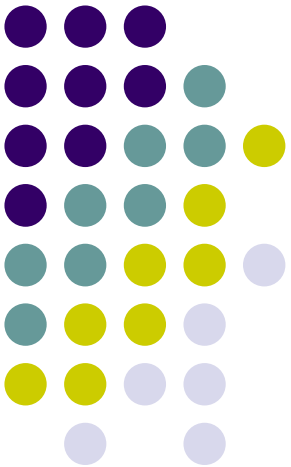
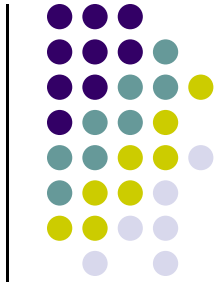
ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Concurrency through CUDA Streams
CUDA Unified Memory

October 28, 2015



Quote of the Day

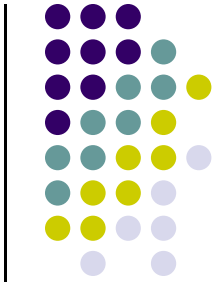


“Innovation distinguishes between a leader and a follower.”

-- Steve Jobs, innovator

[1955 - 2011]

Before We Get Started

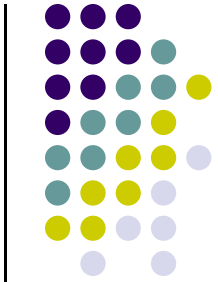


- Issues covered last time:
  - Case study: parallel reduction in CUDA
  - CUDA Streams

- Today's topics
  - A word from Colin (the lab student looking after Euler)
  - CUDA streams [wrap up]
  - CUDA Unified Memory

- Assignment:
  - HW06 – due **Th**, Oct 29 at 11:59 PM
  - HW07 - posted tonight, due on Wd, Nov. 4 at 11:59 PM

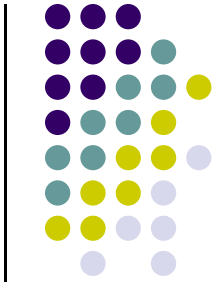
Cluster SLURM Jobs



- The time limit for ME759 jobs is 20 minutes.
  - Your homework should not require longer than this to run.
  - Your homework will wait in queue indefinitely if you attempt to request more time than this.

- ME759 students are not allowed to use interactive jobs.
  - There aren't enough cluster resources for each student to have an interactive job running (some of you have attempted to use several at a time).
  - SLURM won't prevent you from running these jobs, but the sysadmin will disconnect you if you use them while other jobs are waiting in the queue.
  - With the exception of running your code, the head node is capable of completing any task you might need to complete your homework.

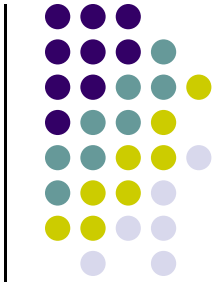
Reminders About Using CUDA



- GPU code will not run if you do not request a GPU for your job.
- In order to run ./yourbinary, your job needs to run in the correct directory. SLURM can do this for you if your executable is in the same folder as your submit script.
- Manually adding the cuda binaries to your PATH does not guarantee that your code will run.
- Instead, the following command will do this for you:
- ~$ module load cuda

```
#!/bin/bash
#SBATCH -N 1 -n 1
#SBATCH -p slurm_me759
#SBATCH -t 0-0:19:59
#SBATCH --gres=gpu:1
#SBATCH -o slurmjob.oe%j
cd $SLURM_SUBMIT_DIR
module load cuda
...
```

Euler – A Shared HW Asset

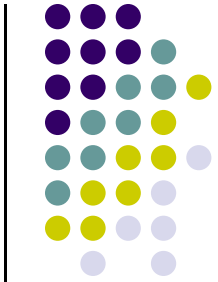


- If your jobs keep getting cancelled, it might mean that something is wrong.
  - Instead of trying to submit the same thing over and over again, email your sysadmin about the problem. Better yet, he might have emailed you about it already.
  - (colin.vandenheuvel@wisc.edu)

- Colin is perfectly happy to help with cluster problems
  - He will do so as quickly as he can.
  - Waiting until late on the night that your homework is due to send him panicked email about not being able to run your job is not the best approach

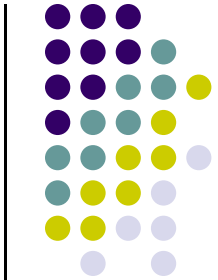
Example 1: Using One Stream

[Enable both CPU and GPU to mind their business at the same time]



- 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) – goes back to the use of `cudaMemcpyAsync`
    - 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)

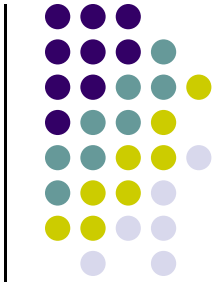
This Example's Kernel



- Computes some 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)
  - Inputs: a and b
  - Output: c

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

#define N   1048576  // this is 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



```
01| int main( void ) {
02|     cudaEvent_t     start, stop;
03|     float           elapsedTime;
04|
05|     cudaStream_t    stream;
06|     int *host_a, *host_b, *host_c;
07|     int *dev_a, *dev_b, *dev_c;
08|
09|     // start the timers
10|     cudaEventCreate( &start );
11|     cudaEventCreate( &stop  );
12|
13|     // initialize the stream; only one stream for now…
14|     cudaStreamCreate( &stream ) );
15|
16|     // allocate the memory on the GPU
17|     cudaMalloc( (void**)&dev_a, N * sizeof(int) );
18|     cudaMalloc( (void**)&dev_b, N * sizeof(int) );
19|     cudaMalloc( (void**)&dev_c, N * sizeof(int) );
20|
```

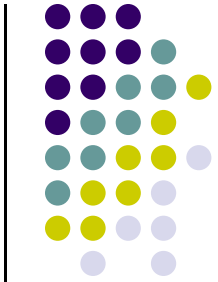
Stage 1

```
21|     // allocate host pinned memory, used to stream
22|     cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
23|     cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
24|     cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
25|
26|     for (int i=0; i<FULL_DATA_SIZE; i++) {
27|         host_a[i] = rand();
28|         host_b[i] = rand();
29|     }
```

Stage 2

The “main()” Function

[Cntd.]



