Asynchronous Concurrent Execution in CUDA
Handling Multiple Streams in CUDA
March 10, 2011

“Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.”
– Norman Augustine
Before We Get Started…

- Last time
  - CUDA optimization rules of thumb
  - Discussed two parallel implementations of the prefix scan operation

- Today
  - Asynchronous Concurrent Execution in CUDA
  - Using a CUDA stream and asynchronous mem copy to decouple CPU and GPU execution
  - Handling Multiple Streams in CUDA as a means to enable task parallelism

- Other issues
  - Syllabus firmed up, we'll have three guest lecturers later in the semester
Concurrent Execution between Host and Device

- In order to facilitate concurrent execution between host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task

- Examples of asynchronous calls
  - Kernel launches
  - Device ↔ device memory copies
  - Host ↔ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async

- NOTE: When an application is run via a CUDA debugger or profiler (cuda-gdb, CUDA Visual Profiler, Parallel Nsight), all launches are synchronous
Concurrent Kernel Execution
[CUDA 3.2]

- Feature allows up to 16 kernels to be run on the device at the same time
- When is this useful?
  - Devices of compute capability 2.x are pretty wide (large number of SMs)
  - Sometimes you launch kernels whose execution configuration is smaller than the GPU’s “width”
  - Then, two or three independent kernels can be “squeezed” on the GPU at the same time
- Represents one of GPU’s attempts to look like a MIMD architecture
Host-Device Data Transfer Issues

- In general, host ↔ device data transfers using cudaMemcpy() are blocking
  - Control is returned to the host thread only after the data transfer is complete

- There is a non-blocking variant, cudaMemcpyAsync()

  cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
  kernel<<<grid, block>>>(a_d);
  cpuFunction();

  - The host does not wait on the device to finish the mem copy and the kernel call for it to start execution of cpuFunction() call
  - The launch of “kernel” only happens after the mem copy call finishes

- NOTE: the asynchronous transfer version requires pinned host memory (allocated with cudaHostAlloc()), and it contains an additional argument (a stream ID)
Overlapping Host ↔ Device Data Transfer with Device Execution

- When is this overlapping useful?
  - Imagine a kernel executes on the device and only works with the lower half of the device memory
  - Then, you can copy data from host to device in the upper half of the device memory?
  - These two operations can take place simultaneously

- Note that there is a issue with this idea:
  - The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer

- This issue was addressed by the use of CUDA “streams”
CUDA Streams: Overview

- A programmer can manage concurrency through *streams*

- A stream is a sequence of CUDA commands that execute in order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream

- One host thread can define multiple CUDA streams
  - Think of a stream as a task that gets executed by the GPU. You can have multiple tasks and sometimes the GPU can execute parts of these tasks simultaneously

- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events
CUDA Streams: Overview

[Cntd.]

- With respect to each other, different CUDA streams execute their commands as they see fit
  - Inter-stream relative behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication for kernels allocated to different streams is undefined)
  - Another way to look at it: streams can by synchronized at barrier points, but correlation of sequence execution within different streams is not supported

- When thinking about the typical GPU operations in the stream (see previous slide), remember that the GPU hardware has two types of engines
  - One or more engines for copy operations
  - One engine to execute kernels

- The fact that there are two hardware engines becomes relevant in relation to how you organize the queuing of GPU operations in a stream
  - For improved efficiency you want to have these two engines work simultaneously...
CUDA Streams: Creation

- A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host ↔ device memory copies.

