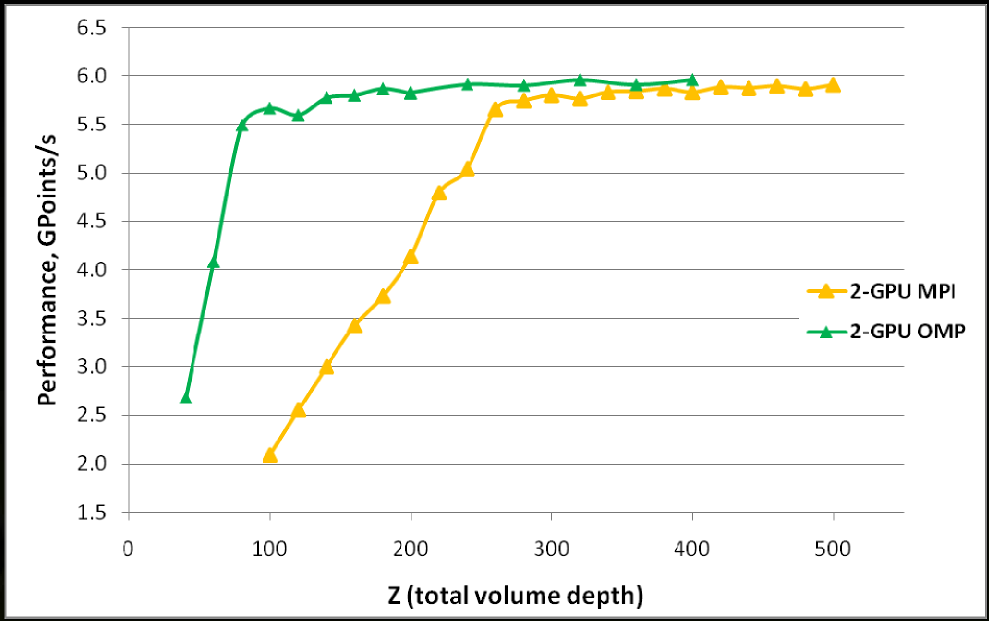
- **Application is responsible for moving data between GPUs:**
  - **Copy data from GPU to host thread A**
  - **Copy data from host thread A to host thread B**
    - **Use any CPU library (MPI, ...)**
  - **Copy data from host thread B to its GPU**
- **Use asynchronous memcopies to overlap kernel execution 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**
  - 8th order in space, 2nd 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

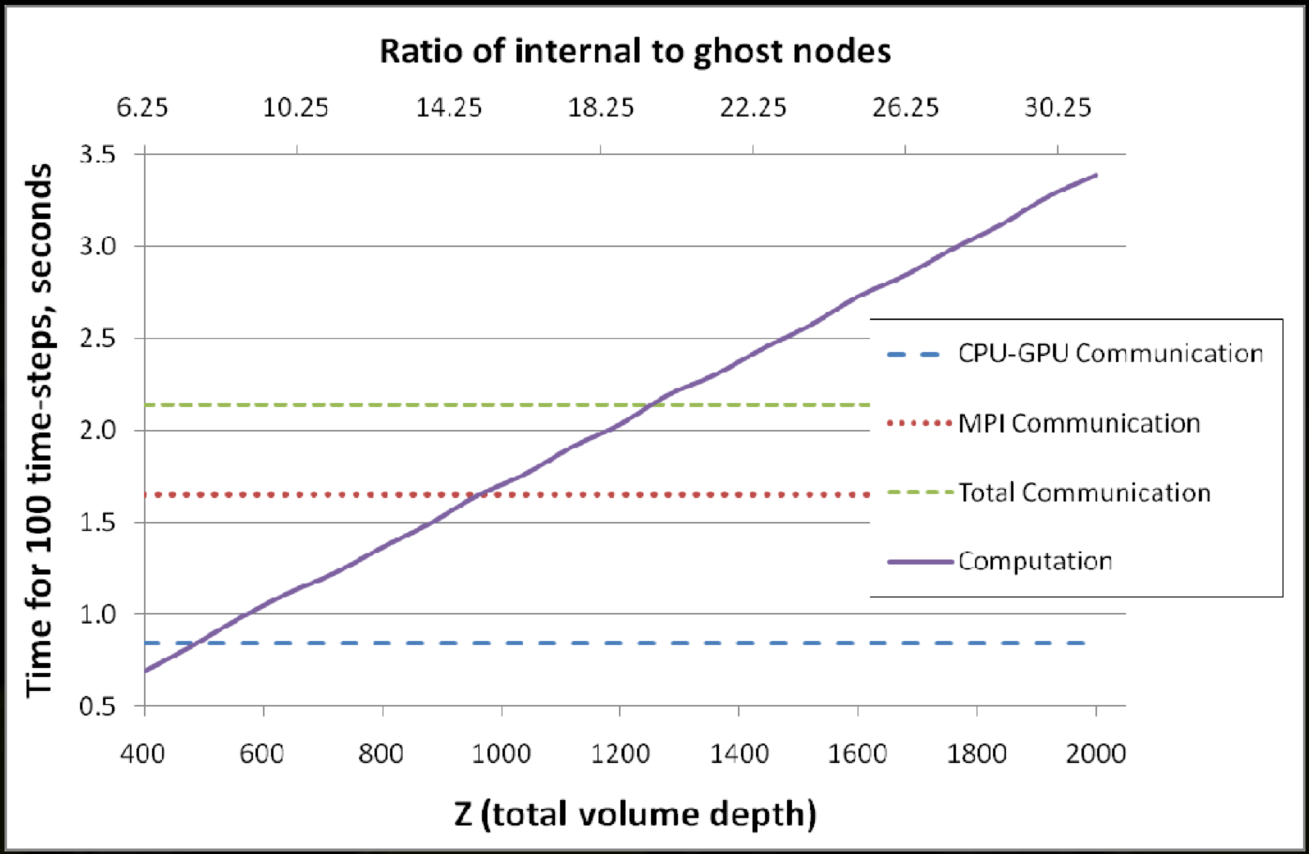