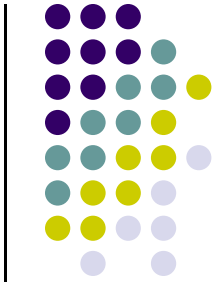
```
30|
31|    cudaEventRecord( start, 0 );
32|    // now loop over full data, in bite-sized chunks
33|    for (int i=0; i<FULL_DATA_SIZE; i+= N) {
34|        // copy the locked memory to the device, async
35|        cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );
36|        cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream );
37|
38|        kernel<<<(N+255)/256,256,0,stream>>>( dev_a, dev_b, dev_c );
39|
40|        // copy the data from device to locked memory
41|        cudaMemcpyAsync( host_c+i, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost, stream );
42|
43|    }
44|
45|    cudaStreamSynchronize( stream );
46|
47|    cudaEventRecord( stop, 0 );
48|
49|    cudaEventSynchronize( stop );
50|    cudaEventElapsedTime( &elapsedTime, start, stop ) );
51|    printf( "Time taken:  %3.1f ms\n", elapsedTime );
52|
53|    // cleanup the streams and memory
54|    cudaFreeHost( host_a );
55|    cudaFreeHost( host_b );
56|    cudaFreeHost( host_c );
57|    cudaFree( dev_a );
58|    cudaFree( dev_b );
59|    cudaFree( dev_c );
60|    cudaStreamDestroy( stream );
61|
62|    return 0;
63|}
```

Stage 3

Stage 4

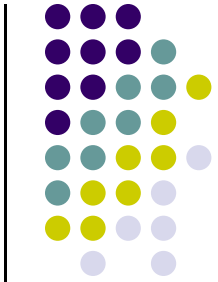
Stage 5

Example 1, Summary



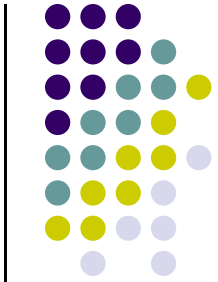
- Stage 1 sets up the events needed to time the execution of the program
- Stage 2 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device
- Stage 3 enques the set of GPU operations that need to be undertaken (the “chunkification”)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

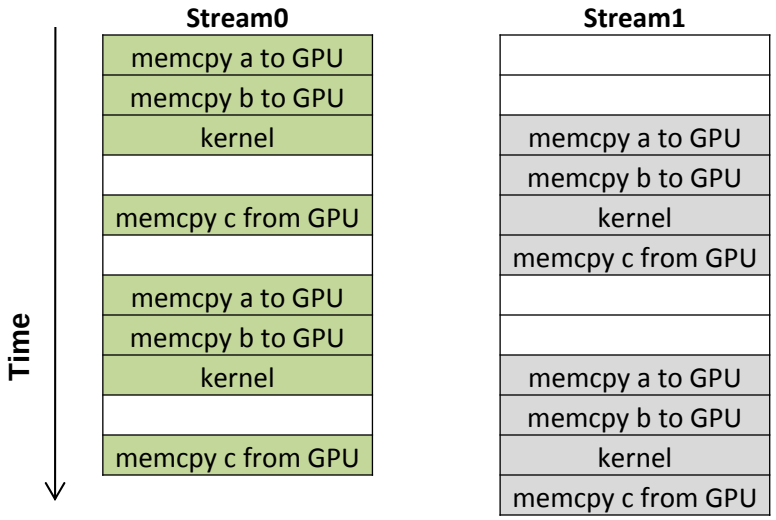
Example 2: Using Multiple Streams
[Version 2.1]



- Implement the same example but use two streams to this end
- Why would you want to use multiple streams?
  - Overlapping GPU execution with host ↔ device data movement can improve overall performance
- Two ideas underlie the process
  - The idea of "chunkification" of the computation
    - 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 PCI host ↔ device data movement
- NOTE: "chunkification" similar to "tiling". However, "tiling" is something that happens exclusively on the device (from global to shared memory). Here, the "chunkification" happens on the host

Overlapping Execution and Data Transfer: A Desirable Scenario

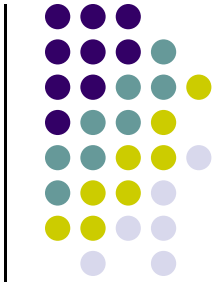




Timeline of intended application execution using two independent streams

- 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 subengines)

The "main()" Function, Two Streams



```
01|  int main( void ) {
02|  cudaDeviceProp  prop;
03|  int whichDevice;
04|  HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
05|  HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
06|  if (!prop.deviceOverlap) {
07|      printf( "Device will not handle overlaps, so no speed up from streams\n" );
08|      return 0;
09|  }
```

Stage 1

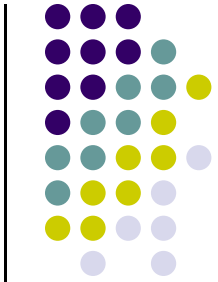
```
11|  cudaEvent_t     start, stop;
12|  float           elapsedTime;
13|
14|  cudaStream_t    stream0, stream1;
15|  int *host_a, *host_b, *host_c;
16|  int *dev_a0, *dev_b0, *dev_c0;
17|  int *dev_a1, *dev_b1, *dev_c1;
18|
19|  // start the timers
20|  cudaEventCreate( &start );
21|  cudaEventCreate( &stop  );
22|
23|  // initialize the streams
24|  cudaStreamCreate( &stream0 );
25|  cudaStreamCreate( &stream1 );
26|
```

