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.
jbpm1@duke.edu

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
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:

Why GPU programming?

- Parallelism
  - CPUs recently moved to dual- and quad-core chips
  - The current top-of-the-line GPU (GeForce-200) 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
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

CPU

Memory

GPU

Memory

HT 20.8GB/s

GPU Bus 141.7GB/s

PCle-x16 4GB/s
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

```c
__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 */
    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

- 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

```c
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

Kernel Func: \[\begin{array}{ccc} -1 & 2 & -1 \end{array}\]

Input: \[\begin{array}{cccccccc} 4 & 2 & 1 & 5 & 6 & 3 & 4 & 3 \end{array}\]

Output: \[\begin{array}{cccccccc} 6 & -1 & -5 & 3 & 4 & -4 & 2 & 2 \end{array}\]

Sequential computation ... 8 clock-ticks

Parallel (GPU) View of Stream Computing

Kernel Func: \[\begin{array}{ccc} -1 & 2 & -1 \end{array}\]

Input: \[\begin{array}{cccccccc} 4 & 2 & 1 & 5 & 6 & 3 & 4 & 3 \end{array}\]

Output: \[\begin{array}{cccccccc} 6 & -1 & -5 & 3 & 4 & -4 & 2 & 2 \end{array}\]

Parallel (4-way) computation ... 2 clock-ticks

... NVIDIA 200-series 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 200-series: 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
1-D x 1-D Example

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

```
<table>
<thead>
<tr>
<th>Block Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Block Index (0,0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread Number</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Vector:</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=1024</td>
</tr>
<tr>
<td>0-255</td>
</tr>
<tr>
<td>256-511</td>
</tr>
<tr>
<td>512-767</td>
</tr>
<tr>
<td>768-1024</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Block #s:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Thread #s:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Vector:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-63</td>
</tr>
<tr>
<td>64-127</td>
</tr>
<tr>
<td>128-191</td>
</tr>
<tr>
<td>192-255</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Vector:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-255</td>
</tr>
</tbody>
</table>
```
2-D x 2-D Example

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

Block Number

(0,0) (0,1)
(1,0) (1,1)

Thread Number

(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
(3,0) (3,1) (3,2) (3,3)

Block Index (1,1,0)
Thread Index (2,1,0)

2-D x 2-D Example, cont’d

Raw Image
800 x 800

Grid
2x2

Each block does 400x400

Threads
4x4

Each thread does 100x100
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

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

```cpp
uint3 m,n;
m = make_uint3(128,128,1);
n = make_uint3(32,32,1);
vcos<<<m,n>>>( n, dev_x, dev_y );
```

Mapping the Parallelism to Threads

`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 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)

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

__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”

```c
__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] );
    }
}
```

```c
__global__ void image_pro( int wd, int ht, float* x, float* y ) {
    if( (blockIdx.x*blockDim.x+threadIdx.x) < wd ) {
        if( (blockIdx.y*blockDim.y+threadIdx.y) < ht ) {
            ...
        }
    }
}
```

Compilation

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

```bash
% nvcc -o simple simple.cu
```
Running a CUDA Program

* Just execute it!

```bash
./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

Compilation Details (-keep)

```
nvcc myprog
myprog (CUDA/GPU Code)
gcc myprog (C Code)
nvcc
myprog (GPU ASM)
ptxas myprog (PTX ASM)
nvcc myprog.exe
```
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.

Error Handling

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

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

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

```c
    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 …

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!

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

```
func1<<<grd,blk>>>( arguments );
err1 = cudaGetLastError();
...
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:

```c
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
  - Inf + number => Inf
  - 0 / number => Inf
  - NaN + anything => NaN
  - Inf - Inf => NaN
  - 0/0 or Inf/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!

---

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.

**Occupancy**

- E.g. a GeForce-8 GPU has 8192 registers per GPU-core, and can run 768 threads simultaneously.
  - You can run multiple blocks on 1 GPU-core, but you cannot run 1 block across 2 GPU-cores.

- Step 1, compile with `-Xptxas=-v` to see your register usage.

- E.g. 10 registers per thread and 300 threads per block.
  - Each block requires 3000 registers ... so 2 blocks per GPU-core.
  - But 2 blocks is only 600 running threads ... we COULD do 768.
    - 78% Occupancy.

- We could adjust the threads-per-block (hint - try 256).
- Or we could re-compile with `--maxrregcount=8` (won’t actually help with this example).
Occupy, 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%
  - .. 256 th/bk .. 4 bk/core .. 512 th/core .. 67%
  - 16 reg .. 128 th/bk .. 2 bk/core .. 512 th/core .. 67%
  - .. 256 th/bk .. 3 bk/core .. 384 th/core .. 50%
  - 20 reg .. 128 th/bk .. 1 bk/core .. 768 th/core .. 33%

GeForce-200 has 16384 registers, and 1024 simultaneous threads

- Varying Register Use (GF200):
  - 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 16 GPU-cores
    - E.g. 10 reg/thread, 256 thr/blk, 3 blk/core ... minimum of 48 blocks
  - GeForce-200 has 30 GPU-cores
    - E.g. 10 reg/thread, 256 thr/blk, 8 blk/core ... minimum of 240 blocks!

- 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 $x + ix \times 4$ (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.2.0 and yet it supports 4 versions of GPUs, and ??? graphics products
Tuning Performance to a Specific GPU

What kind of GPU am I running on?

```c
CUDAdeviceProperties( 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 ... GeForce-200 ... 1024 threads per core, double-precision
- props.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

```c
__global__ void func(int n, float* x) {
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    x[ix] = 0.0f;
}
```

Be careful with integer division!
Some More Examples

```c
__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);
```

```c
#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

  - 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!
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

Block-Shared Memory, cont’d

```c
__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:

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

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

```c
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 );
    ...
}
```

- 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
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 GF200) 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)

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 (GF9, GF200?), you can potentially overlap GPU-memory transfers and GPU-function computations:

```c
/* 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

* Synchronizing all of this gets complicated
  ◆ See cudaEvent and cudaStream functions

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]

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
int ttl_nthreads = gridDim.x*blockDim.x;
for(i=idx;i<ttl_nthreads;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

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 GF200s into an external box
  - 9800GX2 is 2 GF9s on a single PCI card

- Best approach is to spawn multiple CPU-threads, one per GPU
  - Each CPU-thread then attaches to a separate GPU
    ```c
    cudaSetDevice( n );
    ```
  - Note that 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 direct GPU-to-GPU synchronization or communication 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 launch a new batch of threads onto their GPUs
### Conditionals

- Generally, conditionals on some F(threadIdx) are bad for performance
  - The way the GPU works, 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; 0-31, 32-63, etc.

- Conditionals on F(blockIdx) are fine

- Be careful with loop bounds

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

- The end-clause is just a conditional

### 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?

```c
__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