

A decorative element on the left side of the slide, consisting of two vertical lines: a blue line on the left and an orange line on the right, both extending from the top to the bottom of the slide.

Advanced CUDA Feature Highlights

Homework Assignment #3

Problem 2: Select one of the following questions below. Write a CUDA program that illustrates the “optimization benefit” (OB) or “performance cliff” (PC) in the example. These codes will be shared with the rest of the class. Also provide a brief (a few sentences) description of what is happening as a comment inside the code.

[PC] Show an example code where you fill up the register file due to too many threads. You should have two versions of the code, one where the number of threads is within the range of registers, and one where the register capacity is exceeded.

[OB] Show the performance impact of unrolling an innermost loop in a nest. See how far you can push it before you run into the problems of a. above.

[OB/PC] Explore when the compiler decides to put array variables that are local to the device function in registers. What access patterns lead to the compiler using a register vs. using local memory.

[OB/PC] Show the performance advantage of constant memory when the data is cached, and what happens to performance when the data exceeds the cache capacity and locality is not realized.

Homework Assignment #3

Problem 2, cont.:

[OB] Show the performance impact of control flow versus no control flow. For example, use the trick from slide #13 of Lecture 9 and compare against testing for divide by 0.

[PC] Demonstrate the performance impact of parallel memory access (no bank conflicts) in shared memory. For example, implement a reduction computation like in Lecture 9 in shared memory, with one version demonstrating bank conflicts and the other without.

[OB] Show the performance impact of global memory coalescing by experimenting with different data and computation partitions in the matrix addition example from lab1.

General

Timing accuracy

Event vs. timer

Duration of run as compared to timer granularity

What is standard deviation?

Consider other overheads that may mask the thing you are measuring

For example, global memory access versus control flow

Errors encountered

Erroneous results if max number of threads exceeded (512), but apparently no warning...

a. Exceeding register capacity

Compile fails if code exceeds number of available registers. (supposed to spill to “local” memory?)

Simple array assignment with slightly more variables

Compare 7680 registers vs. 8192 registers

1.5x performance difference!

b. Impact of Loop Unrolling

Unroll inner loop from a tiled code Program

Compute 16 elements with fully unrolled loop

Performance difference negligible

EITHER, too much unrolling so performance harmed

OR, timing problem

d. Constant cache

```
// d_b in constant memory and small enough to fit in cache
__global__ void cache_compute(float *a) :
    for(int j=0; j<100000; j++) a[(j+threadIdx.x) % n] += d_b[(j+threadIdx.x)
% n];

// d_b2 in constant memory
__global__ void bad_cache_compute(float *a):
    for(int j=0; j<100000; j++) a[(j+threadIdx.x) % BadCacheSize] +=
d_b2[(j+threadIdx.x) % BadCacheSize];

// b in global memory
__global__ void no_cache_compute(float *a, float *b) :
    for(int j=0; j<100000; j++) a[(j+threadIdx.x) % n] += b[(j+threadIdx.x) %
n];
```

1.2x and 1.4x performance improvements, respectively, when input fits in cache vs. not as compared to global memory.

Similar example showed 1.5X improvement.

e. Control flow versus no control flow

```
float val2 = arr[index];  
// has control flow to  
check for divide by zero  
if(val1 != 0)  
    arr[index] =  
val1/val2;  
else  
    arr[index] = 0.0;
```

```
float val2 = arr[index];  
// approximation to avoid to  
control flow  
  
val1 += 0.00000000000000000001;  
arr[index] = val1/val2;
```

2.7X performance difference!
(similar examples showed 1.9X and 4X
difference!)

