Intro to CUDA Programming

http://www.oit.duke.edu/scsc
http://wiki.duke.edu/display/SCSC
scsc@duke.edu
hpc-support@duke.edu

John Pormann, Ph.D.
jbp1@duke.edu

Overview

- Basic Introduction
- 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

CUDA and NVIDIA

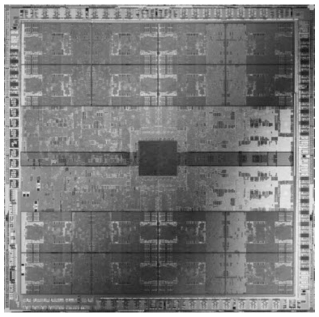
- CUDA is an NVIDIA product and only runs on NVIDIA GPUs
  - AMD/ATI graphics chips will NOT run CUDA
  - Older NVIDIA GPUs may not run CUDA either
  - *Some* laptops may be capable of running CUDA
    - Not sure what this will do to battery life
  - All current and future display drivers from NVIDIA will include support for CUDA
    - You don't need to download anything else to run a CUDA program
  - To see if your GPU is CUDA-enabled, go to:
    - http://www.nvidia.com/object/cuda_learn_products.html

Why GPU programming?

- Parallelism
  - CPUs recently moved to dual- and quad-core chips
  - The current G100 GPU has 240 cores
- Memory bandwidth
  - CPU (DDR-400) memory can go 3.2GB/sec
  - GPU memory system can go 141.7GB/sec
- Speed
  - CPUs can reach 20GFLOPS (per core)
  - GPUs can reach 933GFLOPS (single-precision or integer)
  - ... 78GFLOPS (double-precision)
- Cost ... $400-1000

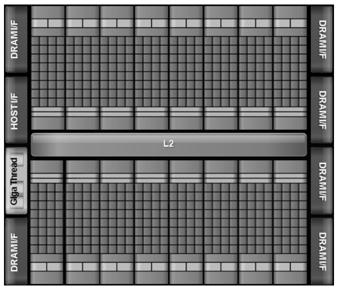
Yesterday’s Announcement

- NVIDIA recently held their annual developer conference and released info on the next generation of GPUs ... “Fermi”



3B transistors, 40nm

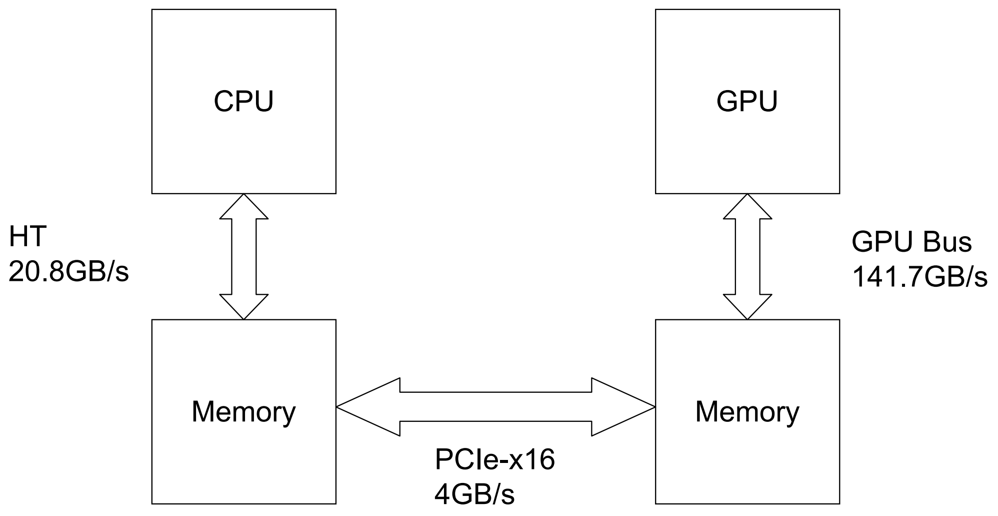
512 compute elements
8x increase in DP performance (~700GFLOPS)
GDDR5 memory (230GB/sec)
ECC memory
L1 and L2 Cache memory (“configurable”?)



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 100th time-step

```
Transfer data to GPU
Loop:
   Run the code on the GPU
   Compute error on the GPU
   Transfer error to CPU
   If error > tol, continue
Transfer data to CPU
```

```
Transfer data to GPU
For t=1 to 1000000:
   Run the code on the GPU
   If (t%100)==0, transfer
      data to CPU
   Print/save data on CPU
Transfer data to 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

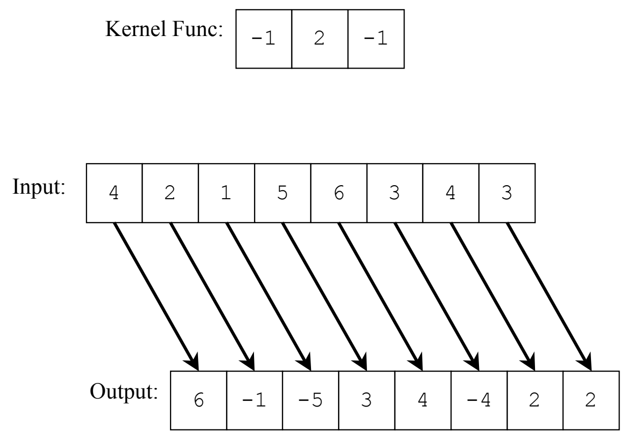
```
cudaMemcpy( dev_x, host_x, n*sizeof(float), cudaMemcpyHostToDevice );

. . .

