

Intro to CUDA Programming

Based upon slides by John Pormann
Duke University

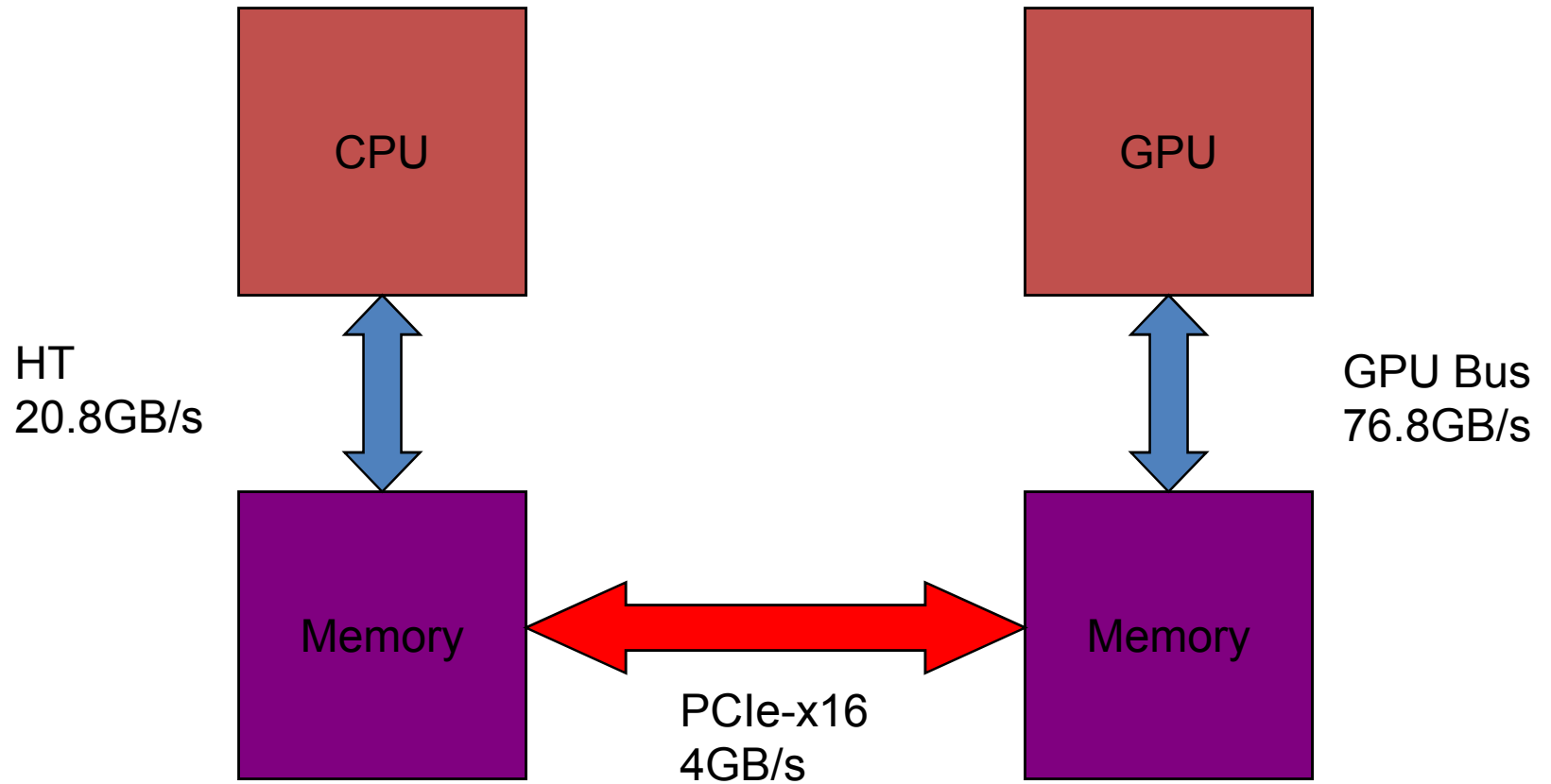
Overview

- Intro to the Operational Model
- Simple Example
 - Memory Allocation and Transfer
 - GPU-Function Launch
- Grids of Blocks of Threads
- GPU Programming Issues
- Performance Issues/Hints

Operational Model

- CUDA assumes a heterogeneous architecture -- both CPUs and GPUs -- with separate memory pools
 - CPUs are “masters” and GPUs are the “workers”
 - CPUs launch computations onto the GPU
 - CPUs can be used for other computations as well
 - GPUs have limited communication back to CPU
 - CPU must initiate data transfers to the GPU memory
 - Synchronous Xfer -- CPU waits for xfer to complete
 - Async Xfer -- CPU continues with other work, can check if xfer is complete

Operational Model, cont'd



Basic Programming Approach

- Transfer the input data out to the GPU
- Run the code on the GPU
 - Simultaneously run code on the CPU (??)
 - Can run multiple GPU-code-blocks on the GPU sequentially
- Transfer the output data back to the CPU

Slightly-Less-Basic Programming Approach

- In many cases, the output data doesn't need to be transferred as often
 - Iterative process -- leave data on the GPU and avoid some of the memory transfers
 - ODE Solver -- only transfer every 10th time-step
- Transfer the input data out to the GPU
- Loop:
 - Run the code on the GPU
 - Compute error on the GPU
 - If error > tolerance, continue
- Transfer the output data back to the CPU

Simple Example

```
__global__ void vcos( int n, float* x, float* y ) {
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    y[ix] = cos( x[ix] );
}

int main() {
    float *host_x, *host_y;
    float *dev_x, *dev_y;
    int n = 1024;

    host_x = (float*)malloc( n*sizeof(float) );
    host_y = (float*)malloc( n*sizeof(float) );
    cudaMalloc( &dev_x, n*sizeof(float) );
    cudaMalloc( &dev_y, n*sizeof(float) );

    /* TODO: fill host_x[i] with data here */
    cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

    /* launch 1 thread per vector-element, 256 threads per block */
    bk = (int)( n / 256 );
    vcos<<<bk,256>>>( n, dev_x, dev_y );

    cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
    /* host_y now contains cos(x) data */

    return( 0 );
}
```

Simple Example, cont'd

```
host_x = (float*)malloc( n*sizeof(float) );  
host_y = (float*)malloc( n*sizeof(float) );  
cudaMalloc( &dev_x, n*sizeof(float) );  
cudaMalloc( &dev_y, n*sizeof(float) );
```

- This allocates memory for the data
 - C-standard 'malloc' for host (CPU) memory
 - 'cudaMalloc' for GPU memory
 - DON'T use a CPU pointer in a GPU function !
 - DON'T use a GPU pointer in a CPU function !
 - And note that CUDA cannot tell the difference, YOU have to keep all the pointers straight!!!