Another example,
check for divide by 0 in reciprocal
1.75X performance difference!

e. Control flow vs. no control

```
for(int i=0; i < ARRAYLOOP; i++)
  switch(z)
    case 0: a_array[threadIdx.x] += 18;
           break;
    case 1: a_array[threadIdx.x] += 9;
           break;
    ...
    case 7: a_array[threadIdx.x] += 15;
           break;
}
```

flow (switch)

```
for(int i=0; i < ARRAYLOOP; i++)
  efficientArray[i] = 18;
  ...
  efficientArray[7] = 15;
  __syncthreads();
  for(int j=0; j < ARRAYLOOP;
    j++)
    for(int i=0; i <
      ARRAYLOOP; i++)
      a_array[threadIdx.x] +=
        efficientArray[z];
```

Eliminating the switch statement makes a 6X performance difference!

f. Impact of bank conflicts

```
if ( cause_bank_conflicts ) {
    min = id * num_banks ;
    stride = 1;
    max = (id + 1) * num_banks;
}
else {
    min = id;
    stride = num_banks ;
    max = ( stride * ( num_banks -
1))
    + min + 1;
}
```

```
for (j = min; j < max; j+=
stride )
```

```
    mem[j] = 0;
```

```
for (i = 0; i < iters ; i++)
```

```
    for (j = min; j < max;
j+= stride )
```

```
        mem[j]++;
```

```
for (j = min; j < max; j+=
stride )
```

```
    out[j] = mem[j];
```

5X difference in performance!

Another example showed 11.3X difference!

g. Global memory coalescing

Experiment with different computation and data partitions for matrix addition code

Column major and row major, with different data types

Row major?

Column major results

Exec time for

Double 77 ms

Float 76ms

Int 57 ms

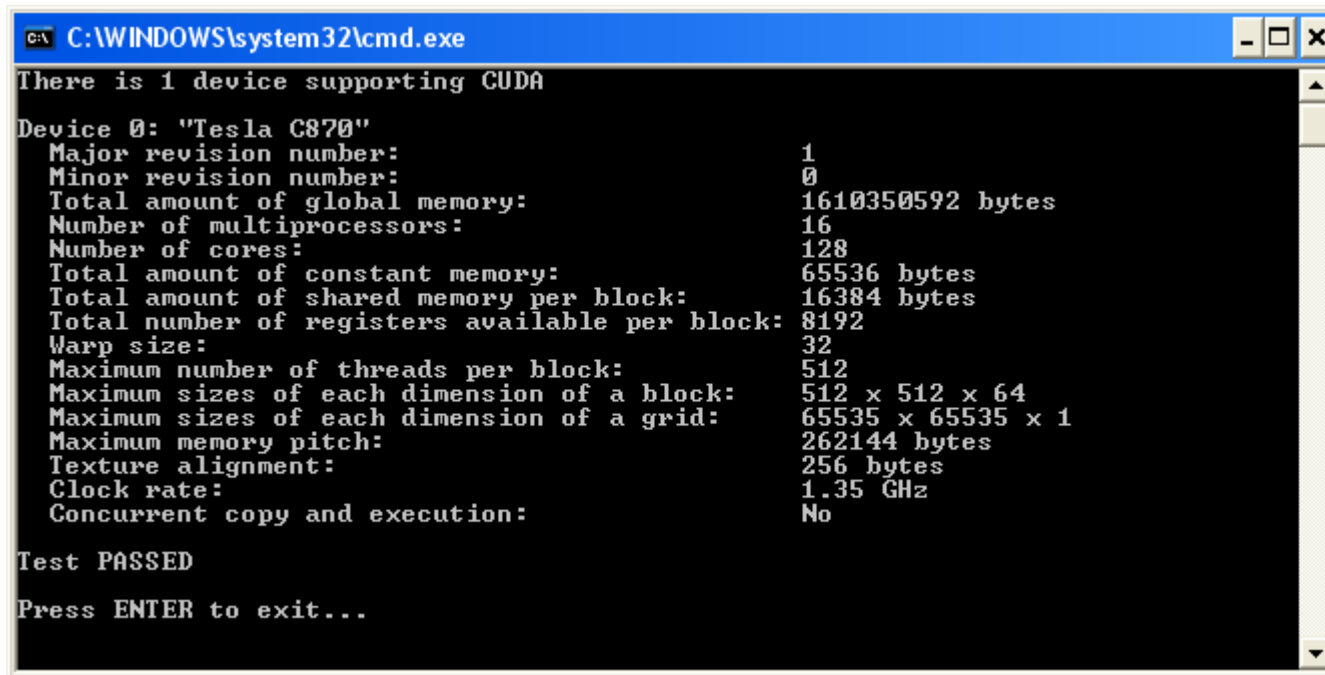
Char 31 ms

Capacity Questions

How much shared memory, global memory, registers, constant memory, constant cache, etc.?

deviceQuery function (in SDK) instantiates variable of type cudaDeviceProp with this information and prints it out.