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

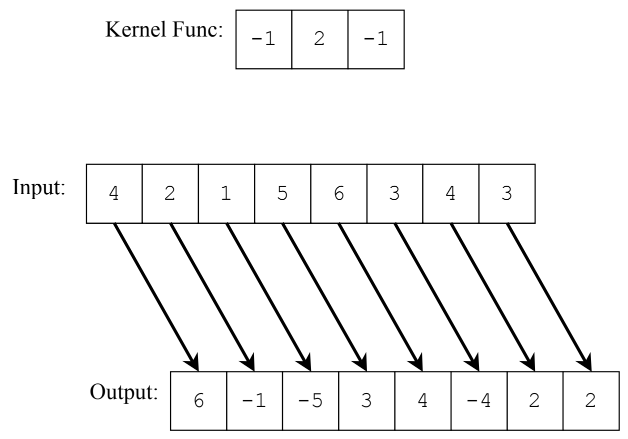
- GPUs are multi-threaded computational engines
  - They can execute hundreds of threads simultaneously, and can keep track of thousands of pending threads
    - Note that GPU-threads are expected to be short-lived, you should not program them to run for hours continuously
  - With thousands of threads, 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 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



Parallel (4-way) computation ... 2 clock-ticks
... NVIDIA G100 has 240-way parallelism !!

CPU Threads vs. GPU Threads

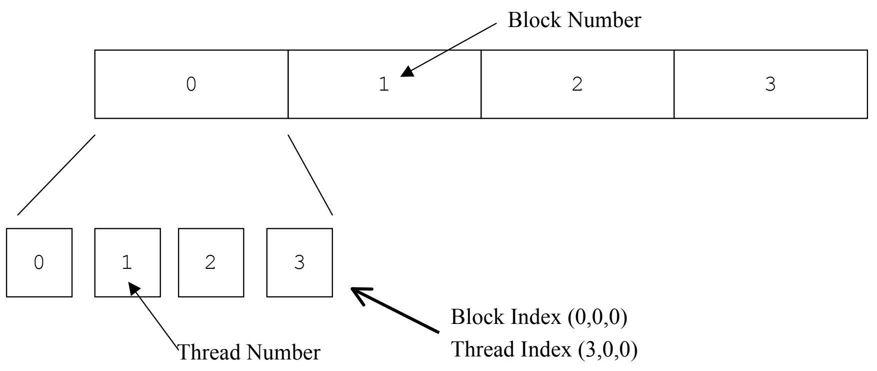
- CPU Threads (POSIX Threads) are generally considered long-lived computational entities
  - You fork 1 CPU-thread per CPU-core in your system, and you keep them alive for the duration of your program
  - CPU-thread creation can take several uSec or mSec -- you need to do a lot of operations to amortize the start-up cost
- GPU Threads are generally short-lived
  - You fork 1000’s of GPU-threads, and they do a small amount of computation before exiting
  - GPU-thread creation is generally very fast -- you can create 1000’s of them in a few ticks of the clock

GPU Task/Thread Model

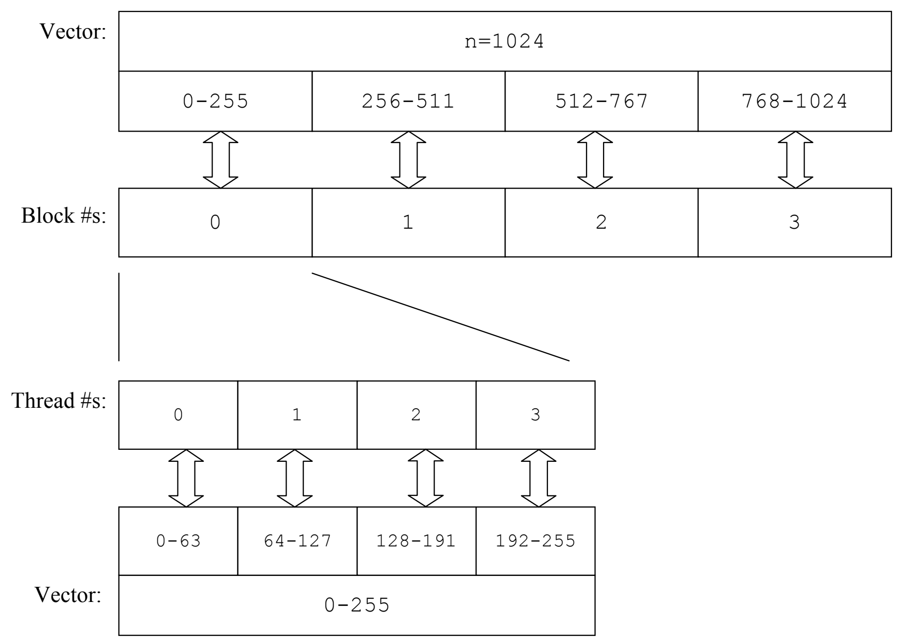
- We don't launch *A* thread onto a GPU, we launch hundreds or thousands threads all at once
  - The GPU hardware will handle how to run/manage them
- In CUDA, we launch a "grid" of "blocks" of "threads" onto a GPU
  - Grid = 1- or 2-D (eventually 3-D) config of a given size
    - Grid dims <= 65536
  - Block = 1-,2-,3-D config of a given size
    - Block dims <= 512, total <= 768 threads ← NVIDIA G100: 1024
  - The GPU program (each thread) must know how to configure itself using only these two sets of coordinates
    - Similar to MPI's MPI_Comm_rank and MPI_Comm_size