Simple Example, con'd

```
cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );
```

```
. . .
```

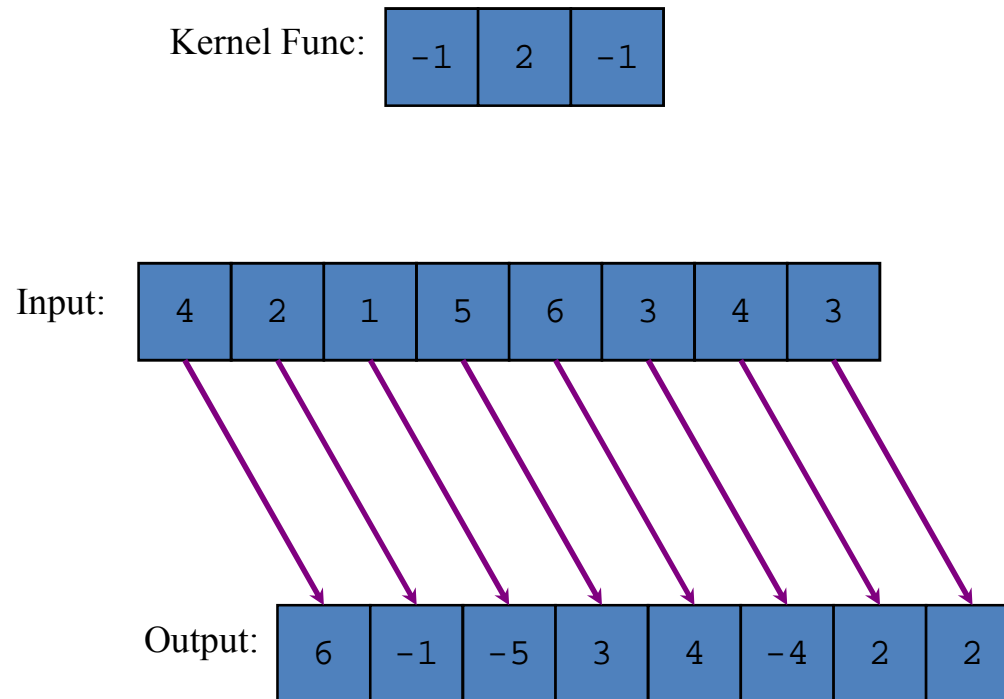
```
cudaMemcpy( host_y, dev_y, n*sizeof(float), cudaMemcpyDeviceToHost );
```

- This copies the data between CPU and GPU
 - Again, be sure to keep your pointers and direction (CPU-to-GPU or GPU-to-CPU) consistent !
 - CUDA cannot tell the difference so it is up to YOU to keep the pointers/directions in the right order
 - ‘cudaMemcpy’ ... think ‘destination’ then ‘source’

Stream Computing

- GPUs are multi-threaded computational engines
 - They can execute hundreds of threads simultaneously, and can keep track of thousands of pending threads
 - Note that GPU-threads are expected to be short-lived, you should not program them to run for hours continuously
 - With thousands of threads, general-purpose multi-threaded programming gets very complicated
 - We usually restrict each thread to be doing “more or less” the same thing as all the other threads ... SIMD/SPMD programming
 - Each element in a stream of data is processed with the same kernel-function, producing an element-wise stream of output data
 - Previous GPUs had stronger restrictions on data access patterns, but with CUDA, these limitations are gone (though performance issues may still remain)

Sequential View of Stream Computing

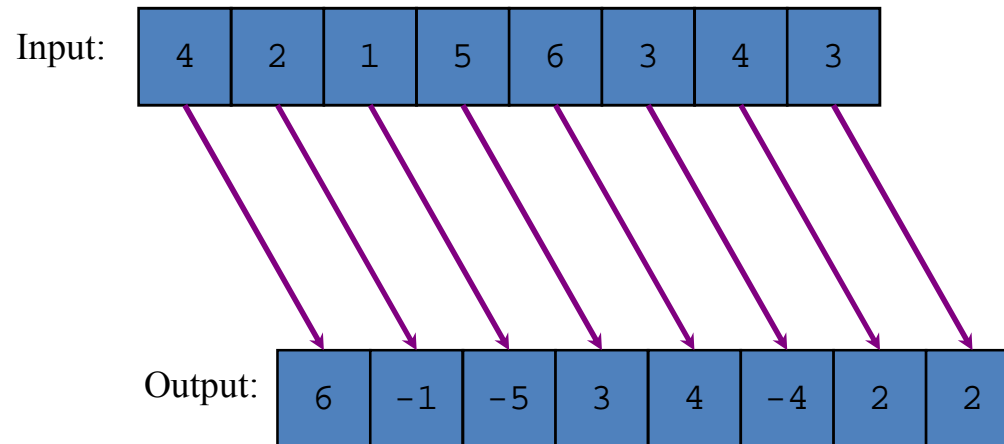


Sequential computation ... 8 clock-ticks

Parallel (GPU) View of Stream Computing

Kernel Func:

-1	2	-1
----	---	----



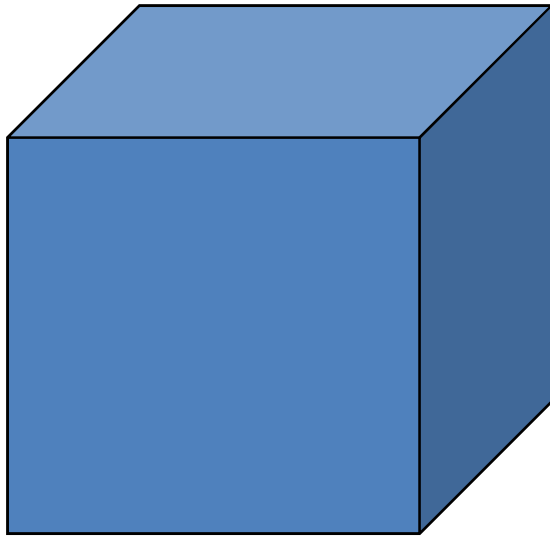
Parallel (4-way) computation ... 2 clock-ticks
... NVIDIA G80 has 128-way parallelism !!

GPU Task/Thread Model

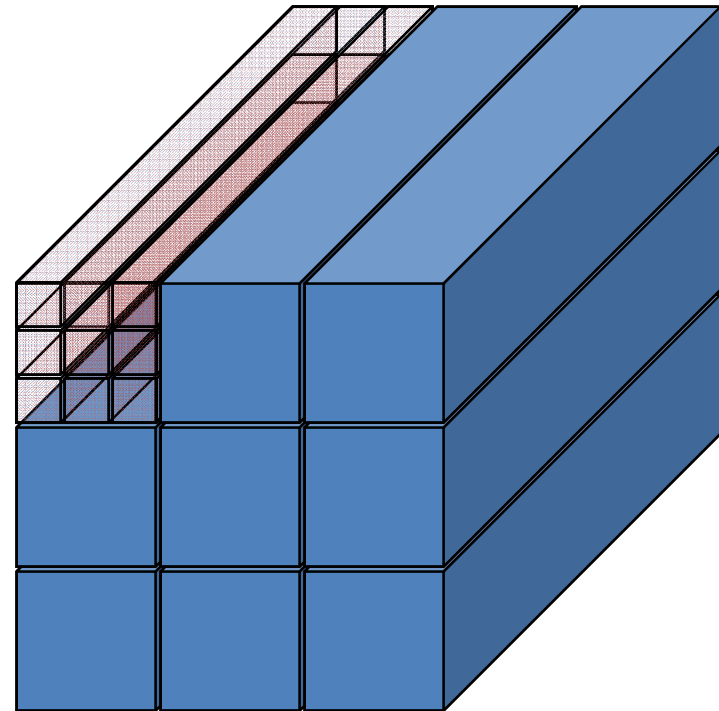
- launch hundreds or thousands GPU threads all at once
 - hardware handles how to run/manage them
- a “grid” of “blocks” of “threads” onto a GPU
 - Grid = 1- or 2-D configuration of a given size
 - Grid dims ≤ 65536
 - Block = 1-,2-,3-D config of a given size
 - Block dims ≤ 512 , total ≤ 768 threads
 - each thread must know how to configure itself using only Grid and Block coordinates
 - Similar to MPI's `MPI_Comm_rank`