Summary for my card



```
C:\WINDOWS\system32\cmd.exe
There is 1 device supporting CUDA
Device 0: "Tesla C870"
Major revision number:          1
Minor revision number:          0
Total amount of global memory:  1610350592 bytes
Number of multiprocessors:      16
Number of cores:                 128
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 16384 bytes
Total number of registers available per block: 8192
Warp size:                       32
Maximum number of threads per block: 512
Maximum sizes of each dimension of a block: 512 x 512 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
Maximum memory pitch:           262144 bytes
Texture alignment:              256 bytes
Clock rate:                     1.35 GHz
Concurrent copy and execution:   No

Test PASSED
Press ENTER to exit...
```

Objective

- To mention and categorize some of the most relevant “advanced” features of CUDA
 - The goal is awareness, not necessarily detailed instruction
 - Be aware that I haven't personally tried many of these
- The majority of features here will probably not be necessary or useful for any particular application
- These features encompass a range of programming prowess needed to use them effectively
- I'll be referencing CUDA Programming Manual (CPM) 2.0 sections frequently if you want to dive in more
 - Chapter 4 is the API chapter, if you're browsing for features 13

Agenda

- Tools
 - More nvcc features, profiler, debugger, Komrade, MCUDA
- A note on pointer-based data structures
- Warp-level intrinsics
- Streams
- Global memory coalescing
- Short Vectors
- Textures
- Atomic operations
- Page-locked memory & zero-copy access
- Graphics interoperability
- Dynamic compilation

Tools: nvcc

- Some nvcc features:
 - `--ptxas-options=-v`
 - Print the smem, register and other resource usages
 - `#pragma unroll X`
 - You can put a pragma right before a loop to tell the compiler to unroll it by a factor of X
 - Doesn't enforce correctness if the loop trip count isn't a multiple of X
 - CPM 4.2.5.2

Tools: profiler and debugger

- The cuda profiler can be used from a GUI or on the command line
 - Cuda profiler collects information from specific counters for things like branch divergence, global memory accesses, etc.
 - Only instruments one SM: so your results are only as representative as the sample scheduled to that SM.
- `cuda-gdb`
 - Debugger with gdb-like interface that lets you set breakpoints in kernel code while it's executing on the device, examine kernel threads, and contents of host and device memory

reduction - CUDA Visual Profiler - [kernel0]

File Profile Session Options Window Help

kernel0
kernel1
kernel2
kernel3
kernel4
kernel5
kernel6

Profiler Output

	Timestamp	Method	GPU Time	CPU Time	Occupancy	grid size X	grid size Y	block size X	block size Y
1	50286.7	memcpy	1573.82	4341.88					
2	54703.4	reduce0_sm10	2027.23	2050.6	1	1	8192	128	1
3	56956.5	reduce0_sm10	2024.9	2042.85	1	1	8192	128	1
4	59164.6	reduce0_sm10	18.4	31.397	1	1	64	128	1
5	59338.5	reduce0_sm10	7.264	24.089	0.667	1	1	64	1
6	59505.2	reduce0_sm10	2023.71	2040.15	1	1	8192	128	1
7	61687.3	reduce0_sm10	18.368	31.069	1	1	64	128	1
8	61859.7	reduce0_sm10	7.232	23.577	0.667	1	1	64	1
9	62026.8	reduce0_sm10	2026.66	2042.92	1	1	8192	128	1
10	64211.4	reduce0_sm10	18.368	30.923	1	1	64	128	1
11	64383.5	reduce0_sm10	7.296	23.535	0.667	1	1	64	1
12	64550.2	reduce0_sm10	2025.41	2041.86	1	1	8192	128	1
13	66737.3	reduce0_sm10	18.4	31.402	1	1	64	128	1
14	66910.3	reduce0_sm10	7.296	24.099	0.667	1	1	64	1
15	67077.3	reduce0 sm10	2023.23	2039.63	1	1	8192	128	1