Stage 2

Stage 3

```
27|  // allocate the memory on the GPU
28|  cudaMalloc( (void**)&dev_a0, N * sizeof(int) );
29|  cudaMalloc( (void**)&dev_b0, N * sizeof(int) );
30|  cudaMalloc( (void**)&dev_c0, N * sizeof(int) );
31|  cudaMalloc( (void**)&dev_a1, N * sizeof(int) );
32|  cudaMalloc( (void**)&dev_b1, N * sizeof(int) );
33|  cudaMalloc( (void**)&dev_c1, N * sizeof(int) );
34|
35|  // allocate host locked memory, used to stream
36|  cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
37|  cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
38|  cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault );
```

The "main()" Function, Two Streams
[Cntd.]



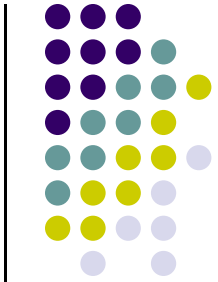
Still Stage 3

```
39|    for (int i=0; i<FULL_DATA_SIZE; i++) {
40|        host_a[i] = rand();
41|        host_b[i] = rand();
42|    }
43|
44|    cudaEventRecord( start, 0 );
```

Stage 4

```
45|    // now loop over full data, in bite-sized chunks
46|    for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
47|        // copy data from pinned memory to the device, async
48|        cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
49|        cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
50|
51|        kernel<<<(N+255)/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
52|
53|        // copy the data from device to locked memory
54|        cudaMemcpyAsync( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 );
55|
56|
57|        // copy the locked memory to the device, async
58|        cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
59|        cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
60|
61|        kernel<<<(N+255)/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
62|
63|        // copy the data from device to locked memory
64|        cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 );
65|}
66|
```

The "main()" Function, Two Streams
[Cntd.]

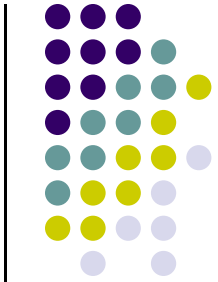


```
67|    cudaStreamSynchronize( stream0 );
68|    cudaStreamSynchronize( stream1 );
69|
70|    cudaEventRecord( stop, 0 );
71|
72|    cudaEventSynchronize( stop );
73|    cudaEventElapsedTime( &elapsedTime, start, stop ) );
74|    printf( "Time taken:  %3.1f ms\n", elapsedTime );
75|
76|    // cleanup the streams and memory
77|    cudaFreeHost( host_a );
78|    cudaFreeHost( host_b );
79|    cudaFreeHost( host_c );
80|    cudaFree( dev_a0 );
81|    cudaFree( dev_b0 );
82|    cudaFree( dev_c0 );
83|    cudaFree( dev_a1 );
84|    cudaFree( dev_b1 );
85|    cudaFree( dev_c1 );
86|    cudaStreamDestroy( stream0 );
87|    cudaStreamDestroy( stream1 );
88|
89|    return 0;
90|}
```

Stage 5

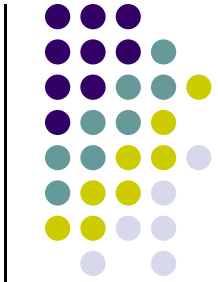
NOTE: the kernel doesn't actually change…

Example 2.1 [Version 1], Summary



- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer
- 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

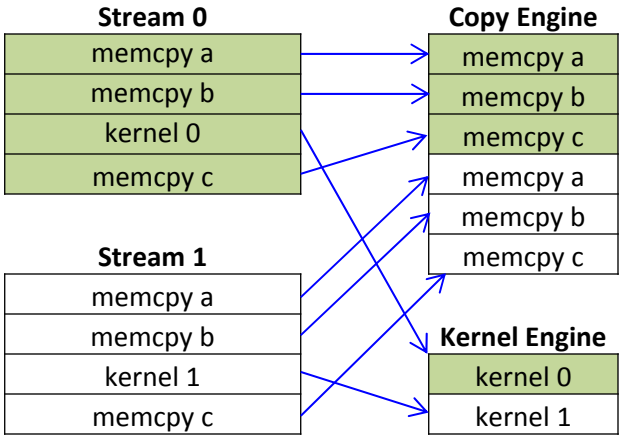
Comments, Using Two Streams
[Version 2.1]



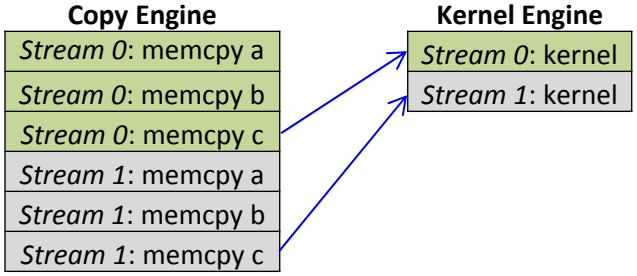
- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,”
  - Sanders and Kandrot reported results on NVIDIA GTX285
- Using one stream (in Example 1): 62 ms
- Using two streams (this example, version 1): 61 ms
- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled

The Two Stream Example, Version 2.1

Looking Under the Hood



Mapping of CUDA streams onto GPU engines

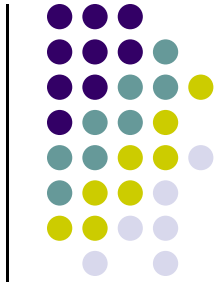


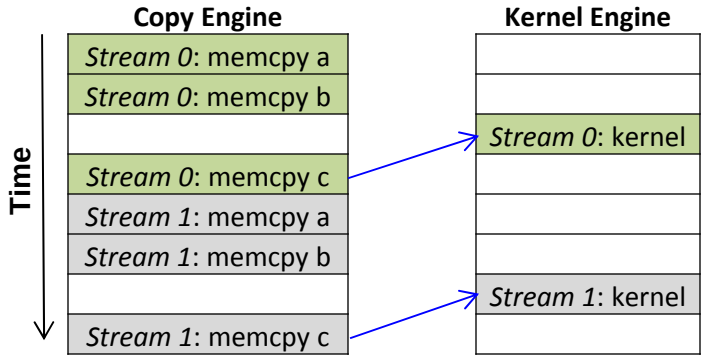
Arrows depicting the dependency of cudaMemcpyAsync () calls on kernel executions in the 2 Streams example

- 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)
  - Important remark: FIFO is also observed in relation to scheduling the engines (not only the streams)
- 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 implied stream dependencies determines the scheduling of the Copy and Kernel Engines (see next slide)

The Two Stream Example

Looking Under the Hood



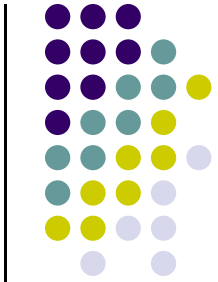


Execution timeline of the 2 Stream example (blue line shows dependency; empty boxes represent idle segments)

- Note that due to the *specific* way in which the streams were defined (depth first), basically there is no overlap of copy & execution…
  - 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 set up and the existing scheduling and lack of dependency checks available in the current version of CUDA

The Two Stream Example

[Version 2.2: A More Effective Implementation: Breadth First]

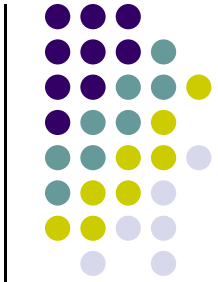


- Old way (the depth first approach):
  - Assign the copy of a, copy of b, kernel execution, and copy of c to stream0. Subsequently, do the same for stream1

- 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, 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.

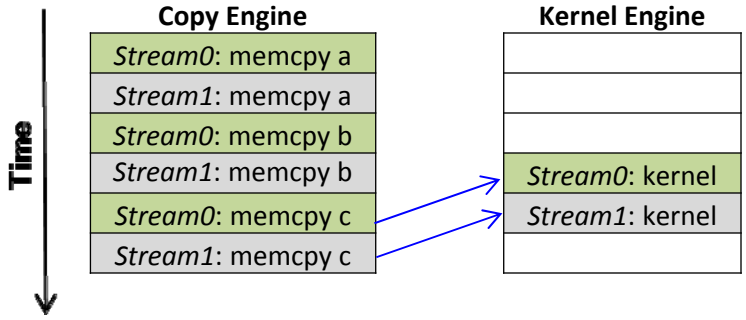
The Two Stream Example

A 20% More Effective Implementation (48 vs. 61 ms)



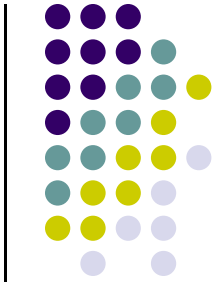
```
A|     // loop over full data, in bite-sized chunks
B|    for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
C|       // enqueue copies of a in stream0 and stream1
D|        cudaMemcpyAsync( dev_a0, host_a+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
E|        cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
F|        // enqueue copies of b in stream0 and stream1
G|        cudaMemcpyAsync( dev_b0, host_b+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 );
H|        cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 );
I|
J|        // enqueue kernels in stream0 and stream1
K|        kernel<<<(N+255)/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
L|        kernel<<<(N+255)/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
M|
N|        // enqueue copies of c from device to locked memory
O|        cudaMemcpyAsync( host_c+i, dev_c0,   N * sizeof(int), cudaMemcpyDeviceToHost, stream0 );
P|        cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 );
Q|    }
```

Replaces Previous Stage 4



Execution timeline of the breadth-first approach
(blue line shows dependency)

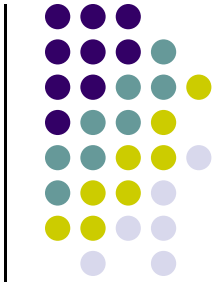
Using Streams, Lessons Learned



- 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

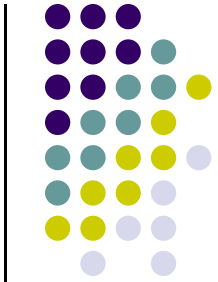
CUDA Streams: More recent developments



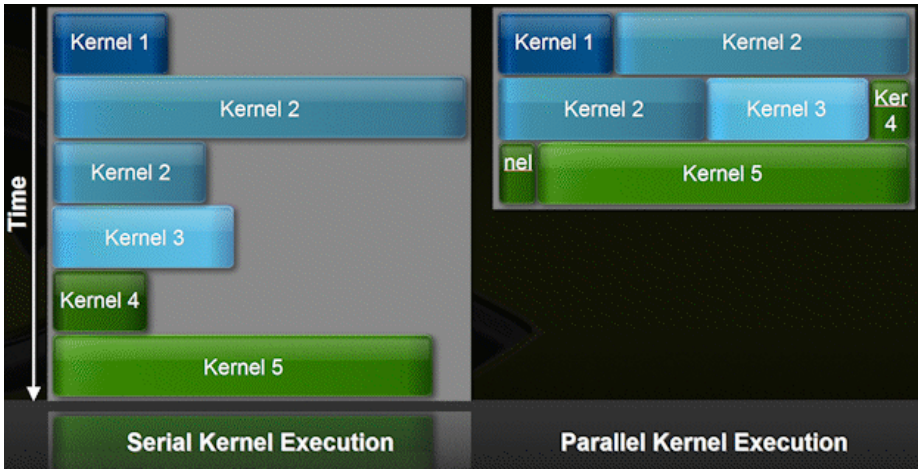
- Since CUDA 5.0
  - Stream Callbacks, will call host function when stream has finished all work
  - **`cudaStreamAddCallback ( cudaStream_t stream, cudaStreamCallback_t callback, void* userData, unsigned int  flags )`**
- Since CUDA 5.5
  - You can give streams priority
    - **`cudaStreamCreateWithPriority`**
    - Use **`cudaDeviceGetStreamPriorityRange`** to get the available priorities
- Since CUDA 7:
  - nvcc –default-stream per-thread
  - Each host thread will get its own stream
  - Each stream becomes a regular non-blocking stream