← NVIDIA G100 allows 1024

CUDA Grid Example



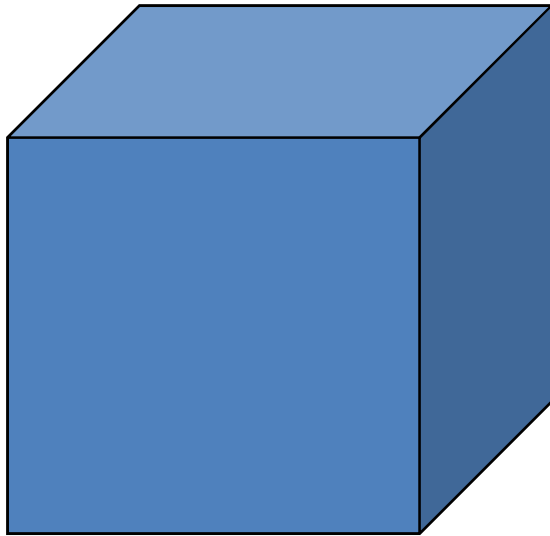
Real problem: 300x300x300



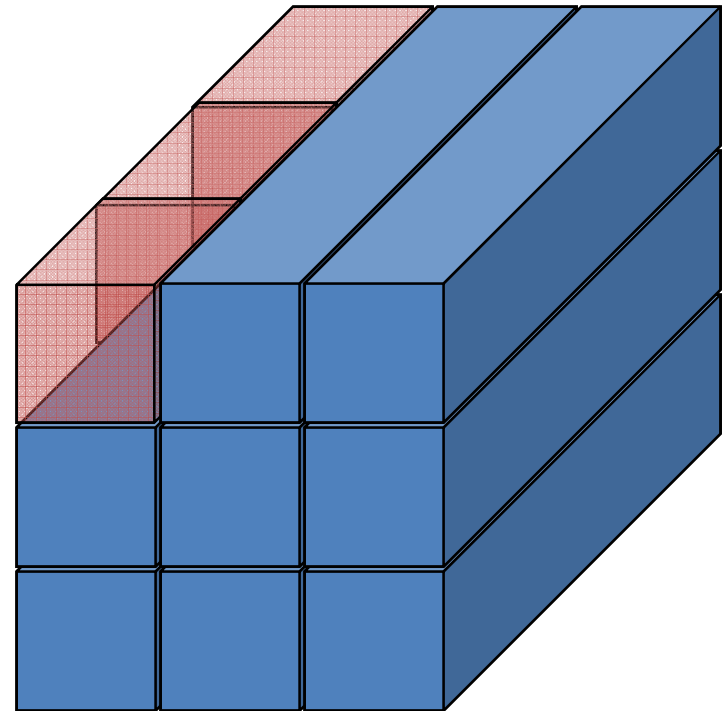
Grid: 3x3x1
Block: 3x3x1

Each block handles 100x100x300
Each thread handles ~ 33x33x300

CUDA Grid Example, cont'd



Real problem: 300x300x300



Grid: 3x3x1
Block: 1x1x3

Each block handles 100x100x300
Each thread handles 100x100x100

Returning to the Simple Example

```
/* launch 1 thread per vector-element, 256 threads per block */  
bk = (int)( n / 256 );  
vcos<<<bk,256>>>( n, dev_x, dev_y );
```

- The 'vcos<<<m,n>>>' syntax is what launches ALL of the GPU threads to execute the 'vcos' GPU-function
 - Launches 'm' grid blocks, each of size 'n' threads
 - Total of 'm*n' GPU-threads are created
 - Each thread has a unique {blockIdx.x,threadIdx.x}
 - 'm' and 'n' can also be 'uint3' (3-D) objects

```
uint3 m,n;  
m = make_uint3(128,128,1);  
n = make_uint3(32,32,1);  
  
uint3 m,n;  
m.x=128; m.y=128; m.z=1;  
n.x=32; n.y=32; n.z=1;  
  
vcos<<<m,n>>>( n, dev_x, dev_y );
```


CPU Threads vs. GPU Threads

- CPU Threads (POSIX Threads) are generally considered long-lived computational entities
 - Create 1 CPU-thread per CPU-core
 - keep them for the duration of the program
 - CPU-thread creation can take several micro secs
 - need a lot of operations to amortize start-up cost
- GPU Threads are generally short-lived
 - Create 1000's of GPU-threads
 - do small amount of computation before exiting
 - GPU-thread creation is very fast
 - create 1000's GPU-threads in a few clock ticks

Mapping the Parallelism to Threads

```
__global__ void vcos( int n, float* x, float* y ) {  
    int ix = blockIdx.x*blockDim.x + threadIdx.x;  
    y[ix] = cos( x[ix] );  
}
```

- GPU function, vcos or vector-cosine
 - ‘int n’ is the vector size
 - ‘float* x’ is the input vector
 - ‘float* y’ is the output vector
- ‘int ix’ is the global index number for this thread
 - We compute it from the built-in, thread-specific variables (set by the run-time environment)
 - Each GPU-thread has a unique combination of
`{blockIdx.x, threadIdx.x}`
 - GPU-thread has a unique ‘ix’ value
 - make sure that all data is processed (i.e. that all valid ‘ix’ values are hit)