1-D x 1-D Example

- 1-D Grid ... 4 (or 4x1x1)
- 1-D Blocks ... 4 (or 4x1x1)

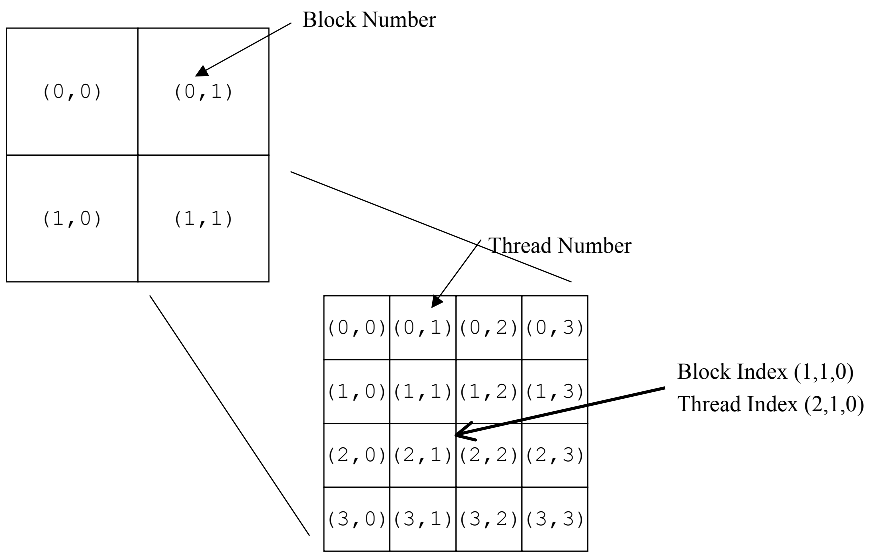


1-D x 1-D Example, cont'd

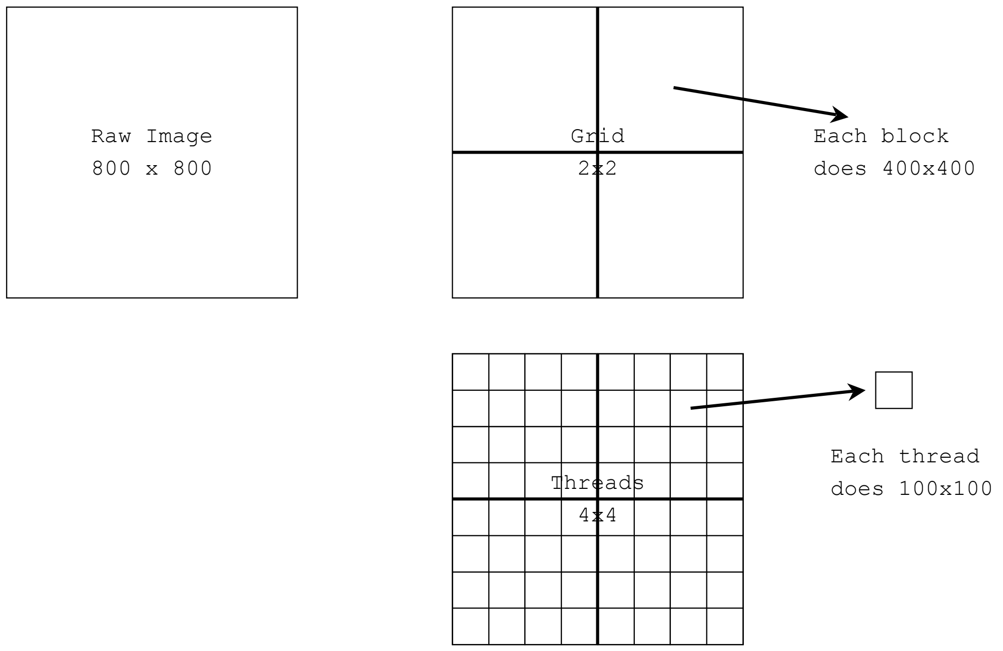


2-D x 2-D Example

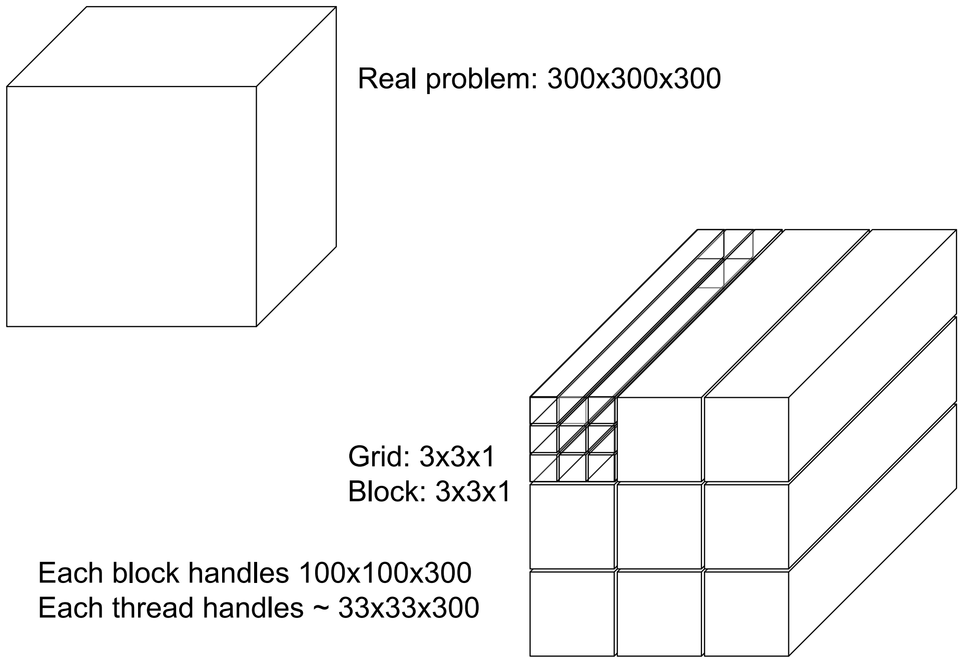
- 2-D Grid ... 2x2x1
- 2-D Blocks ... 4x4x1



2-D x 2-D Example, cont'd

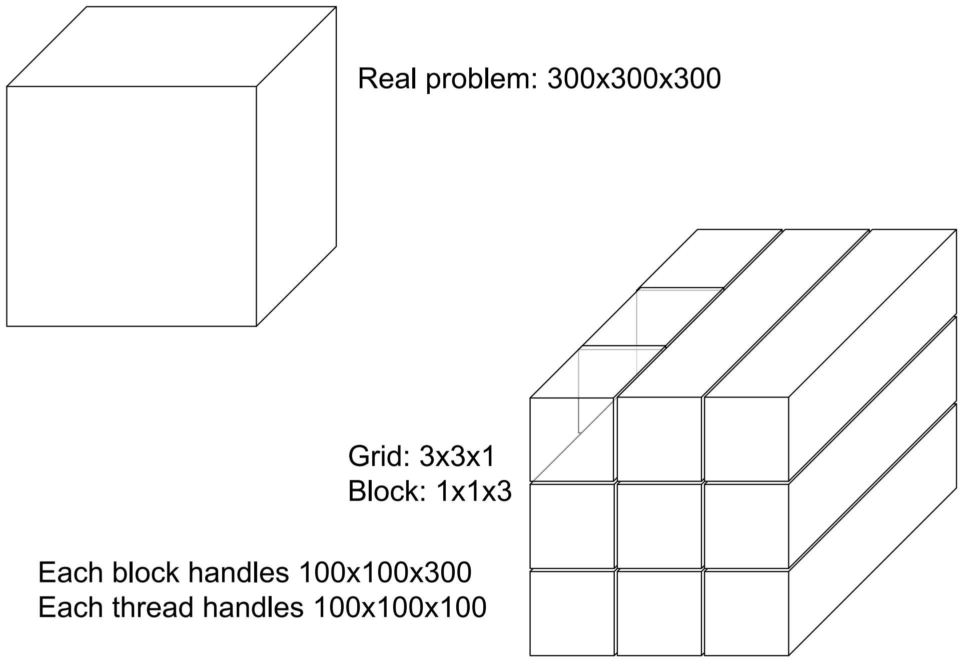


CUDA Grid Example



Each block handles 100x100x300
Each thread handles ~ 33x33x300

CUDA Grid Example, cont’d



Each block handles 100x100x300
Each thread handles 100x100x100

Simple Example (again)

```
__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 );
}
```

Returning to the Simple Example

the __global__ function name

```
/* 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}

also available: {blockDim.x,gridDim.x}

  - ‘m’ and ‘n’ can also be ‘uint3’ (3-D) objects

```
uint3 m,n;                              uint3 m,n;
m = make_uint3(128,128,1);              m.x=128; m.y=128; m.z=1;
n = make_uint3(32,32,1);                n.x=32; n.y=32; n.z=1;

          vcos<<<m,n>>>( n, dev_x, dev_y );
```

now launching m.x*m.y*m.z*n.x*n.y*n.z threads (but not necessarily simultaneous)

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] );
}
```