- The following code sample creates two streams and allocates an array “hostPtr” of float in page-locked memory
  - hostPtr will be used in asynchronous host ↔ device memory transfers

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```
CUDA Streams: Making of Use of Them

- In the code below, each of two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```c
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>> (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we’ll revisit shortly…
CUDA Streams: Clean Up Phase

- Streams are released by calling `cudaStreamDestroy()`

```c
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

- Note that `cudaStreamDestroy()` waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread.
CUDA Streams: Caveats

- Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device ↔ device memory copy,
  - Any CUDA command to stream 0 (including kernel launches and host ↔ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations
CUDA Streams: Synchronization Aspects

- `cudaThreadSynchronize()` waits until all preceding commands in all streams have completed.

- `cudaStreamSynchronize()` takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

- `cudaStreamWaitEvent()` takes a stream and an event as parameters and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()` delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to `cudaStreamWaitEvent()` wait on the event.

- `cudaStreamQuery()` provides applications with a way to know if all preceding commands in a stream have completed.

- **NOTE**: To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.
Example 1: Using One GPU Stream

- Example draws on material presented in the “CUDA By Example” book
  - J. Sanders and E. Kandrot, authors

- What is the purpose of this example?
  - Shows an example of using page-locked (pinned) host memory
  - Shows one strategy that you should invoke when dealing with applications that require more memory than you can accommodate on the GPU
  - [Most importantly] Shows a strategy that you can follow to get things done on the GPU without blocking the CPU (host)
    - While the GPU works, the CPU works too

- Remark:
  - In this example the magic happens on the host side. Focus on host code, not on the kernel executed on the GPU (the kernel code is basically irrelevant)
Kernel

- Computes an average, it’s not important, simply something that gets done and allows us later on to gauge efficiency gains when using *multiple* streams (for now dealing with one stream only)

```c
#include "../common/book.h"

#define N (1024*1024)
#define FULL_DATA_SIZE (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```
The “main()” Function

```c
int main( void ) {
    cudaEvent_t start, stop;
    float elapsedTime;
    cudaStream_t stream;
    int *host_a, *host_b, *host_c;
    int *dev_a, *dev_b, *dev_c;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    // initialize the stream
    HANDLE_ERROR( cudaStreamCreate( &stream ) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // allocate host locked memory, used to stream
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );

    for (int i=0; i<FULL_DATA_SIZE; i++) {
        host_a[i] = rand();
        host_b[i] = rand();
    }
}
```
The "main()" Function

```c
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );
}

// copy result chunk from locked to full buffer
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken:  %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
HANDLE_ERROR( cudaStreamDestroy( stream ) );

return 0;
}
```
Example 1, Summary

- Stage 1 sets up the events needed to time the execution of the program
- Stage 2 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device
- Stage 3 enques the set of GPU operations that need to be undertaken (the “chunkification”)
- Stage 4 needed for timing reporting
- Stage 5: clean up time
Example 2: Using Multiple Streams

[Version 1]

- Implement the same example but use two streams to this end

- Why would you want to use multiple streams?
  - For devices that are capable of overlapping execution with host↔device data movement, you might hide this data movement and improve overall performance

- Two ideas underlie the process
  - The idea of “chunkification” of the computation
    - Large computation is broken into pieces that are queued up for execution on the device (we already saw this in Example 1, which uses one stream)
  - The idea of overlapping execution with host↔device data movement
Overlapping Execution and Data Transfer: The Ideal Scenario

Observations:
- “memcpy” actually represents an asynchronous cudaMemcpyAsync() memory copy call
- White (empty) boxes represent time when one stream is waiting to execute an operation that it cannot overlap with the other stream’s operation
- The goal: keep both GPU engine types (execution and mem copy) busy
  - Note: recent hardware allows two copies to take place simultaneously: one from host to device, at the same time one goes on from device to host (you have two copy engines)
Kernel

- Note that the kernel actually doesn’t change...

```c
#include "../common/book.h"

#define N   (1024*1024)
#define FULL_DATA_SIZE   (N*20)

__global__
void
kernel(
    int* a,
    int* b,
    int* c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float   as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float   bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```
The “main()” Function, 2 Streams

```c
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no speed up from streams\n" );
        return 0;
    }

    cudaEvent_t start, stop;
    float elapsedTime;
    cudaStream_t stream0, stream1;
    int *host_a, *host_b, *host_c;
    int *dev_a0, *dev_b0, *dev_c0;
    int *dev_a1, *dev_b1, *dev_c1;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a1, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b1, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c1, N * sizeof(int) ) );

    // allocate host locked memory, used to stream
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
}
```
The “main()” Function, 2 Streams
[Cntd.]