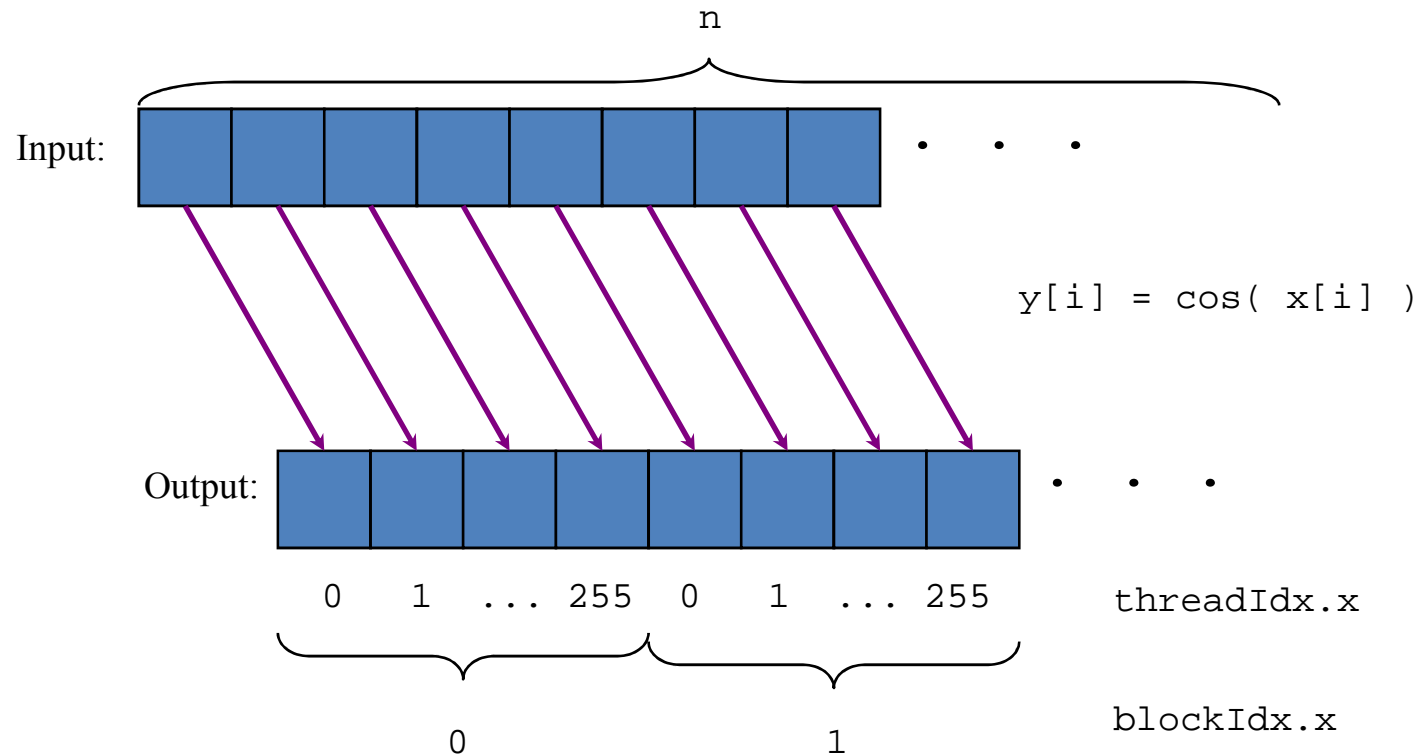
Grids/Blocks/Threads vs. Data Size

- ‘m*n’ threads being launched
 - == ‘grid.x*grid.y*block.x*block.y*block.z’ threads
- This may not match up with how much data you actually need to process
- You can turn threads (and blocks) “off”

```
__global__ void vcos( int n, float* x, float* y ) {  
    int ix = blockIdx.x*blockDim.x + threadIdx.x;  
    if( ix < n ) {  
        y[ix] = cos( x[ix] );  
    }  
}
```

```
__global__ void image_proc( int wd, int ht, float* x, float* y ) {  
    if( ((blockIdx.x*threadDim.x) < wd)  
        && ((blockIdx.y*threadDim.y) < ht) ) {  
        . . .  
    }  
}
```

Simple Example, cont'd



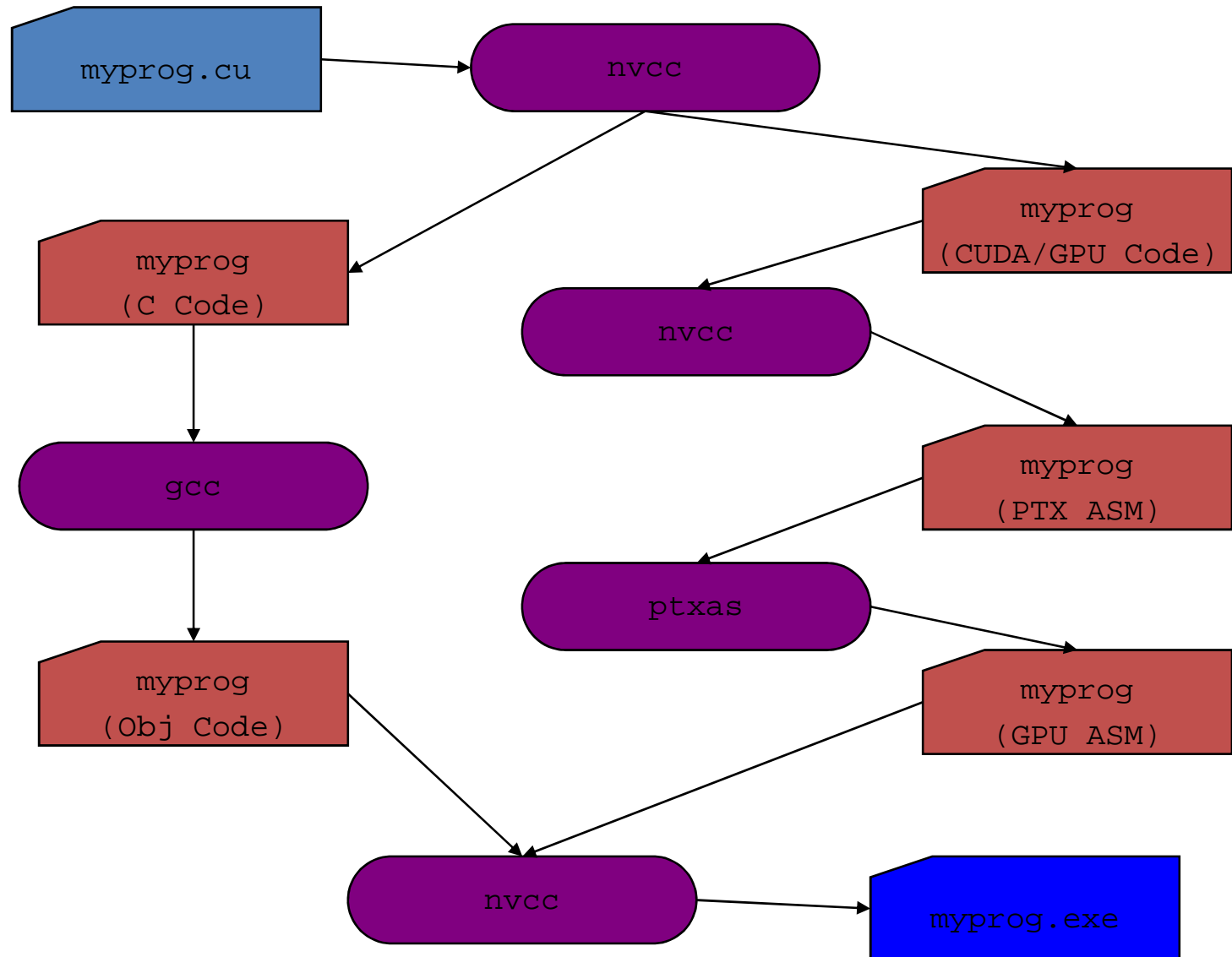
block numbers run 0 to $(n/256)$

Compilation

```
% nvcc -o simple simple.cu
```

- The compilation process is handled by the 'nvcc' wrapper
 - It splits out the CPU and GPU parts
 - The CPU parts are compiled with 'gcc'
 - The GPU parts are compiled with 'ptxas' (NV assembler)
 - The parts are stitched back together into one big object or executable file
 - Usual options also work
 - -I/include/path
 - -L/lib/path
 - -O

Compilation Details (-keep)



Compilation, cont'd

- -Xcompiler 'args'
 - For compiler-specific arguments
- -Xlinker 'args'
 - For linker-specific arguments
- --maxrregcount=16
 - Set the maximum per-GPU-thread register usage to 16
 - Useful for making “big” GPU functions smaller
 - Very important for performance ... more later!
- -Xptxax=-v
 - ‘verbose’ output from NV assembler
 - Gives register usage, shared-mem usage, etc.