- ‘int ix’ is the global index number for this thread’s calculations
  - We compute it from the built-in, thread-specific variables (set by the run-time environment)
    - Each GPU-thread will have a unique combination of

      `{blockIdx.x,threadIdx.x}`
    - So each GPU-thread will also have a unique ‘ix’ value
      - It is up to YOU to make sure that all data is processed (i.e. that all valid ‘ix’ values are hit)

```
__global__ void vcos( int n, float* x, float* y ) {
    int i;
    int ix0 = blockIdx.x*blockDim.x + 64*threadIdx.x;
    for(i=0;i<64;i++) {
        y[i+ix0] = cos( x[i+ix0] );
    }
}
```

64 vector-elements per thread

Grids/Blocks/Threads vs. Data Size

- The way the launch process works you end up with ‘m*n’ threads being launched
    - or ‘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” by letting them exit the GPU-function

```
__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*blockDim.x+threadIdx.x) < wd)
        && ((blockIdx.y*blockDim.y+threadIdx.y) < ht) ) {
        . . .
    }
}
```

__global__ Functions

- Note that __global__ functions must return type ‘void’ ... that is, they do not return a value
  - If your function encounters an error, you must provide that error/return value some other way
  - There are ways of detecting if a function could not be launched, or other “CUDA errors” -- but “user-defined errors” must be sent back through some other means
    - And note that you can’t send a __global__ function a CPU-pointer!
      - So you have to save the error/return code to GPU-memory, then do a mem-copy
      - Watch out for race conditions if all threads write to same error/return code area

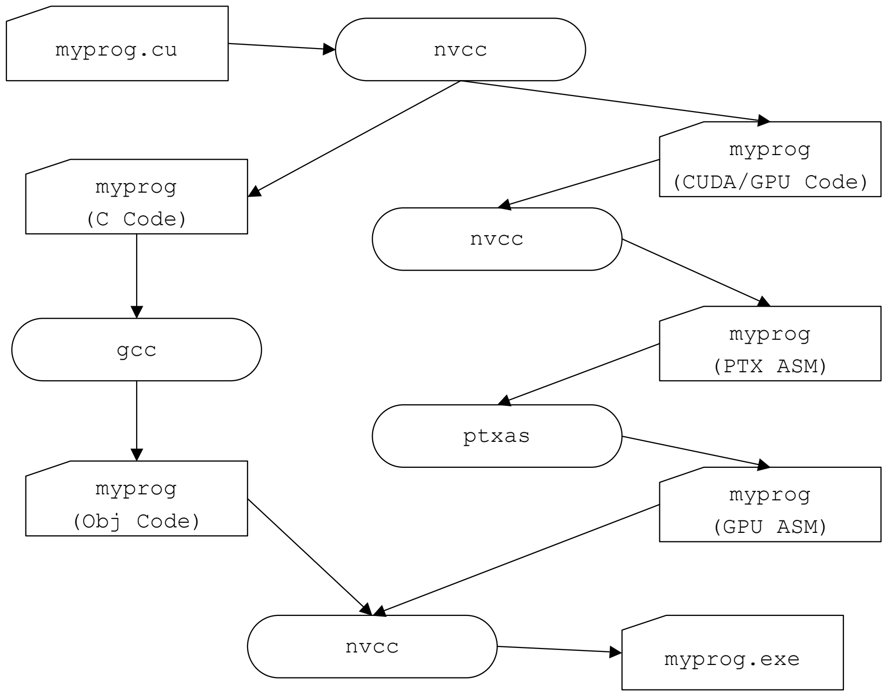
Compilation

```
% 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 (nvcc -keep)



Compilation Details, 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!
- -Xptxas=-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 (“fatbin” or “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) );
}
```

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

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

Error Handling, cont'd

- Error handling is not as simple as you might think ...
- The GPU function-launch is async, so the launch "returns" to the CPU immediately, even though the GPU code has not finished executing
  - So 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:

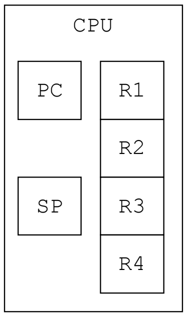
```
err = cudaGetLastError();
printf("Error: %s\n", cudaGetErrorString(err) );
```

- 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 in the output and you have to detect that separately
    - Inf + number => Inf number / 0 => Inf
    - NaN + anything => NaN Inf - Inf => NaN
    - 0/0 or Inf/Inf => NaN 0 * Inf => 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!

DON'T DESPAIR !!