kernel5 : Column 'local store' having all zero values is hidden.
kernel5 : Column 'warp serialize' having all zero values is hidden.
kernel6 : Column 'stream id' having all zero values is hidden.
kernel6 : Column 'gld uncoalesced' having all zero values is hidden.
kernel6 : Column 'local load' having all zero values is hidden.
kernel6 : Column 'local store' having all zero values is hidden.
kernel6 : Column 'warp serialize' having all zero values is hidden.

reduction - CUDA Visual Profiler - [kernel0]

File Profile Session Options Window Help

kernel0
kernel1
kernel2
kernel3
kernel4
kernel5
kernel6

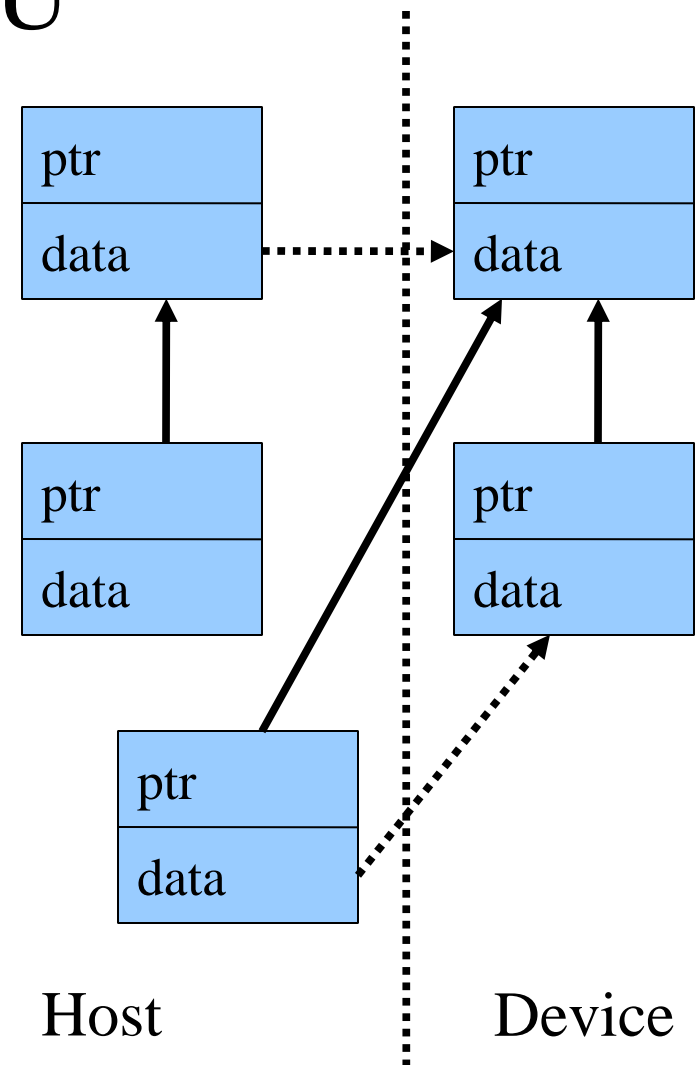
Profiler Output

	block size Z	dynamic shared memory per block	static shared memory per block	registers per thread	mem transfer size	mem transfer dir	gld coalesced	gst uncoalesced	gst coalesced	branch	divergent branch	instructions	cta launched
1					4194304	0							
2	1	512	32	10			8192	1024	2048	64000	512	325656	1024
3	1	512	32	10			8192	1024	2048	64000	512	329662	1024
4	1	512	32	10			64	8	16	500	4	2914	8
5	1	256	32	10			4	0	4	55	1	567	1
6	1	512	32	10			8192	2048	0	64000	512	326310	1024
7	1	512	32	10			64	16	0	500	4	2885	8
8	1	256	32	10			0	0	0	0	0	0	0
9	1	512	32	10			8192	2048	0	64000	512	327962	1024
10	1	512	32	10			64	16	0	500	4	2884	8
11	1	256	32	10			0	0	0	0	0	0	0
12	1	512	32	10			8192	2048	0	64000	512	324599	1024
13	1	512	32	10			64	16	0	500	4	2838	8
14	1	256	32	10			0	0	0	0	0	0	0
15	1	512	32	10			8192	2048	0	64000	512	325555	1024