Running a CUDA Program

- Just execute it! `% ./simple`
 - The CUDA program includes all the CPU-code and GPU-code inside it “fat binary”)
 - The CPU-code starts running as usual
 - The “run-time” (cudart) pushes all the GPU-code out to the GPU
 - This happens on the first CUDA function or GPU-launch
 - The run-time/display-driver control the mem-copy timing and sync
 - The run-time/display-driver “tell” the GPU to execute the GPU-code

Error Handling

- All CUDA functions return a 'cudaError_t' value
 - This is a 'typedef enum' in C ... '#include <cuda.h>'

```
cudaError_t err;  
err = cudaMemcpy( dev_x, host_x, nbytes, cudaMemcpyDeviceToHost );  
if( err != cudaSuccess ) {  
    /* something bad happened */  
    printf("Error: %s\n", cudaGetErrorString(err) );  
}
```

```
cudaError_t err;  
func_name<<<grd,blk>>>( arguments );  
err = cudaGetLastError();  
if( err != cudaSuccess ) {  
    /* something bad happened during launch */  
}
```

- Function launches do not directly report an error, but you can use:

Error Handling, cont'd

- Error handling is not as simple as you might think ...
- Since the GPU function-launch is async, only a few “bad things” can be caught immediately at launch-time:
 - Using features that your GPU does not support (double-precision?)
 - Too many blocks or threads
 - No CUDA-capable GPU found (pre-G80?)
- But some “bad things” cannot be caught until AFTER the launch:
 - Array overruns don't happen until the code actually executes; so the launch may be “good,” but the function crashes later
 - Division-by-Zero, NaN, Inf, etc.
 - MOST of your typical bugs CANNOT be caught at launch!

Error Handling, cont'd

```
func1<<<grd,blk>>>( arguments );  
err1 = cudaGetLastError();  
...  
err2 = cudaMemcpy( host_x, dev_x, nbytes, cudaMemcpyDeviceToHost );
```

- In this example, 'err2' could report an error from running func1, e.g. array-bounds overrun
 - Can be very confusing

```
func_name<<<grd,blk>>>( arguments );  
err1 = cudaGetLastError();  
err1b = cudaThreadSynchronize();  
...  
err2 = cudaMemcpy( host_x, dev_x, nbytes, cudaMemcpyDeviceToHost );
```

- 'err1b' now reports func1 run-time errors, 'err2' only reports memcpy errors

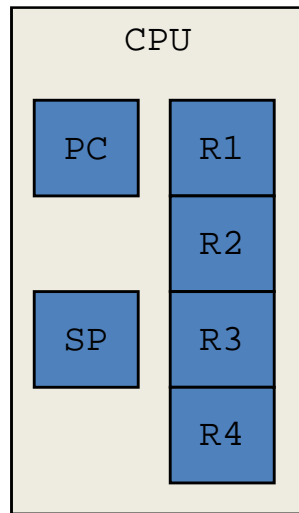
Error Handling, cont'd

- To get a human-readable error output:

```
err = cudaGetLastError();  
printf("Error: %s\n", cudaGetErrorString(err) );
```

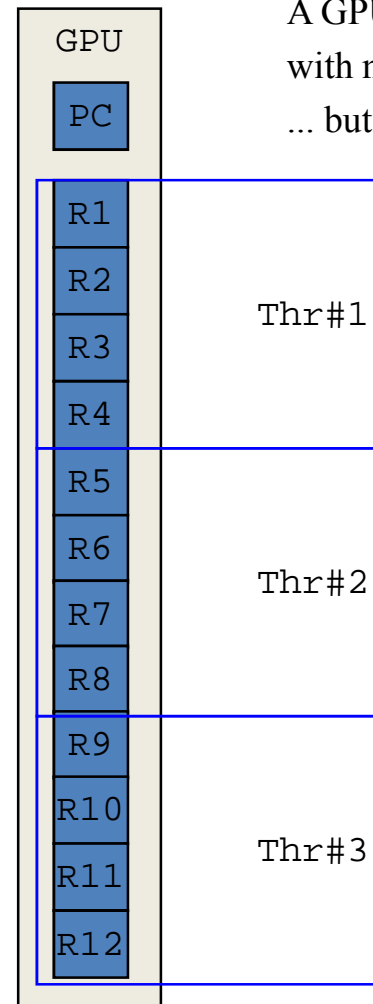
- NOTE: there are no “signaling NaNs” on the GPU
 - E.g. divide-by-zero in a GPU-thread is not an error that will halt the program, it just produces a Inf/NaN in the output and you have to detect that separately
 - $\text{Inf} + \text{number} \Rightarrow \text{Inf}$ $0 / \text{number} \Rightarrow \text{Inf}$
 - $\text{NaN} + \text{anything} \Rightarrow \text{NaN}$ $\text{Inf} - \text{Inf} \Rightarrow \text{NaN}$
 - $0/0$ or $\text{Inf}/\text{Inf} \Rightarrow \text{NaN}$ $0 * \text{Inf} \Rightarrow \text{NaN}$
 - Inf/NaN values tend to persist and propagate until all your data is screwed up
 - But the GPU will happily crank away on your program!

A Very Brief Overview of GPU Architecture



When a CPU thread runs, it “owns” the whole CPU.

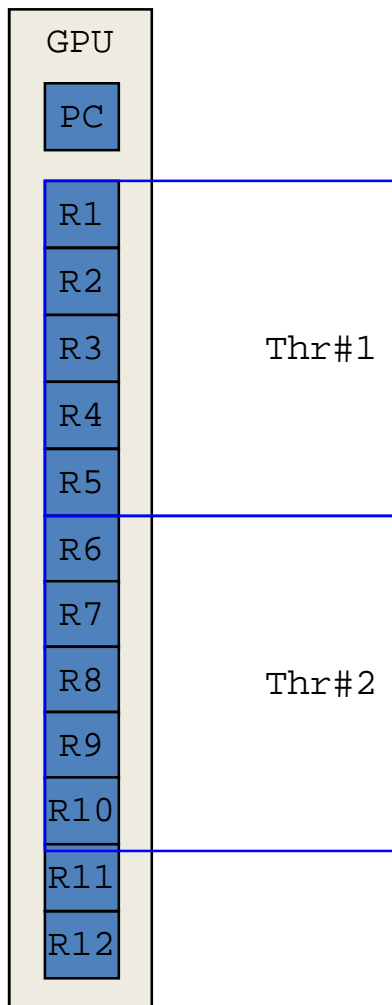
If more registers are needed, the compiler stores some register values to the stack and then reads them back later.