```c
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

HANDLE_ERROR( cudaEventRecord( start, 0 ) );

// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );

    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

    // copy the data from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );

    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

    // copy the data from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```

Still Stage 3

Stage 4
The “main()” Function, 2 Streams

[Cntd.]

```c
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );
return 0;
}
```
Example 2 [Version 1], Summary

- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer (ok in devices of compute capability 1.1 and higher)

- Stage 2 sets up the events needed to time the execution of the program

- Stage 3 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device and initializes data

- Stage 4 enques the set of GPU operations that need to be undertaken (the “chunkification”)

- Stage 5 takes care of timing reporting and clean up
Comments, Using 2 Streams
[Version 1]

- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,“
  - Sanders and Kandrot reported results on NVIDIA GTX285

- Using one stream (in Example 1): 62 ms

- Using two streams (this example, version 1): 61 ms

- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled
The Two Stream Example, Version 1
Looking Under the Hood

At the left:
- An illustration of how the work queued up in the streams ends up being assigned by the CUDA driver to the two GPU engines (copy and execution)

At the right:
- Image shows dependency that is implicitly set up in the two streams given the way the streams were defined in the code
- The queue in the Copy Engine, combined with the dependencies defined determines the scheduling of the Copy and Kernel Engines (see next slide)

Mapping of CUDA streams onto GPU engines

Arrows depicting the dependency of cudaMemcpyAsync () calls on kernel executions in the 2 Streams example
Note that due to the *specific* way in which the streams were defined (depth first), basically there is no overlap of copy & execution…

- This explains the no net-gain in efficiency compared to the one stream example

Remedy: go breadth first, instead of depth first
- In the current version, execution on the two engines was inadvertently blocked by the way the streams have been organized and the existing scheduling and lack of dependency checks available in the current version of CUDA
The 2 Stream Example
[Version 2: A More Effective Implementation]

- Old way (the depth first approach):
  - Assign the copy of \(a\), copy of \(b\), kernel execution, and copy of \(c\) to stream0. Subsequently, do the same for stream1.