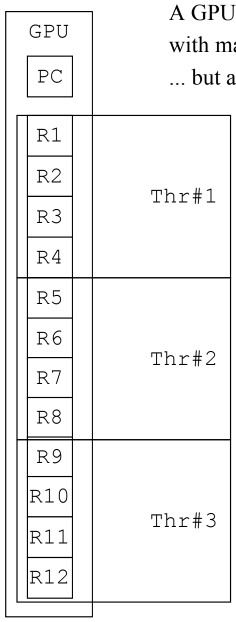
- Performance tuning on GPUs is definitely a black art
  - Grid size, Block size, GPU "size", registers per thread, occupancy, computational density, loop overheads, if/then statements, memory access pattern, shared memory, texture references
    - ALL impact performance
    - Some of these are "low-order bits" and can often be ignored
- Keep in mind that you're starting with 1TFLOPS of performance
  - If you hit 50% efficiency, that's still not too bad

A Very Brief Overview of GPU Architecture



When a CPU thread runs, it "owns" the whole CPU.

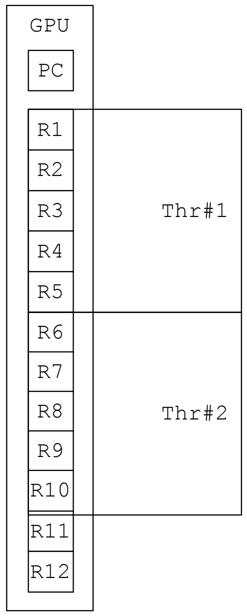
If more registers are needed, the compiler stores some register values to the stack and then reads them back later.



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

- Real GPU-cores have 8192 (now 16384) registers as well as 768 (now 1024) thread "place-holders"
  - So you can be working on 768 threads simultaneously
  - But only if you can fit 768 threads in the register set

G80 ... 8K registers
G100 ... 16K registers
Fermi ... 32K registers

Thread Usage and Performance

- You can consider the GPU-core to have 768 adder/multiplier units
  - So if you launch 768 threads, you should be using 100% of the GPU's computational power
    - A block size of 48x16, 32x24, 128x6, 256x3 would "fill" a GPU-core
      - Note: only 512 threads per dimension, so 768x1 is not possible
    - ... but a block size of 16x16 would potentially use only 33% of the computational units

  - However, you can launch more than one block onto a GPU-core
    - In fact, the GPU will automatically launch 3 16x16 blocks, simultaneously, onto each GPU-core
      - 100% of the computational units would be used
      - 18x18 .. 324 threads/block .. 2 per GPU .. 84% utilization

    - Note: you cannot run 1 block across 2 GPU-cores

This is BEST CASE utilization

Register Usage and Performance

- If your GPU code uses 20 registers per thread, then you can only fit 409 threads per GPU-core ... which is 1 block per GPU-core
  - You could launch blocks of 20x20, 16x25, 100x4, etc.
  - At best, you could use 53% of the adders/multiliers
    - You'll never be able to use 100% of peak performance
  - E.g. 12 registers per thread and 256 threads per block
    - Each block requires 3072 registers ... so 2 blocks per GPU-core
    - But 2 blocks is only 512 running threads ... we COULD do 768
      - 67% Occupancy
  - We could adjust the threads-per-block
    - 128 threads per block ... 5 blocks per GPU-core ... 83%
  - We could re-compile with '--maxrregcount=10'
    - 2560 registers ... 3 blocks per GPU-core ... 100% utilization

Compile with '-Xptxas=-v' to see your register usage

Occupancy, cont'd

- GeForce-8 has 8192 registers, and 768 simultaneous threads

```
Varying Register Use (GF8, GF9):
10 reg .. 128 th/bk .. 6 bk/core .. 768 th/core .. 100%
       .. 256 th/bk .. 3 bk/core .. 768 th/core .. 100%
12 reg .. 128 th/bk .. 5 bk/core .. 640 th/core .. 83%
16 reg .. 128 th/bk .. 4 bk/core .. 512 th/core .. 67%
       .. 256 th/bk .. 2 bk/core .. 512 th/core .. 67%
20 reg .. 128 th/bk .. 3 bk/core .. 384 th/core .. 50%
32 reg .. 128 th/bk .. 2 bk/core .. 256 th/core .. 33%
       .. 256 th/bk .. 1 bk/core .. 256 th/core .. 33%
```

- G100 has 16384 registers, and 1024 simultaneous threads

```
Varying Register Use (G100):
16 reg .. 128 th/bk .. 8 bk/core .. 1024 th/core .. 100%
       .. 256 th/bk .. 4 bk/core .. 1024 th/core .. 100%
       .. 512 th/bk .. 2 bk/core .. 1024 th/core .. 100%
18 reg .. 128 th/bk .. 6 bk/core ..  768 th/core ..  75%
       .. 256 th/bk .. 3 bk/core ..  768 th/core ..  75%
22 reg .. 128 th/bk .. 5 bk/core ..  640 th/core ..  63%
       .. 256 th/bk .. 2 bk/core ..  512 th/core ..  50%
       .. 512 th/bk .. 2 bk/core ..  512 th/core ..  50%
26 reg .. 128 th/bk .. 4 bk/core ..  512 th/core ..  50%
34 reg .. 128 th/bk .. 3 bk/core ..  384 th/core ..  33%
       .. 256 th/bk .. 1 bk/core ..  256 th/core ..  25%
```

Grid Size

- The general guidance is that you want “lots” of grid-blocks
  - Lots of blocks per grid means lots of independent parallel work
- Helps to “future-proof” your code since future GPUs will be able to handle more grid-blocks simultaneously
  - GeForce-8 has up to 16 GPU-cores
    - E.g. 10 reg/thread, 256 thr/blk, 3 blk/core ... minimum of 48 blocks
  - G100 has up to 30 GPU-cores
    - E.g. 10 reg/thread, 256 thr/blk, 8 blk/core ... minimum of 240 blocks!