A GPU thread shares the GPU with many other threads ... but all share a Prog. Ctr.

Note: no stack pointer!

Register Usage



- If your algorithm is too complex, it may require additional registers for each thread
 - But that can reduce the number of threads that a given GPU-core can handle
- NVIDIA calls this “Occupancy”
 - The percent of the GPU resources that your threads are using
 - Other factors come into play:
 - Raw GPU capabilities & stats
 - Block-shared memory usage
 - Threads-per-block in your launch

Occupancy

- E.g. a G80 has 8192 registers per GPU-core
 - 8 registers per thread ... should fit 1024 threads per GPU-core
 - G80 can only simultaneously process 768 threads per GPU-core
 - The 1024 threads would be time-shared in batches of 768
 - 100% Occupancy
 - 16 registers per thread ... only 512 threads will fit on a GPU-core
 - 67% Occupancy!

Varying Register Use (G80):

10 reg	.. 128 th/bk	.. 6 bk/sm	.. 768 th/gpu	.. 100%
	.. 256 th/bk	.. 3 bk/sm	.. 768 th/gpu	.. 100%
12 reg	.. 128 th/bk	.. 5 bk/sm	.. 640 th/gpu	.. 83%
16 reg	.. 128 th/bk	.. 4 bk/sm	.. 512 th/gpu	.. 67%
	.. 256 th/bk	.. 2 bk/sm	.. 512 th/gpu	.. 67%
20 reg	.. 128 th/bk	.. 3 bk/sm	.. 384 th/gpu	.. 50%
32 reg	.. 128 th/bk	.. 2 bk/sm	.. 256 th/gpu	.. 33%
	.. 256 th/bk	.. 1 bk/sm	.. 256 th/gpu	.. 33%

--Xptxas=-v to see your usage
--maxrregcount=N to tweak

Occupancy and Grid/Block Sizes

- The general guidance is that you want “lots” of grid-blocks and “lots” of threads per block
 - Lots of blocks per grid means lots of independent parallel work
 - Helps to “future-proof” your code since future GPUs may be able to handle more grid-blocks simultaneously
 - Lots of threads per block means better GPU efficiency
 - Helps to keep math units and memory system “full”
- But how many is “lots”?

Grid/Block Sizes, cont'd

- 256 threads per block is a good starting point
 - Only blocks can effectively share memory
 - If you need some kind of memory synchronization in your code, this may drive your block size
 - Check occupancy table (following slide) and see how many blocks can fit in a GPU-core
 - High-end G80 == 16 GPU-cores
 - High-end G100 == 30 GPU-cores
- Then you can compute the number of blocks needed to fill your GPU
 - But you can have MORE blocks than fit in the GPU

Grid/Block Sizes, cont'd


- Future GPUs are likely to have more GPU-cores
- Future GPUs are likely to have more threads per core
- BUT a GPU-core can run multiple, smaller blocks simultaneously
 - So if you have many, smaller blocks, a “bigger” GPU-core could run 3 or 4 blocks and retain the efficiency of “lots” of threads
 - E.g. current generation GPU can handle 1 512-thread block per core
 - Next generation GPU might be capable of handling 1 1024-thread block per core, but could also handle 2 512-thread blocks per core
 - So your “old” code will still work efficiently
 - Err on the side of more blocks per grid, with a reasonable number of threads per block (128 min, 256 is better)
- GPUs are rapidly evolving so while future-proofing your code is nice, it might not be worth spending too much time and effort on

Some Examples

```
__global__ void func( int n, float* x ) {  
    int ix = blockIdx.x*blockDim.x + threadIdx.x;  
    x[ix] = 0.0f;  
}  
nblk = size/nthr;  
func<<<nblk,nthr>>>( size, x );
```

```
#define BLK_SZ (256)  
__global__ void func( int n, float* x ) {  
    int ix = 4*(blockIdx.x*BLK_SZ + threadIdx.x);  
    x[ix]          = 0.0f;  
    x[ix+BLK_SZ]   = 0.0f;  
    x[ix+2*BLK_SZ] = 0.0f;  
    x[ix+3*BLK_SZ] = 0.0f;  
}  
nb = size/(4*BLK_SZ);  
func<<<nb,BLK_SZ>>>( size, x );
```

Be careful with integer division!



Some More Examples

```
__global__ void func( int n, float* x ) {  
    int i, ix = blockIdx.x*blockDim.x + threadIdx.x;  
    for(i=ix; i<n; i+=blockDim.x*gridDim.x) {  
        x[i] = 0.0f;  
    }  
}  
func<<<m,n>>>>( size, x );
```

```
#define GRD_SZ (32)  
#define BLK_SZ (256)  
__global__ void func( int n, float* x ) {  
    int i, ix = blockIdx.x*BLK_SZ + threadIdx.x;  
    for(i=ix; i<n; i+=BLK_SZ*GRD_SZ) {  
        x[i] = 0.0f;  
    }  
}  
func<<<GRD_SZ, BLK_SZ>>>>( size, x );
```

Performance Issues

- Hard-coding your grid/block sizes can help reduce register usage

```
#define BLK_SZ (256)
```

- E.g. BLK_SZ (vs. blockDim) is then encoded directly into the instruction stream, not stored in a register
- Choosing the number of grid-blocks based on problem size can essentially “unroll” your outer loop ... which can improve efficiency and reduce register count
 - E.g. nblks = (size/nthreads)
 - You may want each thread to handle more work, e.g. 4 data elements per thread, for better thread-level efficiency (less loop overhead)
 - That may reduce the number of blocks you need

Performance Issues, cont'd

- Consider writing several different variations of the function where each variation handles a different range of sizes, and hard-codes a different grid/block/launch configuration
 - E.g. small, medium, large problem sizes
 - ‘small’ ... (size/256) blocks of 256 threads ... maybe not-so-efficient, but for small problems, it’s good enough
 - ‘medium’ ... 32 blocks of 256 threads
 - ‘large’ ... 32 blocks of 256 threads with 4 data elements per thread
 - It might be worth picking out special-case sizes (powers-of-2 or multiples of blockDim) which allow you to keep warps together or allow for fixed-length loops, etc.
 - Some CUBLAS functions have 1024 sub-functions
 - There is some amazing C-macro programming in the CUBLAS, take a look at the (open-)source code!

Performance Measurement ...