- New way (the breadth first approach):
  - Add the copy of \(a\) to stream0, and then add the copy of \(a\) to stream1.
  - Next, add the copy of \(b\) to stream0, and then add the copy of \(b\) to stream1.
  - Next, enqueue the kernel invocation in stream0, and then enqueue one in stream1.
  - Finally, enqueue the copy of \(c\) back to the host in stream0 followed by the copy of \(c\) in stream1.
The 2 Stream Example
A 20% More Effective Implementation (48 vs. 61 ms)

```c
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // enqueue copies of a in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

    // enqueue copies of b in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

    // enqueue kernels in stream0 and stream1
    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

    // enqueue copies of c from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```

Execution timeline of the breadth-first approach (blue line shows dependency)
Using Streams, Lessons Learned

- Streams provide a basic mechanism that enables task-level parallelism in CUDA C applications

- Two requirements underpin the use of streams in CUDA C
  - `cudaHostAlloc()` should be used to allocate memory on the host so that it can be used in conjunction with a `cudaMemcpyAsync()` non-blocking copy command
    - The use of pinned (page-locked) host memory improves data transfer performance even if you only work with one stream
  - Effective latency hiding of kernel execution with memory copy operations requires a breadth-first approach to enqueuing operations in different streams
    - This is a consequence of the two engine setup associated with a GPU
CUDA 4.0
Application Porting Made Simpler

- Rapid Application Porting
  *Unified Virtual Addressing*

- Faster Multi-GPU Programming
  *NVIDIA GPUDirect™ 2.0*

- Easier Parallel Programming in C++
  *Thrust*
CUDA 4.0: Highlights

Easier Parallel Application Porting
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

Faster Multi-GPU Programming
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

New & Improved Developer Tools
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
No-copy Pinning of System Memory

Reduce system memory usage and CPU memcpy() overhead
Easier to add CUDA acceleration to existing applications
Just register malloc’d system memory for async operations and then call cudaMemcpy() as usual

<table>
<thead>
<tr>
<th>Before No-copy Pinning</th>
<th>With No-copy Pinning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra allocation and extra copy required</td>
<td>Just register and go!</td>
</tr>
<tr>
<td>malloc(a)</td>
<td></td>
</tr>
<tr>
<td>cudaMallocHost(b)</td>
<td></td>
</tr>
<tr>
<td>memcpy(b, a)</td>
<td>cudaMemcpy(a)</td>
</tr>
<tr>
<td>cudaMemcpy() to GPU, launch kernels, cudaMemcpy() from GPU</td>
<td></td>
</tr>
<tr>
<td>memcpy(a, b)</td>
<td></td>
</tr>
<tr>
<td>cudaFreeHost(b)</td>
<td>cortaHostUnregister(a)</td>
</tr>
</tbody>
</table>

All CUDA-capable GPUs on Linux or Windows
Requires Linux kernel 2.6.15+ (RHEL 5)
C++ Templatized Algorithms & Data Structures (Thrust)

- Powerful open source C++ parallel algorithms & data structures
- Similar to C++ Standard Template Library (STL)
- Automatically chooses the fastest code path at compile time
- Divides work between GPUs and multi-core CPUs
- Parallel sorting @ 5x to 100x faster

### Data Structures
- thrust::device_vector
- thrust::host_vector
- thrust::device_ptr
- Etc.

### Algorithms
- thrust::sort
- thrust::reduce
- thrust::exclusive_scan
- Etc.

© NVIDIA Corporation 2011
CUDA 4.0: Highlights

Easier Parallel Application Porting
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

Faster Multi-GPU Programming
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

New & Improved Developer Tools
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
Before NVIDIA GPUDirect™ v2.0

**Required Copy into Main Memory**

Two copies required:
1. cudaMemcpy(GPU2, sysmem)
2. cudaMemcpy(sysmem, GPU1)
NVIDIA GPUDirect™ v2.0: Peer-to-Peer Communication

Direct Transfers between GPUs

Only one copy required:
1. cudaMemcpy(GPU2, GPU1)
Unified Virtual Addressing
Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces

- System Memory
  - 0x0000
  - 0xFFFF

- GPU0 Memory
  - 0x0000
  - 0xFFFF

- GPU1 Memory
  - 0x0000
  - 0xFFFF

- CPU
  - 0x0
  - 0xFFFF

- GPU0
  - 0x0
  - 0xFFFF

- GPU1
  - 0x0
  - 0xFFFF

UVA: Single Address Space

- System Memory
  - 0x0
  - 0xFFFF

- GPU0 Memory
  - 0x0
  - 0xFFFF

- GPU1 Memory
  - 0x0
  - 0xFFFF

- CPU
  - 0x0
  - 0xFFFF

- GPU0
  - 0x0
  - 0xFFFF

- GPU1
  - 0x0
  - 0xFFFF

© NVIDIA Corporation 2011
Unified Virtual Addressing

One address space for all CPU and GPU memory
- Determine physical memory location from pointer value
- Enables libraries to simplify their interfaces (e.g. cudaMemcpy)

<table>
<thead>
<tr>
<th>Before UVA</th>
<th>With UVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate options for each permutation</td>
<td>One function handles all cases</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td>cudaMemcpyDefault</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td>(data location becomes an implementation detail)</td>
</tr>
</tbody>
</table>

Supported on Tesla 20-series and other Fermi GPUs
- 64-bit applications on Linux and Windows TCC
CUDA 4.0: Highlights

Easier Parallel Application Porting
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

Faster Multi-GPU Programming
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

New & Improved Developer Tools
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
## CUDA Features Overview

<table>
<thead>
<tr>
<th>New in CUDA 4.0</th>
<th>Platform</th>
<th>Programming Model</th>
<th>Parallel Libraries</th>
<th>Development Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUDirect&lt;sup&gt;TM&lt;/sup&gt; (v 2.0) Peer-Peer Communication</td>
<td>Hardware Features</td>
<td>Hardware Features</td>
<td>NVIDIA Library Support</td>
<td>NVIDIA Developer Tools</td>
</tr>
<tr>
<td>ECC Memory</td>
<td>C support</td>
<td>Unified Virtual Addressing</td>
<td>NVIDIA Library Support</td>
<td>Parallel Nsight Pro 1.5</td>
</tr>
<tr>
<td>Double Precision</td>
<td>C++ new/delete</td>
<td>C++ Virtual Functions</td>
<td>Complete math.h</td>
<td>NVIDIA Developer Tools</td>
</tr>
<tr>
<td>Native 64-bit Architecture</td>
<td>Function Pointers</td>
<td>Thrust C++ Library</td>
<td>Complete BLAS Library</td>
<td>Parallel Nsight</td>
</tr>
<tr>
<td>Concurrent Kernel Execution</td>
<td>Recursion</td>
<td>Tempered Performance</td>
<td>Sparse Matrix Math Library</td>
<td>for MS Visual Studio</td>
</tr>
<tr>
<td>Dual Copy Engines</td>
<td>Atomics</td>
<td>Primitives Library</td>
<td>RNG Library</td>
<td>cuda-gdb Debugger</td>
</tr>
<tr>
<td>6GB per GPU supported</td>
<td>malloc/free</td>
<td>CUDA Fortran (PGI)</td>
<td>FFT Library (1D, 2D and 3D)</td>
<td>with multi-GPU support</td>
</tr>
<tr>
<td>Operating System Support</td>
<td>C++ support</td>
<td>CUDA Fortran (PGI)</td>
<td>Video Decoding Library (NVCUVID)</td>
<td>CUDA/OpenCL Visual Profiler</td>
</tr>
<tr>
<td>MS Windows 32/64</td>
<td>Classes/Objects</td>
<td>NVIDIA Library Support</td>
<td>Video Encoding Library (NVCUENC)</td>
<td>CUDA Memory Checker</td>
</tr>
<tr>
<td>Linux 32/64</td>
<td>Class Inheritance</td>
<td>Complete BLAS Library</td>
<td>Image Processing Library (NPP)</td>
<td>CUDA Disassembler</td>
</tr>
<tr>
<td>Mac OS X 32/64</td>
<td>Polymorphism</td>
<td>Sparse Matrix Math Library</td>
<td>Video Processing Library (NPP)</td>
<td>GPU Computing SDK</td>
</tr>
<tr>
<td>Designed for HPC</td>
<td>Operator Overloading</td>
<td>RNG Library</td>
<td>CUDA Fortran (PGI)</td>
<td>NVML</td>
</tr>
<tr>
<td>Cluster Management</td>
<td>Class Templates</td>
<td>FFT Library (1D, 2D and 3D)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUPTI</td>
</tr>
<tr>
<td>GPUDirect</td>
<td>Function Templates</td>
<td>Video Decoding Library (NVCUVID)</td>
<td>CUDA Fortran (PGI)</td>
<td>3rd Party Developer Tools</td>
</tr>
<tr>
<td>Tesla Compute Cluster (TCC)</td>
<td>Virtual Base Classes</td>
<td>Video Encoding Library (NVCUENC)</td>
<td>CUDA Fortran (PGI)</td>