kernel5 : Column 'local store' having all zero values is hidden.
kernel5 : Column 'warp serialize' having all zero values is hidden.
kernel6 : Column 'stream id' having all zero values is hidden.
kernel6 : Column 'gld uncoalesced' having all zero values is hidden.
kernel6 : Column 'local load' having all zero values is hidden.
kernel6 : Column 'local store' having all zero values is hidden.
kernel6 : Column 'warp serialize' having all zero values is hidden.

Moving pointer-based data structures to the GPU

- Device pointers and host pointers are not the same
- For an internally-consistent data structure on the device, you need to write data structures with device pointers on the host, and then copy them to the device

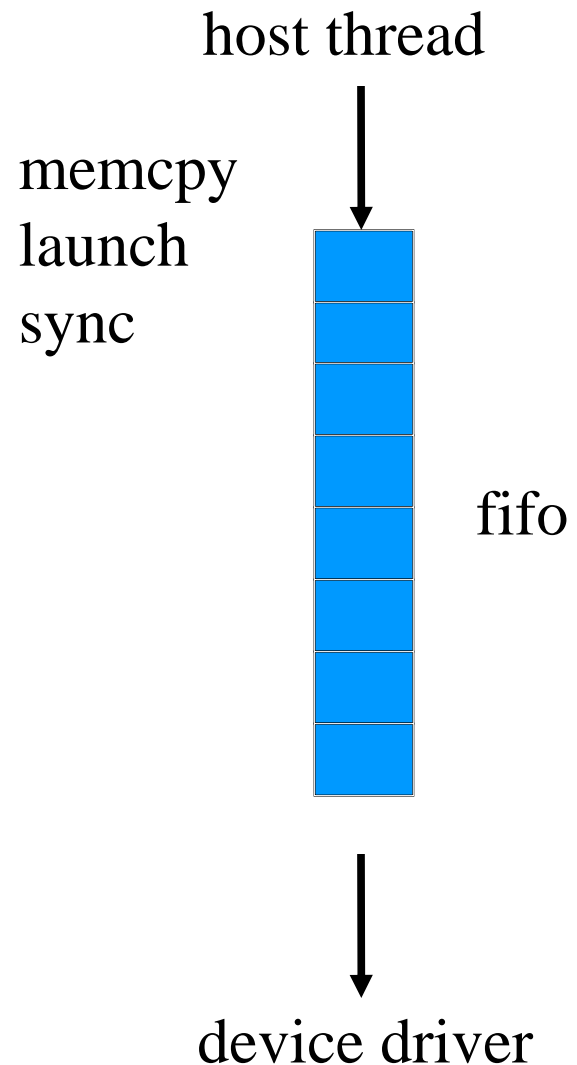


Warp-level intrinsics

- `warpSize`
 - Another built-in variable for the number of threads in a warp
 - If you -have- to write code dependent on the warp size, do it with this variable rather than “32” or something else
- Warp voting
 - `WarpAnd`, `warpOr`
 - Allows you to do a one-bit binary reduction in a warp with one instruction, returning the result to every thread
 - CPM 4.4.5