CUDA_PROFILE

```
% setenv CUDA_PROFILE 1  
% ./simple  
% cat cuda_profile.log
```

- Turning on the profiler will produce a log file with all the GPU-function launches and memory transfers recorded in it
- Also reports GPU occupancy for GPU-function launches
- There is now a “visual” CUDA Profiler as well

Asynchronous Launches

- When your program executes 'vcos<<<m,n>>>', it launches the GPU-threads and then IMMEDIATELY returns to your (CPU) program
 - So you can have the CPU do other work WHILE the GPU is computing 'vcos'
- If you want to wait for the GPU to complete before doing any other work on the CPU, you need to explicitly synchronize the two:

```
vcos<<<m,n>>>( n, dev_x, dev_y );  
cudaThreadSynchronize();  
/* do other CPU work */
```

- Note that 'cudaMemcpy' automatically does a synchronization, so you do NOT have to worry about copying back bad data

Async Launches, cont'd

- With more modern GPUs (G90?, G100?), you can potentially overlap GPU-memory transfers and GPU-function computations:

```
/* read data from disk into x1 */  
cudaMemcpy( dev_x1, host_x1, nbytes, cudaMemcpyHostToDevice );  
func1<<<m,n>>>( dev_x1 );  
/* read data from disk into x2 */  
cudaMemcpy( dev_x2, host_x2, nbytes, cudaMemcpyHostToDevice );  
func2<<<m,n>>>( dev_x2 );
```

- File-read of x2 should happen WHILE func1 is running
 - Data transfer for x2 may happen WHILE func1 is running
- Synchronizing all of this gets complicated
 - See cudaEvent and cudaStream functions

Block-Shared Memory

- CUDA assumes a GPU with block-shared as well as program-shared memory
 - A thread-block can communicate through block-shared memory
 - Limited resource: ~16KB per block
 - I.e. all threads in Block (1,0,0) see the same data, but cannot see Block (1,1,0)'s data
 - Main memory is shared by all grids/blocks/threads
 - NOTE: main memory is `_not_` guaranteed to be consistent (at least not right away)
 - NOTE: main memory writes may not complete in-order
 - Need a newer GPU (G90 or G100) to do “atomic” operations on main memory

Block-Shared Memory, cont'd

```
__shared__ float tmp_x[256];
__global__ void partial_sums( int n, float* x, float* y ) {
    int i, ix = blockIdx.x*blockDim.x + threadIdx.x;
    tmp_x[threadIdx.x] = x[ix];
    __syncthreads();
    for(i=0; i<threadIdx.x; i++) {
        y[ix] = tmp_x[i];
    }
}
```

- Block-shared memory is not immediately synchronized
 - You must call `__syncthreads()` before you read the data
- There is often a linkage between the `__shared__` memory size and the block-size (`blockDim.x`)
 - Be careful that these match or that you don't overrun the `__shared__` array bounds

Block-Shared Memory, cont'd

- Since block-shared memory is so limited in size, you often need to “chunk” your data
 - Make sure to check your array bounds
 - Make sure you `__syncthreads` every time new data is read into the `__shared__` array
- You can specify the size of the shared array at launch-time:

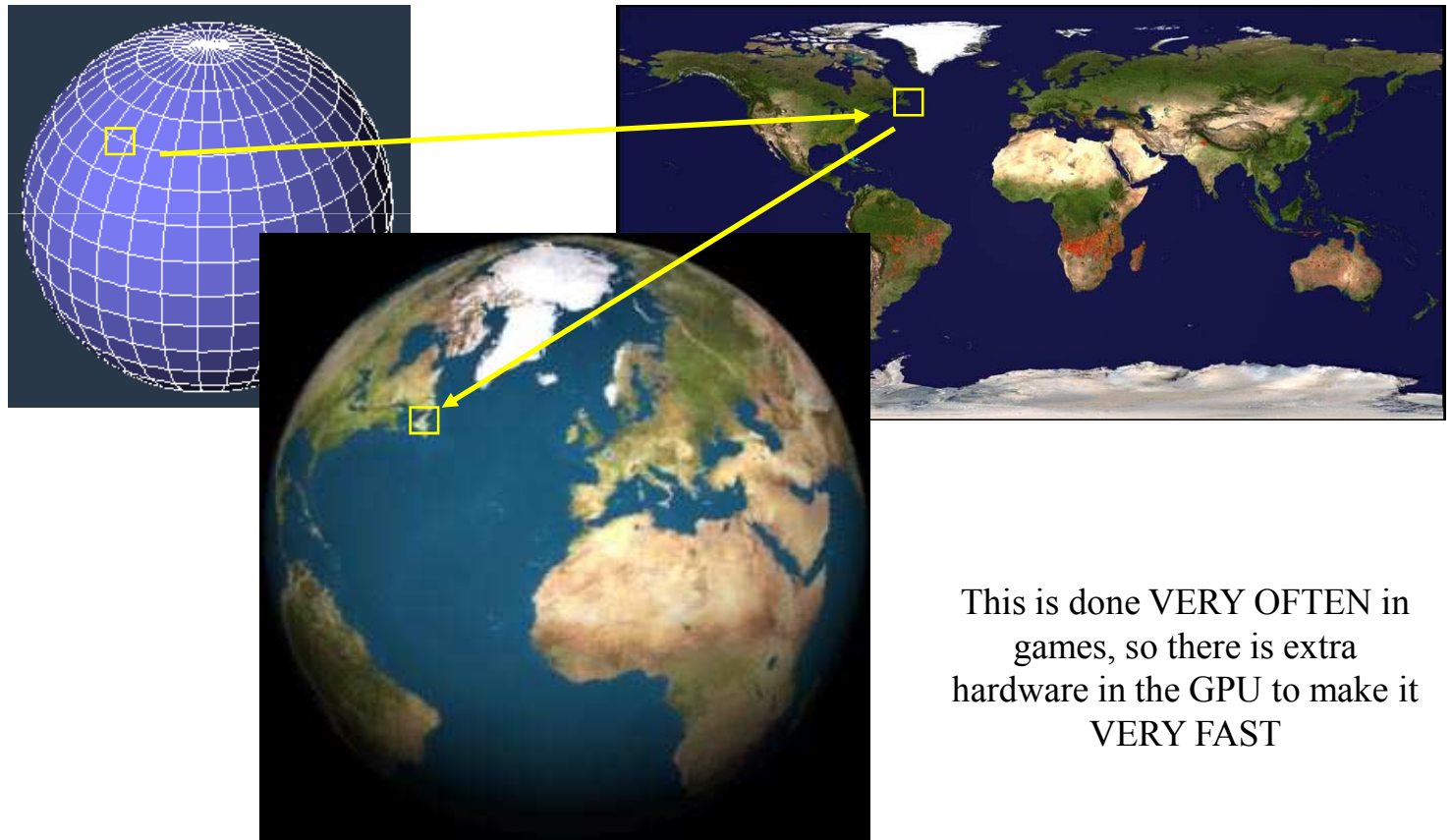
```
__shared__ float* tmp_x;
__global__ void partial_sums( int n, float* x, float* y ) {
    . . .
}
int main() {
    . . .
    partial_sums<<<m,n,1024>>>( n, x, y );
    . . .
}
```



size in bytes

Texture References

- “Texrefs” are used to map a 2-D “skin” onto a 3-D polygonal model
 - In games, this allows a low-res (fast) game object to appear to have more complexity



This is done VERY OFTEN in games, so there is extra hardware in the GPU to make it VERY FAST