Fermi has up to 512 GPU-cores

  - Note that if you decrease the threads-per-block, you may increase the number of blocks needed to do the work ... but you also increase the number of blocks-per-core

Grid and Block Sizes

- 256 Threads per block is a good starting point
  - See what your register usage is and what the occupancy is
- Reduce the amount of work per thread inside the __global__ functions so that more blocks are needed
  - E.g. don’t have 1 thread do 64 array entries
  - E.g. 1 thread may just update 1 pixel in the output image
    - Don’t forget: GPU-thread creation is very fast
- However, if the per-thread work gets too small, then there can be other basic performance limiters
  - To read an array entry, we first read the pointer-x, then calculate x+ix*4 (1 mult and 1 add), then we can finally read x[ix]
    - Once we’ve done all that, we can easily read x[ix+1] by just adding 4 to the new pointer

Grid and Block Sizes, cont'd

- Future GPUs are likely to have more GPU-cores
- Future GPUs are likely to have more threads per core

- 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
  - CUDA is only on v.2.3 and yet it supports 4 versions of GPUs, and dozens of graphics products

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 ... GeForce-8 ... 768 threads per core
    - minor=1 ... GeForce-9 ... 768 threads per core, atomic ops
    - minor=3 ... G100 ... 1024 threads per core, double-precision
  - field 'multiProcessorCount' contains the number of GPU-cores
    - GeForce 8600GT ... GF8 chip with 4 cores
    - GeForce 8800GTX ... GF8 chip with 16 cores
    - GeForce 8800GT ... GF9 chip with 14 cores
      - See CUDA Programming Guide, Appendix A