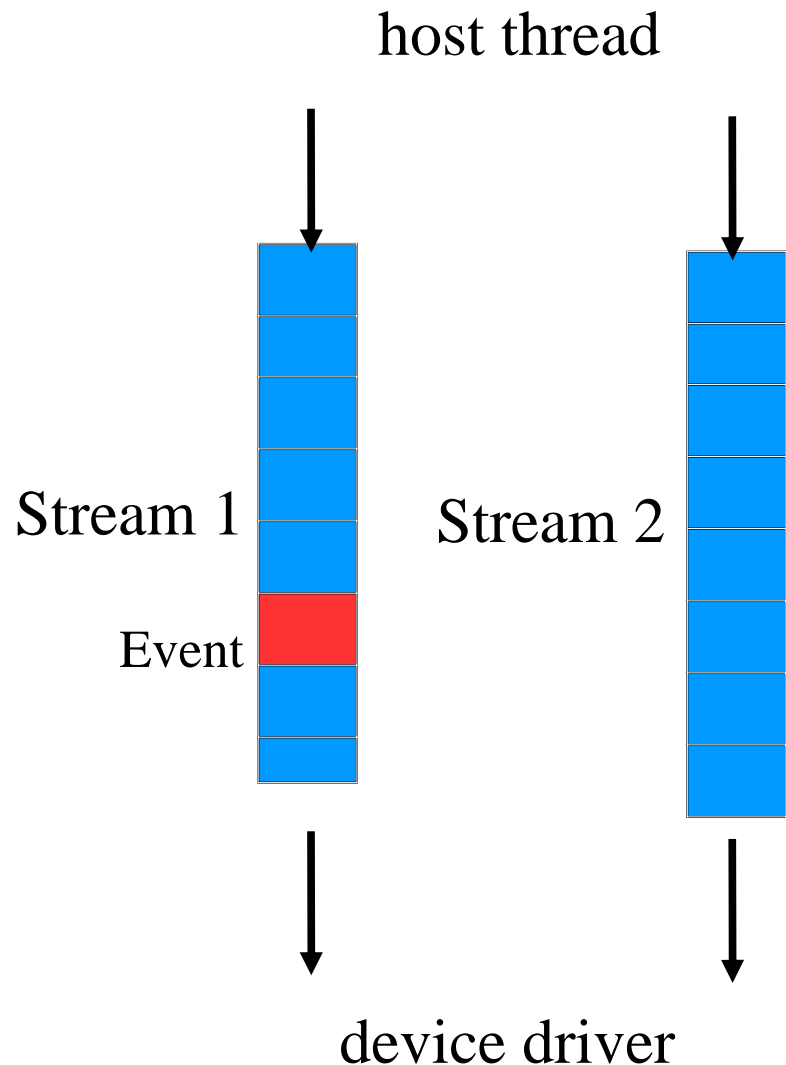
Streams

- All device requests made from the host code are put into a queue
 - Queue is read and processed asynchronously by the driver and device
 - Driver ensures that commands in the queue are processed in sequence. Memory copies end before kernel launch, etc.



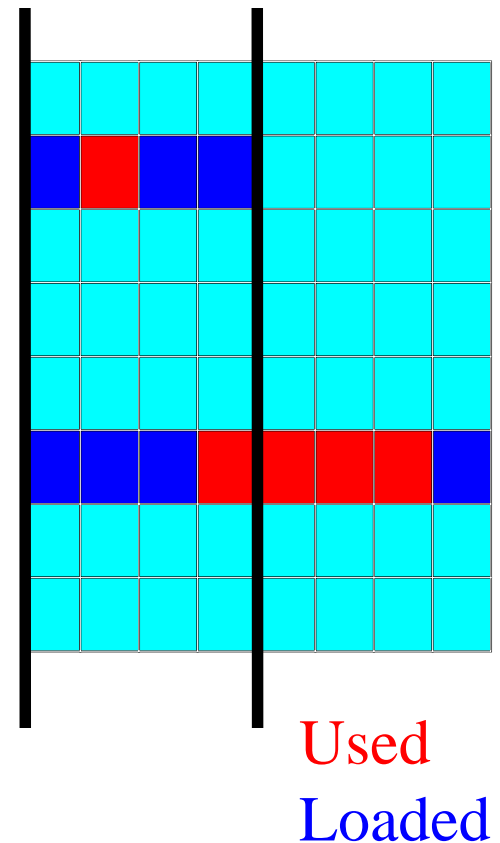
Streams cont.

- To allow concurrent copying and kernel execution, you need to use multiple queues, called “streams”
 - Cuda “events” allow the host thread to query and synchronize with the individual queues.