Texture References, cont'd

- A texref is just an irregular, cached memory access system
 - We can use this if we know (or suspect) that our memory references will not be uniform or strided

```
texture<float> texX;
```

```
__global__ void func( int N, float* x, ... ) {
```

```
    ...
```

```
    for(i=0;i<N;i++) {
```

```
        sum += tex1Dfetch( texX, i );
```

```
    }
```

```
    ...
```

```
    return;
```

```
}
```

```
main() {
```

```
    ...
```

```
    err = cudaBindTexture( &texXofs, texX, x, N*sizeof(float) );
```

```
    ...
```

```
    func<<<grd,blk>>>>( N, x, ... );
```

```
    ...
```

```
    err = cudaUnbindTexture( texXofs );
```

```
    ...
```

```
}
```

Gets the i-th value in X

Returns an offset (usually 0)

Texture References, cont'd

- Textures are a limited resource, so you should bind/unbind them as you need them
 - If you only use one, maybe you can leave it bound all the time
- Strided memory accesses are generally FASTER than textures
 - But it is easy enough to experiment with/without textures, so give it a try if you are not certain
- `__shared__` memory accesses are generally FASTER than textures
 - So if data will be re-used multiple times, consider `__shared__` instead

SIMD and “Warps”

- The GPU really has several program-counters, each one controls a group of threads
 - All threads in a group must execute the same machine instruction
 - For stream computing, this is the usual case
 - What about conditionals?

```
__global__ void func( float* x ) {  
    if( threadIdx.x >= 8 ) {  
        /* codeblock-1 */  
    } else {  
        /* codeblock-2 */  
    }  
}
```

- All threads, even those who fail the conditional, walk through codeblock-1 ... the failing threads just “sleep” or go idle
 - When code-block-2 is run, the other set of threads “sleep” or go idle

Conditionals

- Generally, conditionals on some `F(threadIdx)` are bad for performance
 - This means that some threads will be idle (not doing work) at least some of the time
 - Unless you can guarantee that the conditional keeps Warps together
 - Presently a warp is a set of 32 threads
- Conditionals on `F(blockIdx)` are fine
- Be careful with loop bounds

```
for(i=0;i<threadIdx.x;i++) {  
    /* codeblock-3 */  
}
```

 - The end-clause is just a conditional

Tuning Performance to a Specific GPU

- What kind of GPU am I running on?

```
cudaGetDeviceProperties( dev_num, &props );  
if( props.major < 1 ) {  
    /* not CUDA-capable */  
}
```

- structure returns fields ‘major’ and ‘minor’ numbers
 - major=1 ... CUDA-capable GPU
 - minor=0 ... G80 ... 768 threads per SM
 - minor=1 ... G90 ... 768 threads per SM, atomic ops
 - minor=3 ... G100 ... 1024 threads per SM, double-precision
- props.multiProcessorCount contains the number of SMs
 - GeForce 8600GT ... G80 chip with 4 SMs
 - GeForce 8800GTX ... G80 chip with 16 SMs
 - GeForce 8800GT ... G90 chip with 14 SMs
 - See CUDA Programming Guide, Appendix A

Memory Performance Issues

- For regular memory accesses, have thread-groups read consecutive locations

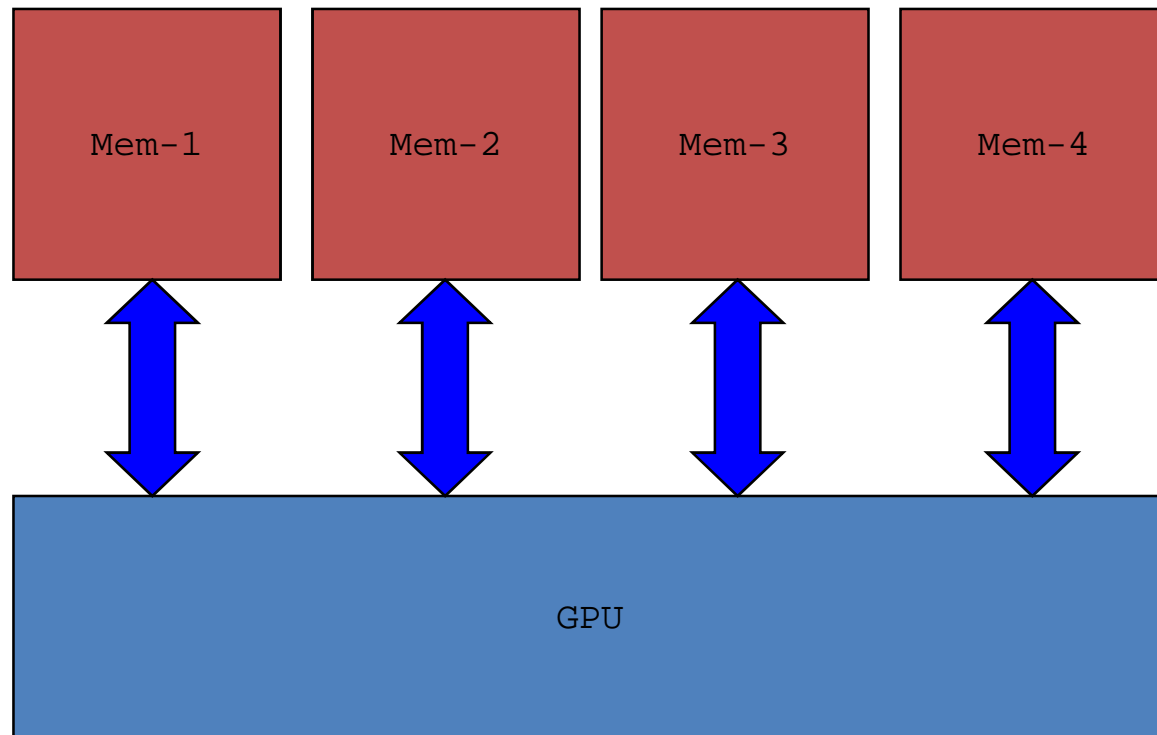
- E.g. thread-0 reads x[0] while thread-1 reads x[1]; then thread-0 reads x[128] while thread-1 reads x[129]

```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
int ttl_nthreads = blockDim.x*blockDim.x;
for(i=idx; i<N; i+=ttl_nthreads) {
    z[i] = x[i] + y[i];
}
```

- Don't have thread-0 touch x[0], x[1], x[2], ..., while thread-1 touches x[128], x[129], x[130], ...
- The GPU executes can execute thread-0/-1/-2/-3 all at once
- And the GPU memory system can fetch x[0],x[1],x[2],x[3] all at once

Memory Performance Issues, cont'd

- GPU memory is “banked”
 - Hard to classify which GPU-products have what banking



Multi-GPU Programming

- If one is good, four must be better!!
 - S870 system packs 4 G80s into an external box (external power)
 - S1070 packs 4 G100s into an external box
 - 9800GX2 is 2 G90s on a single PCI card
- Basic approach is to spawn multiple CPU-threads, one per GPU
 - Each CPU-thread then attaches to a separate GPU

```
cudaSetDevice( n );
```
 - CUDA will time-share the GPUs, so if you don't explicitly set the device, the program will still run (just very slowly)
 - There is no GPU-to-GPU synchronization in CUDA
 - So you must have each CPU-thread sync to its GPU, then have the CPU-threads sync through, e.g. `pthread_barrier`, then have the CPU-threads release a new batch of threads onto their GPUs