Concurrent Kernel Execution

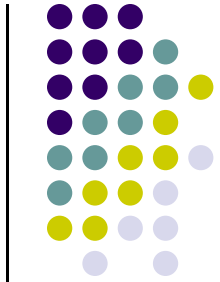
[another type of concurrency through use of CUDA streams]



- Fermi: up to 16 kernels can be run on the device at the same time
- When is this useful?
  - Devices of compute capability 2.x and above are wide (large number of SMs)
  - A kernel might be launched with an execution config. that doesn't fully utilize entire GPU
  - Main idea: two or three independent kernels can be "squeezed" on GPU at the same time
- GPU looks like a MIMD architecture
  - Requires use of multiple streams



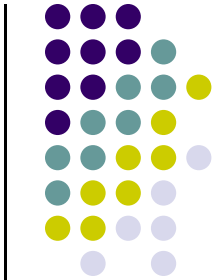
[NVIDIA]→



On Data Access and Transfer in CUDA

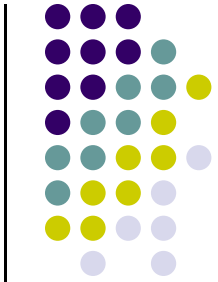
[further issues]

Summary / Objective



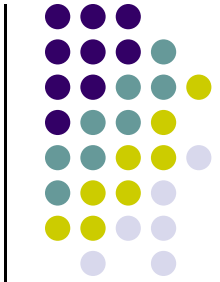
- Premise:
  - Managing and optimizing host-device data transfers is tedious
- Key point:
  - Unified Memory (UM) support in CUDA 6+ simplifies the programmer's job
- This segment's two goals:
  - Review history of CUDA host/device memory management
  - Understand how UM makes host/device memory management easier and more efficient

cudaMemcpy



- A staple of CUDA, available as early as release 1.0
- Setup was simple: one CPU thread dealt with one GPU
  - The drill :
    - Data transferred from host memory into device memory with `cudaMemcpy`
    - Data was processed on the device by invoking a kernel
    - Results transferred from device memory into host memory with `cudaMemcpy`
- Memory allocated on the host with `malloc`
- Memory on device allocated w/ CUDA runtime function `cudaMalloc`
- The bottleneck: data movement over the PCI-E bus

The PCI-E Bus, Putting Things in Perspective

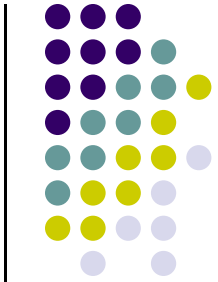


- PCI-E approximate bandwidth, per direction:
  - V1: 3 GB/s
  - V2: 6 GB/s
  - V3 (today): 12 GB/s

- Bandwidths quoted above pretty small compared to
  - Host memory: 25 – 50 GB/s per socket
  - GPU global mem bandwidth 100 – 200 GB/s

Review: cudaHostAlloc

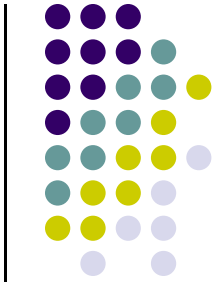
[some bad parts, some good parts]



- What it is:
  - Rather than allocating with `malloc`, host memory allocated using `cudaHostAlloc`
  - No magic on the hardware side - data moves back-and-forth over same PCI-E bus

- `cudaHostAlloc` cons
  - `cudaHostAlloc`-ing large amounts of memory can negatively impact overall system performance
    - Why? It reduces the amount of system memory available for paging
    - How much is too much? Not clear, dependent on system and applications running on system
  - `cudaHostAlloc` is slow - ballpark 5 GB/s
    - Timewise, allocating 5 GB of memory comparable to moving that much data over PCI-E bus

Key Benefits, `cudaHostAlloc`-ing Memory



- Three benefits to replacing `malloc` with `cudaHostAlloc`

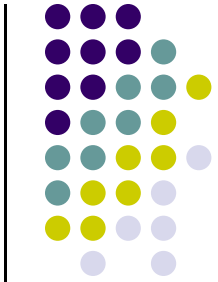
1. Faster device/host back-and-forth transfers

2. Enables the use of asynchronous memory transfer and kernel execution
   - Draws on the concept of CUDA stream

3. Enables mapping of pinned memory into memory space of the device
   - Device now capable to access data on host while executing a kernel or other device function

- Focus next is on 3 above

Zero-Copy (Z-C) GPU-CPU Interaction



- Last argument ("`flag`") controls the magic:

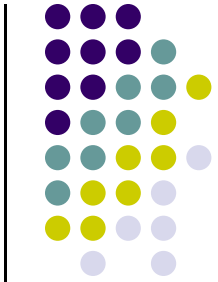
```
cudaError_t cudaHostAlloc(void** pHost, size_t size,unsigned int flag)
```

- "`flag`" values: `cudaHostAllocPortable`, `cudaHostAllocWriteCombined`, etc.