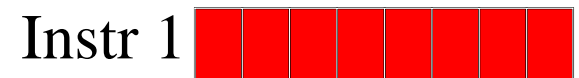
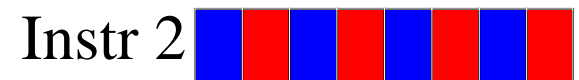
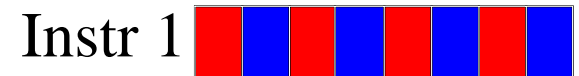
Global memory coalescing

- Global memory locations are laid out contiguously in memory
 - Sets of adjacent locations are stored in DRAM “lines”
 - The memory system is only capable of loading lines, even if only a single element from the line was needed
 - Any data from the line not used is “wasted” bandwidth
- Arrange accesses so that threads in a warp access the fewest lines possible
 - CPM 5.1.2.1



Short vector types

- Array of multi-element data structures
 - Linearized access pattern uses multiple times the necessary bandwidth
 - Short vector types don't waste bandwidth, and use one instruction to load multiple elements
 - int2, char4, etc.
 - It is possible to create your own short-vector types
 - Your code may not already use .x .y .z component names
 - CPM 4.3.1



Page-locked memory and zero-copy access

- Page-locked memory is memory guaranteed to actually be in memory
 - In general, the operating system is allowed to “page” your memory to a hard disk if it's too big, not currently in use, etc.
- `cudaMallocHost()` / `cudaFreeHost()`
 - Allocates page-locked memory on the host
 - Significantly faster for copying to and from the GPU
 - Beginning with CUDA 2.2, a kernel can directly access host page-locked memory – no copy to device needed
 - Useful when you can't predetermine what data is needed
 - Less efficient if all data will be needed anyway
 - Could be worthwhile for pointer-based data structures as well

Graphics interoperability

- Want to render and compute with the same data?
 - CUDA allows you to map OpenGL and Direct3D buffer objects into CUDA
 - Render to a buffer, then pass it to CUDA for analysis
 - Or generate some data in CUDA, and then render it directly, without copying it to the host and back
 - CPM 4.5.2.7 (OpenGL), 4.4.2.8 (Direct3D)

Dynamic compilation

- The CUDA driver has a just-in-time compiler built in
 - Currently only compiles PTX code
 - Still, you can dynamically generate a kernel in PTX, then pass it to the driver to compile and run
 - Some applications have seen significant speedup by compiling data-specific kernels
 - John Stone et al. *High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs*. GPGPU-2, pp. 9-18, 2009

cudaMemcpyAsync

```
cudaError_t cudaMemcpy( void* dst, const void* src, size_t count, enum  
    cudaMemcpyKind kind  
)
```

```
cudaError_t cudaMemcpyAsync( void* dst, const void* src, size_t count, enum  
    cudaMemcpyKind  
kind, cudaStream_t stream )
```

requires pinned host memory (allocated with
“cudaMallocHost”)

Asynchronous API



- Asynchronous host \leftrightarrow device memory copies for pinned memory
- Concurrent execution of kernels and memory copies (on compute capability ≥ 1.1)
- Only possible through stream abstraction
- Stream = Sequence of operations that execute in order
- Stream API:
 - `cudaStreamCreate(&stream)`
 - `cudaMemcpyAsync(src, dst, size, stream)`
 - `kernel<<<grid, block, shared, stream>>> (...)`
 - `cudaStreamQuery(stream)`
 - `cudaStreamSynchronize(stream)`

```
cudaStreamCreate (&stream1) ;  
cudaStreamCreate (&stream2) ;  
cudaMemcpyAsync (dst, src, size, dir, stream1) ; ←  
kernel<<<grid, block, 0, stream2>>> (...) ; ←  
overlapped
```

Overlap CPU computation with data transfer

- 0 = default stream

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

Things keep changing

- Subscribe as an NVIDIA developer if you want to keep up with the newest features as they come down from NVIDIA
 - developer.nvidia.com/page/home.html
- Keep up on publications and interactions if you want to see new features or new uses of features
 - IACAT seminars and brownbags - www.iacat.illinois.edu
 - Workshops with GPGPU-related topics - www.gpgpu.org