2-GPU Performance





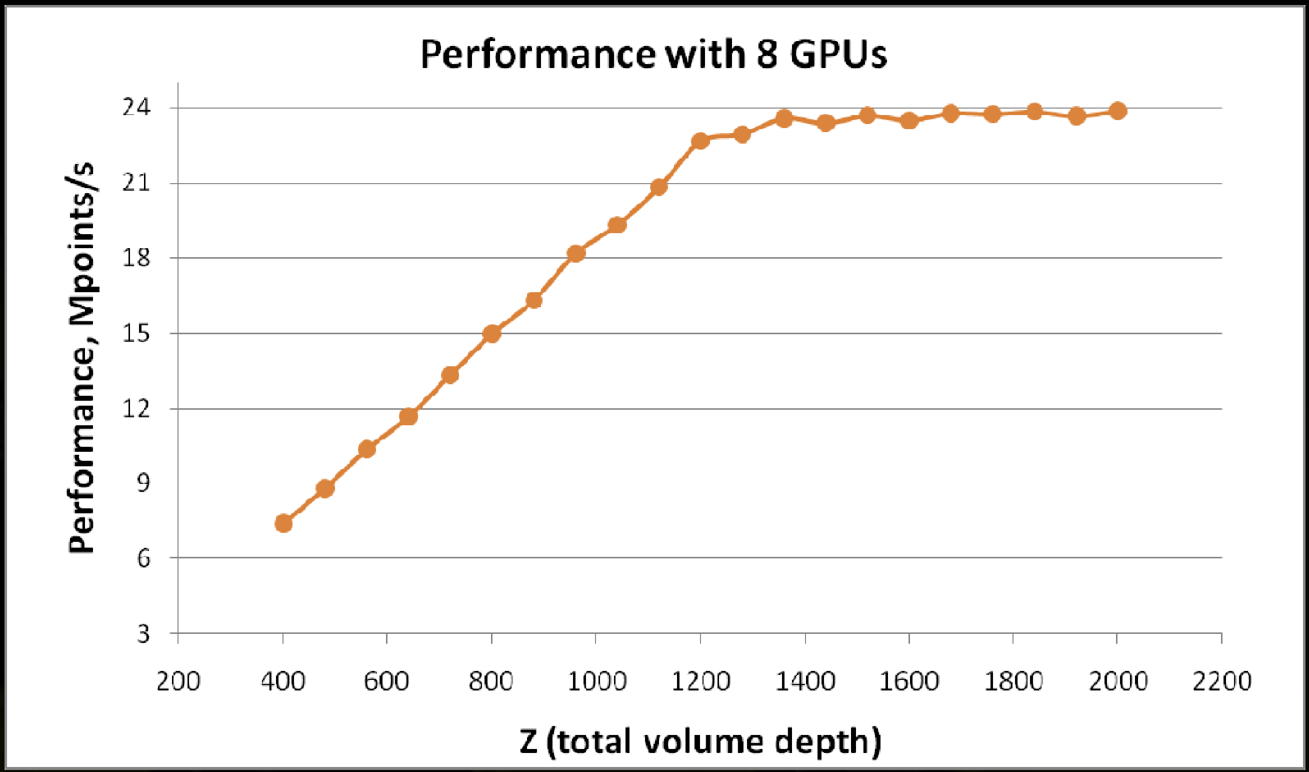
- **Linear scaling is achieved when computation time exceeds communication time**
  - Single GPU performance is ~3.0 Gpoints/s
- **OpenMP case requires no copies on the host side (shared pinned memory)**
  - Communication time includes only PCIe transactions
- **MPI version uses MPI_Sendrecv, which invokes 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

```
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);
...
```