<td>Allinea DDT</td>
</tr>
<tr>
<td>Multi-GPU support</td>
<td>Namespaces</td>
<td>Image Processing Library (NPP)</td>
<td>CUDA Fortran (PGI)</td>
<td>RogueWave / Totalview</td>
</tr>
<tr>
<td>Fortran support</td>
<td>CUDA Fortran (PGI)</td>
<td>Video Processing Library (NPP)</td>
<td>CUDA Fortran (PGI)</td>
<td>Vampir</td>
</tr>
<tr>
<td>CUDA Fortran (PGI)</td>
<td>NVIDIA Library Support</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td>Tau</td>
</tr>
<tr>
<td>NVIDIA Library Support</td>
<td>Complete BLAS Library</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td>CAPS HMPP</td>
</tr>
<tr>
<td>CUDA C Compiler</td>
<td>Sparse Matrix Math Library</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td>© NVIDIA Corporation 2011</td>
</tr>
<tr>
<td>CUDA C Parallel Extensions</td>
<td>RNG Library</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Function Pointers</td>
<td>FFT Library (1D, 2D and 3D)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>Video Decoding Library (NVCUVID)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Atomics</td>
<td>Video Encoding Library (NVCUENC)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>malloc/free</td>
<td>Image Processing Library (NPP)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>NVIDIA C Compiler</td>
<td>Video Processing Library (NPP)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>CUDA C Parallel Extensions</td>
<td>NVIDIA Fortran (PGI)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Function Pointers</td>
<td>NVIDIA Fortran (PGI)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>NVIDIA Fortran (PGI)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>Atomics</td>
<td>NVIDIA Fortran (PGI)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
<tr>
<td>malloc/free</td>
<td>NVIDIA Fortran (PGI)</td>
<td>NVIDIA Fortran (PGI)</td>
<td>CUDA Fortran (PGI)</td>
<td></td>
</tr>
</tbody>
</table>
CUDA Developer Resources from NVIDIA

Development Tools

CUDA Toolkit
Complete GPU computing development kit
cuda-gdb
GPU hardware debugging
cuda-memcheck
Identifies memory errors
cuobjdump
CUDA binary disassembler
Visual Profiler
GPU hardware profiler for CUDA C and OpenCL
Parallel Nsight Pro
Integrated development environment for Visual Studio

SDKs and Code Samples

GPU Computing SDK
CUDA C/C++, DirectCompute, OpenCL code samples and documentation
Books
CUDA by Example
GPU Computing Gems Programming Massively Parallel Processors
Many more...
Optimization Guides
Best Practices for GPU computing and graphics development

Libraries and Engines

Math Libraries
CUFFT, CUBLAS, CUSPARSE, CURAND, math.h
3rd Party Libraries
CULA LAPACK, VSIPL
NPP Image Libraries
Performance primitives for imaging
App Acceleration Engines
Ray Tracing: Optix, iRay
Video Encoding / Decoding
NVCUVENC / VCUVID

http://developer.nvidia.com

© NVIDIA Corporation 2011
PGI CUDA x86 Compiler

Benefits
- Deploy CUDA apps on legacy systems without GPUs
- Less code maintenance for developers

Timeline
- April/May 1.0 initial release
  Develop, debug, test functionality
- Aug 1.1 performance release
  Multicore, SSE/AVX support
CUDA 3rd Party Ecosystem

Cluster Tools

Cluster Management
- Platform HPC
- Platform Symphony
- Bright Cluster manager
- Ganglia Monitoring System
- Moab Cluster Suite
- Altair PBS Pro

Job Scheduling
- Altair PBSpro
- TORQUE
- Platform LSF

MPI Libraries
- Coming soon...

Parallel Language
Solutions & APIs

- PGI CUDA Fortran
- PGI Accelerator (C/Fortran)
- PGI CUDA x86
- CAPS HMPP
- pyCUDA (Python)
- Tideword GPU.NET (C#)
- JCuda (Java)
- Khronos OpenCL
- Microsoft DirectCompute

3rd Party Math Libraries

- CULA Tools (EM Photonics)
- MAGMA Heterogeneous LAPACK
- IMSL (Rogue Wave)
- VSipl (GPU VSipl)
- NAG

Parallel Tools

Parallel Debuggers

- MS Visual Studio with Parallel Nsight Pro
- Allinea DDT Debugger
- TotalView Debugger

Parallel Performance Tools

- ParaTools VampirTrace
- TauCUDA Performance Tools
- PAPI
- HPC Toolkit

Compute Platform Providers

Cloud Providers

- Amazon EC2
- Peer 1

OEM’s

- Dell
- HP
- IBM

Infiniband Providers

- Mellanox
- QLogic