Some Examples

```
__global__ void func( int n, float* x ) {
   int ix = blockIdx.x*blockDim.x + threadIdx.x;
   x[ix] = 0.0f;
}
nblk = size/256;
func<<<nblk,256>>>( 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;
}
nblk = size/(4*BLK_SZ);
func<<<nblk,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<<<48,256>>>( size, x );
```

```
#define GRD_SZ (48)
#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 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
  - Note that if a GPU function is called inside an “inner loop”, you’ll get lots and lots of output!
- Also reports GPU occupancy for GPU-function launches
- There is now a “visual” CUDA Profiler as well

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

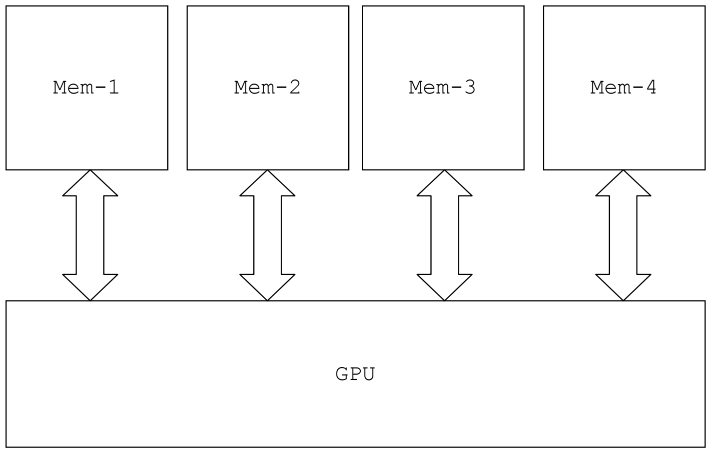
- Consider writing several different variations of the function where each variation handles a different range of sizes, and hard-codes a different grid/block/launch configuration
  - E.g. small, medium, large problem sizes
    - 'small' ... (size/256) blocks of 256 threads ... maybe not-so-efficient, but for small problems, it's good enough
    - 'medium' ... 48 blocks of 256 threads
    - 'large' ... 48 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) ... might allow for fixed-length loops
  - 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!

Main Memory-based "Communication"

- Technically, main memory is shared by all grids/blocks/threads
  - BUT: main memory is _not_ guaranteed to be consistent (at least not right away)
  - BUT: main memory writes may not complete in-order
  - Newer GPU (GF9 or G100) can do "atomic" operations on main memory ... but they essentially lock-out all other threads while they do their atomic operation (could be bad for performance)

Memory Performance Issues

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



Memory Performance Issues, cont’d

- For regular memory accesses, you want to have threads read consecutive memory (or array) 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]

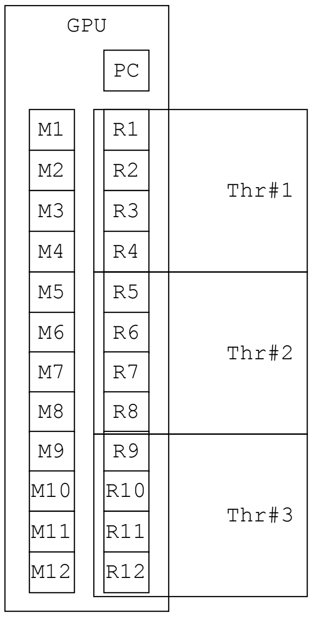
```
int idx = blockIdx.x*blockDim.x + threadIdx.x;
int ttl_nthreads = gridDim.x*blockDim.x;
for(i=idx;i<N;i+=ttl_nthreads) {
   z[i] = x[i] + y[i];
}
```

  - Don’t have thread-0 touch x[0], x[1], x[2], ..., while thread-1 touches x[64], x[65], x[66], ...

  - 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

Block-Shared Memory

- CUDA assumes a GPU with block-shared as well as program-shared memory
  - Threads in the same block can communicate through this shared memory
    - E.g. all threads in Block (1,0,0) see the same data, but cannot see Block (1,1,0)'s data
  - This memory resides on the GPU-chip and is very VERY fast
    - Only 16KB per GPU-core
      - not per-block!
      - your GPU-occupancy matters



Fermi has up to 48KB

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 after every read or write
  - E.g. if Thread-1 writes data and Thread-2 reads it ... still not guaranteed to be the same data
    - You must call __syncthreads() before you read the data
- Be careful that you don't overrun the __shared__ array bounds
- '-Xptxas=-v' will also show your block-shared memory usage

Block-Shared Memory, cont'd

- Since block-shared memory is so limited in size, you often need to "chunk" your data
  - I.e. read a chunk of 256 values, process them, read another chunk of 256 values, process them, ...
  - Make sure you __syncthreads every time new data is read into the __shared__ array

- You can specify the per-block size of the shared array at launch-time:

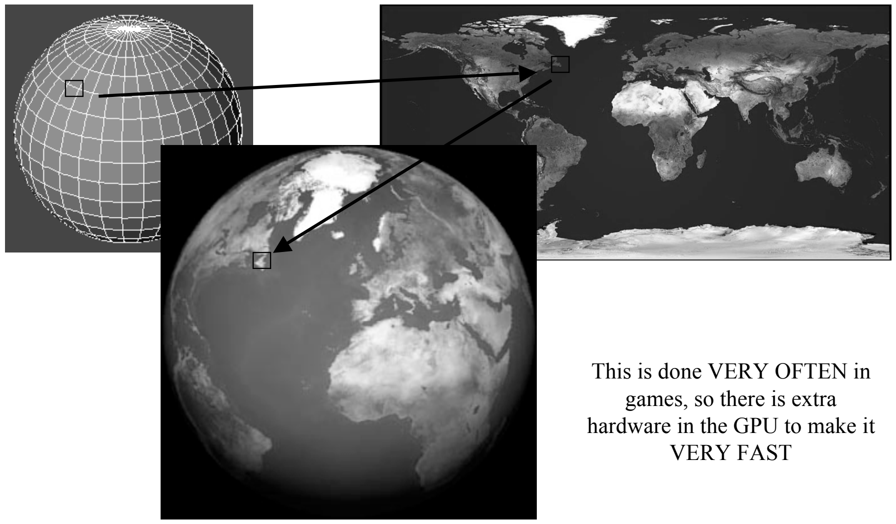
```
__shared__ float* tmp_x;
__global__ void partial_sums( int n, float* x, float* y ) {
    . . .
}
int main() {
    . . .
    partial_sums<<<m,n,1024>>>( n, x, y );
    . . .
}
```

size in BYTES!

You cannot cudaMemcpy into a __shared__ array

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

Our “real” dataset, ‘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

Multi-GPU Programming

- If one is good, four must be better!!
  - S870 system packs 4 GF8s into an external box (external power)
    - S1070 packs 4 G100s into an external box ← 1000W just in GPUs
  - 9800GX2 is 2 GF9s on a single PCI card
- One approach is to switch between GPUs before every CUDA call:

```
cudaSetDevice( n );
```

  - Need to do this before any mem-copy, launch, thread-sync, etc.
  - If you forget what GPU you're talking to ... BAD!!
  - Each GPU has its own memory pool ... need to keep pointers straight
  - Note that CUDA will time-share any GPU, so if you don't explicitly set the device, the program will still run (on GPU#0) ... slowly
  - There is no direct GPU-to-GPU synchronization or communication in CUDA

SIMD and "Warps"

- The GPU really has several program-counters, each one controls a group of threads called a "warp"
  - 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
  - Some threads will be idle (not doing work) some of the time
    - Unless you can guarantee that the conditional keeps “Warps” together
    - Presently a warp is a set of 32 threads; 0-31, 32-63, etc.
- 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

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 );
/* CPU can do work here */
cudaThreadSynchronize();
/* GPU is now done, CPU is sync’d */
```

- 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 (GF9, 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 );
```

  - Mem-copy of x2 should happen WHILE func1 is running

- Synchronizing all of this gets complicated
  - See cudaEvent and cudaStream functions