- The "`flag`" of interest: "`cudaHostAllocMapped`"
  - Maps the memory allocated on the host in the memory space of the device for direct access

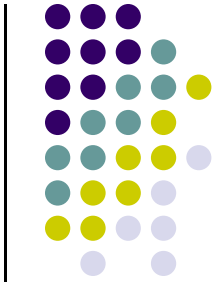
- What's gained:
  - The ability to access a piece of data from pinned and mapped host memory by a thread running on the GPU without a CUDA runtime copy call to explicitly move data onto the GPU
    - This is called zero-copy GPU-CPU interaction, from where the name "zero-copy memory"
    - Note that data is still moved through the PCI-E pipe, but it's done in a transparent fashion

Z-C, Further Comments



- More on the “flag” argument, which can take four values:
  - Use `cudaHostAllocDefault` argument for getting plain vanilla pinned host memory (call becomes identical in this case to cudaMallocHost call)
  - Use `cudaHostAllocMapped` to pick up the Z-C functionality
  - See documentation for `cudaHostAllocWriteCombined` the `cudaHostAllocPortable`
    - These two flags provide additional tweaks, irrelevant here
- The focus **should not be** on `cudaHostAlloc()` and the “`flag`”
- Focus **should be** on the fact that a device thread can directly access host memory

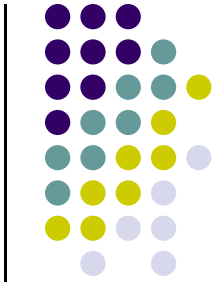
From Z-C to UVA: CUDA 2.2 to CUDA 4.0



- Z-C enabled access of data on the host from the device required one additional runtime call to `cudaHostGetDevicePointer`
  - `cudaHostGetDevicePointer`: given a pointer to pinned host memory produces a new pointer that can be invoked within a kernel to access data stored on the host

- The need for `cudaHostGetDevicePointer` call eliminated in CUDA 4.0 with the introduction of the Unified Virtual Addressing (UVA) mechanism

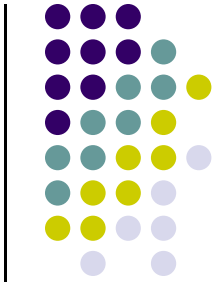
Unified Virtual Addressing: CUDA 4.0



- CUDA runtime identifies where data is stored based on value of the pointer
  - Possible since one address space used for the GPU memory and [some of] the CPU memory

- In a unified virtual address space setup, the runtime manipulates the pointer and allocation mappings used in device code (through `cudaMalloc`) as well as pointers and allocation mappings used in host code (through `cudaHostAlloc`) inside a single unified space

UVA - Consequences



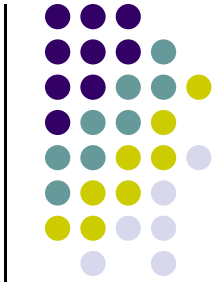
- No need to deal with `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, and `cudaMemcpyDeviceToDevice` scenarios
  - Simply use the generic `cudaMemcpyDefault` flag

- Technicalities regarding the need to call `cudaGetDeviceProperties()` for all participating devices (to check `cudaDeviceProp::unifiedAddressing` flag) to figure out whether they're game for UVA are skipped
  - See more here: http://sbel.wisc.edu/documents/TR-2014-09.pdf

- What this buys us: ability to do, for instance, inter-device copy without relying on the host for staging data movement:

```
cudaMemcpy(gpuDst_memPntr, gpuSrc_memPntr, byteSize, cudaMemcpyDefault);
```

UVA – A Nifty Feature of CUDA

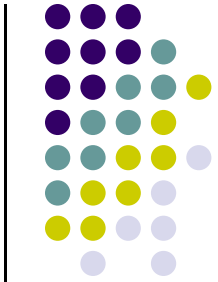


- Commands below can be issued by one host thread to multiple devices
  - No need to use anything beyond `cudaMemcpyDefault`

```
cudaMemcpy(gpu1Dst_memPntr, host_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(gpu2Dst_memPntr, host_memPntr, byteSize2, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu1Dst_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu2Dst_memPntr, byteSize2, cudaMemcpyDefault)
```

- UVA support: enabler for the peer-to-peer (P2P) inter-GPU data transfer
  - P2P not topic of discussion here

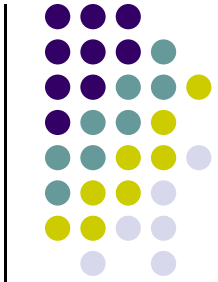
UVA: One Step Beyond Z-C



- Z-C Key Accomplishment: use pointer within device function access host data
  - Z-C focused on a data access issue relevant in the context of functions executed on the device

- UVA had a data access component but also a data transfer component:
  - Data access: A GPU could access data on a different GPU, a CUDA 4.0 novelty
    - This is already more than Z-C could accomplish (supported device to host access only)
  - Data transfer: copy data in between GPUs
    - `cudaMemcpy` is the main character in this play, data transfer initiated on the host side

Zero-Copy, UVA, and How UM Fits In

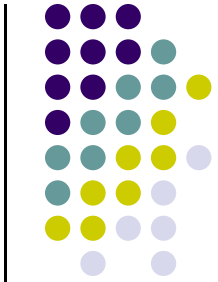


- Both for Z-C and UVA the memory was allocated on the device w/ `cudaMalloc` and on the host with `cudaHostAlloc`

- All the magic was based on the `cudaHostAlloc/cudaMalloc` interplay

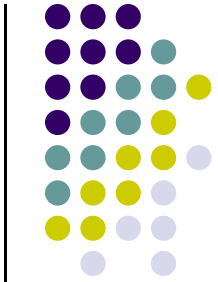
- Summary of the magic that could be done:
  - Data on host can be accessed on the device
  - Data transferred faster in between devices without intermediate staging on the host
  - Data stored by one GPU accessed directly by a different GPU
  - Etc.

Zero-Copy, UVA, and How UM Fits In



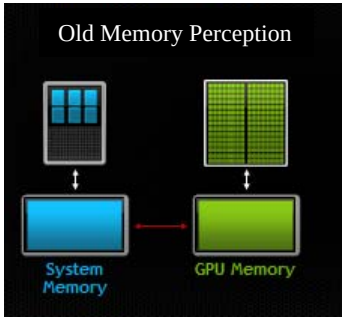
- Unified Memory (UM) eliminates the need to call the `cudaMalloc/cudaHostAlloc` duo
  - It takes a different perspective on handling memory in the GPU/CPU interplay
- Specifically, when it comes to accessing data on the host:
  - One would call cudaHostAlloc once use Z-C to access host data
    - This approach not recommended when having repeated accesses by device to host-side memory
    - Each device request that ends up accessing the host-side memory incurs high latency and low bandwidth (relative to the latency and bandwidth of an access to device global memory)
- This is the backdrop against which the role of UM is justified
  - Data is stored and migrated in a user-transparent fashion
    - To the extent possible, the data is right where it's needed thus enabling fast access

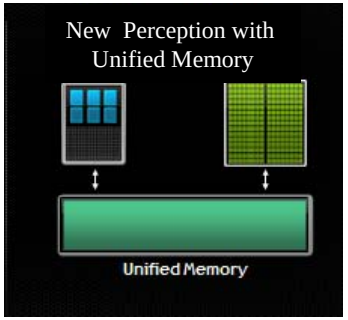
Unified Memory (UM)



- One memory allocation call takes care of memory setup at both ends; i.e., device and host
  - Main actor: the CUDA runtime function `cudaMallocManaged`

- New way of perceiving the memory interplay in GPGPU computing
  - No distinction is made between memory on the host and memory on the device
  - It's just memory, albeit with different access times when accessed by different processors

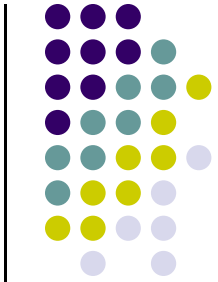




[NVIDIA]→

Unified Memory – Semantics Issues

[clarifications of terms used on previous slide]

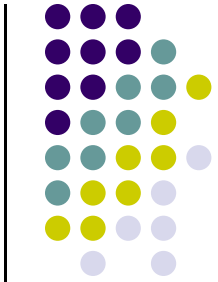


- "processor" (from NVIDIA documentation): "any independent execution unit with a dedicated memory management unit (MMU)"
  - Includes both CPUs and GPUs of any type and architecture

- "different access time": time is higher when, for instance, the host accesses for the first time data stored on the device
  - Subsequent accesses to the same data take place at the bandwidth and latency of accessing host memory
    - This is why access time is different and lower
    - Original access time higher due to migration of data from device to host
  - NOTE: same remarks apply to accesses from the device

[NVIDIA]→

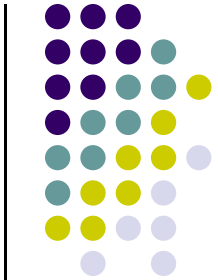
Now and Then: UM vs. no-UM



```cpp
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double* aArray, double val, unsigned int sz)
{
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

    for (int i = 0; i < ARRAY_SIZE; i++)
        mA[i] = 1.*i;

    double inc_val = 2.0;
    increment <<<2, 512 >>>(mA, inc_val, ARRAY_SIZE);
    cudaDeviceSynchronize();

    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += fabs(mA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(mA);
    return 0;
}
```

```cpp
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__global__ void increment(double* aArray, double val, unsigned int sz)
{
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main(int argc, char **argv) {
    double* hA;
    double* dA;
    hA = (double *)malloc(ARRAY_SIZE * sizeof(double));
    cudaMalloc(&dA, ARRAY_SIZE * sizeof(double));

    for (int i = 0; i < ARRAY_SIZE; i++)
        hA[i] = 1.*i;

    double inc_val = 2.0;
    cudaMemcpy(dA, hA, sizeof(double) * ARRAY_SIZE, cudaMemcpyHostToDevice);
    increment <<<2, 512 >>>(dA, inc_val, ARRAY_SIZE);
    cudaMemcpy(hA, dA, sizeof(double) * ARRAY_SIZE, cudaMemcpyDeviceToHost);

    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += fabs(hA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(dA);
    free(hA);
    return 0;
}
```

UM vs. Z-C

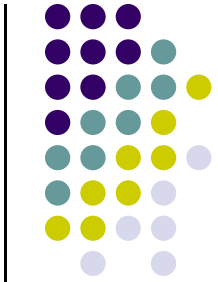


- Recall that with Z-C, data is always on the host in pinned CPU system memory
  - The device reaches out to it

- UM: data stored on the device but made available where needed
  - Data access and locality managed by CUDA runtime, handling transparent to user
  - UM provides “single-pointer-to-data” model

- Support for UM called for only *three* additions to CUDA:
  - `cudaMallocManaged`, `__managed__` , `cudaStreamAttachMemAsync`

Technicalities…



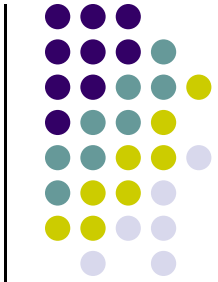
`cudaError_t cudaMallocManaged (void** devPtr, size_t size, unsigned int flag)`

- Returns pointer accessible from both Host and Device
- Drop-in replacement for `cudaMalloc` – they are semantically similar
- Allocates managed memory on the device
  - First two arguments have the expected meaning
- "`flag`" controls the default stream association for this allocation
  - `cudaMemAttachGlobal` - memory is accessible from any stream on any device
  - `cudaMemAttachHost` – memory on this device accessible by host only
- Free memory with the same `cudaFree()`

- `__managed__`
  - Global/file-scope variable annotation combines with `__device__`
  - Declares global-scope migrateable device variable
  - Symbol accessible from both GPU and CPU code

- `cudaStreamAttachMemAsync()`
  - Manages concurrency in multi-threaded CPU applications

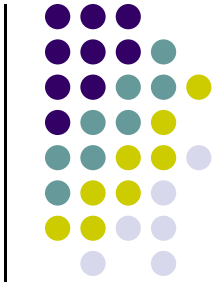
UM, Quick Points



- In the current implementation, managed memory is allocated on the device that happens to be active at the time of the allocation

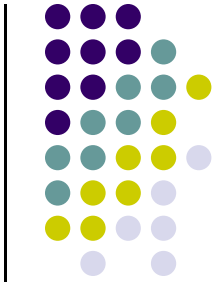
- Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the `cudaMalloc` routine
  - All CUDA operations valid on device memory are also valid on managed memory

Example: UM and thrust



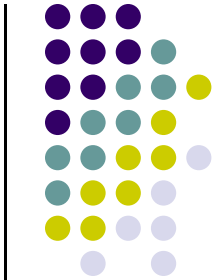
```
#include <ostream>
#include <cmath>
#include <thrust/reduce.h>
#include <thrust/system/cuda/execution_policy.h>
#include <thrust/system/omp/execution_policy.h>

const int ARRAY_SIZE = 1000;

int main(int argc, char **argv) {
  double* mA;
  cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));

  thrust::sequence(mA, mA + ARRAY_SIZE, 1);

  double maximumGPU = thrust::reduce(thrust::cuda::par, mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());
  cudaDeviceSynchronize();
  double maximumCPU = thrust::reduce(thrust::omp::par , mA, mA + ARRAY_SIZE, 0.0, thrust::maximum<double>());

  std::cout << "GPU reduce: " << (std::fabs(maximumGPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;
  std::cout << "CPU reduce: " << (std::fabs(maximumCPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;

  cudaFree(mA);

  return 0;
}
```

Advanced Features: UM



- Managed memory migration is at the page level
  - The default page size is currently the same as the OS page size today (typically 4 KB)
- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCI-E only the dirty pages
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed
- Coherence points are kernel launch and device/stream sync.
  - Important: the same memory cannot be operated upon, at the same time, by the device and host

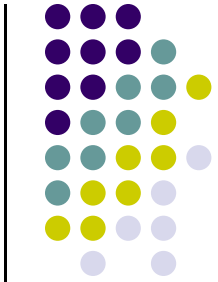
Advanced Features: UM



- Issues related to “managed memory size”:
  - For now, there is no oversubscription of the device memory
    - In fact, if there are several devices available, the max amount of managed memory that can be allocated is the smallest of the memories available on the devices

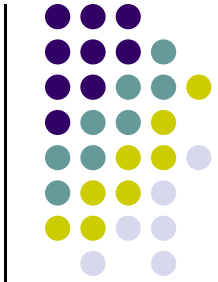
- Issues related to “transfer/execution overlap”:
  - Pages from managed allocations touched by CPU migrated back to GPU before any kernel launch
    - Consequence: there is no kernel execution/data transfer overlap in that stream
    - Overlap possible with UM but just like before it requires multiple kernels in separate streams
      - Enabled by the fact that a managed allocation can be specific to a stream
      - Allows one to control which allocations are synchronized on specific kernel launches, enables concurrency

UM: Coherency Related Issues



- The GPU has *exclusive* access to this memory when any kernel is executed on the device
  - Holds even if during its execution the kernel doesn't touch the managed memory
- The CPU cannot access *any* managed memory allocation or variable as long as GPU is executing
- A `cudaDeviceSynchronize` call required for the host to be allowed to access managed memory
  - To this end, any function that logically guarantees the GPU finished execution is acceptable
    - Examples: `cudaStreamSynchronize`(), `cudaMemcpy`(), `cudaMemset`(), etc.

Left: Seg fault
Right: Runs ok

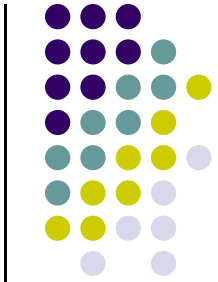


```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    y = 20; // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();
    return 0;
}
```

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    cudaDeviceSynchronize();
    y = 20; // GPU is idle so access is OK
    return 0;
}
```


[NVIDIA, Programming Guide]→

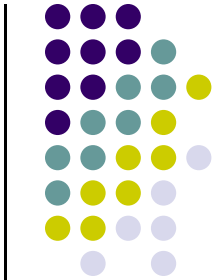
UM – Limitations in CUDA 6.0



- Ability to allocate more memory than the physically available on the GPU
- Prefetching
- Finer Grain Migration

- **NOTE**: haven't checked if any of these addressed in CUDA 7.5

UM – Why Bother?



1. A matter of convenience
   - Much simpler to write code using this memory model
   - For the casual programmer, the code will run faster due to data locality
     - The runtime will take care of moving the data where it ought to be

2. Looking ahead, physical CPU/GPU integration prevalent – memory can be shared
   - Already the case for integrated GPUs that are part of the system chipset
   - The trend in which the industry is moving (AMD's APU, Intel's Haswell, NVIDIA Denver Project)
   - The functionality provided by the current software backend that supports the `cudaMallocManaged()` paradigm will be eventually implemented in hardware