Supercomputing on your desktop:
Programming the next generation of cheap and massively parallel hardware using CUDA

CUDA Advanced #2
During this course, we’ll try to "Borrow Smart" and use existing material ;-)

Friday, January 23, 2009
Wanna Play with The Big Guys?
Here are the keys to High-Performance in CUDA
Hoare said (and Knuth restated)

“Premature optimization is the root of all evil.”
To optimize or not to optimize

Hoare said (and Knuth restated)

“We should forget about small efficiencies, say about 97% of the time:
Premature optimization is the root of all evil.”

⇓

3% of the time we really should worry about small efficiencies
(Every 33rd codeline)
Optimization goals

- We should strive to reach GPU performance
- We must know the GPU performance
  - Vendor specifications
  - Synthetic benchmarks
- Choose a performance metric
  - Memory bandwidth or GFLOPS?
- Use `clock()` to measure
- Experiment and profile!
A kernel is executed as a grid of thread blocks.

A thread block is a batch of threads that can cooperate with each other by:
- Sharing data through shared memory
- Synchronizing their execution

Threads from different blocks cannot cooperate.
Data Movement in a CUDA Program

Host Memory
Device Memory
[Shared Memory]
COMPUTATION
[Shared Memory]
Device Memory
Host Memory

© NVIDIA Corporation 2008
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it’s better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host
Optimize Memory Coherence

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory

- Optimize for spatial locality in cached texture memory

- In shared memory, avoid high-degree bank conflicts
Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory

- Use one / a few threads to load / compute data shared by all threads

- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later
Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks

- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory
Memory optimizations

- Optimizing memory transfers
- Coalescing global memory accesses
- Using shared memory effectively
Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16) vs. 80 GB/s peak (Quadro FX 5600)
  - 8GB/s for PCI-e 2.0

- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory

- Group transfers
  - One large transfer much better than many small ones
Page-Locked Memory Transfers

- `cudaMallocHost()` allows allocation of page-locked host memory
- Enables highest `cudaMemcpy` performance
  - 3.2 GB/s+ common on PCI-express (x16)
  - ~4 GB/s measured on nForce 680i motherboards (overclocked PCI-e)

- See the “bandwidthTest” CUDA SDK sample

- Use with caution
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits
Global Memory Reads/Writes

- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance
  - Coalescing: up to 10x speedup
  - Latency hiding: up to 2.5x speedup
Accessing global memory

- 4 cycles to issue on memory fetch
- but 400-600 cycles of latency
  - The equivalent of 100 MADs
- Likely to be a performance bottleneck
- Order of magnitude speedups possible
  - Coalesce memory access
- Use shared memory to re-order non-coalesced addressing
Coalescing

A coordinated read by a half-warp (16 threads)
A contiguous region of global memory:
- **64 bytes** - each thread reads a word: _int_, _float_, ...
- **128 bytes** - each thread reads a double-word: _int2_, _float2_, ...
- **256 bytes** – each thread reads a quad-word: _int4_, _float4_, ...

Additional restrictions on G8X/G9X architecture:
- Starting address for a region must be a multiple of region size
- The \(k^{th}\) thread in a half-warp must access the \(k^{th}\) element in a block being read

Exception: not all threads must be participating
- Predicated access, divergence within a halfwarp
Coalesced Access: Reading floats

All threads participate

Some Threads Do Not Participate
Uncoalesced Access: Reading floats

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing: Timing Results

- **Experiment on G80:**
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs

- **12K blocks x 256 threads:**
  - 356μs – coalesced
  - 357μs – coalesced, some threads don’t participate
  - 3,494μs – permuted/misaligned thread access
Coalescing: Structures of size ≠ 4, 8, 16 Bytes

Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

If SoA is not viable:
- Force structure alignment: __align(X), where X = 4, 8, or 16
- Use SMEM to achieve coalescing

Point structure

| x | y | z |

AoS

| x | y | z | x | y | z | x | y | z |

SoA

| x | x | x | y | y | y | z | z | z |
Coalescing: Summary

- Coalescing greatly improves throughput

- Critical to memory-bound kernels

- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM

- Additional resources:
  - Aligned Types SDK Sample
Parallel Memory Architecture

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth

- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
Bank Addressing Examples

**No Bank Conflicts**
- Linear addressing
- stride == 1

```
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
<tr>
<td>Thread 15</td>
<td>Bank 15</td>
</tr>
</tbody>
</table>
```

**Random 1:1 Permutation**

```
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Bank 1</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Bank 2</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Bank 3</td>
</tr>
<tr>
<td>Thread 4</td>
<td>Bank 4</td>
</tr>
<tr>
<td>Thread 5</td>
<td>Bank 5</td>
</tr>
<tr>
<td>Thread 6</td>
<td>Bank 6</td>
</tr>
<tr>
<td>Thread 7</td>
<td>Bank 7</td>
</tr>
<tr>
<td>Thread 15</td>
<td>Bank 15</td>
</tr>
</tbody>
</table>
```
Bank Addressing Examples

2-way Bank Conflicts
- Linear addressing
  stride == 2

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 8
- Thread 9
- Thread 10
- Thread 11

- Bank 0
- Bank 1
- Bank 2
- Bank 3
- Bank 4
- Bank 5
- Bank 6
- Bank 7
- Bank 15

8-way Bank Conflicts
- Linear addressing
  stride == 8

- Thread 0
- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Thread 6
- Thread 7

- Bank 0
- Bank 1
- Bank 2
- Bank 7
- Bank 8
- Bank 9
- Bank 15

x8

Friday, January 23, 2009
How addresses map to banks on G80

- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp
Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts

The fast case:
- If all threads of a half-warp access different banks, there is no bank conflict
- If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)

The slow case:
- Bank Conflict: multiple threads in the same half-warp access the same bank
- Must serialize the accesses
- Cost = max # of simultaneous accesses to a single bank
Use the right kind of memory

- **Constant memory:**
  - Quite small, \( \approx 20K \)
  - As fast as register access if all threads in a warp access the same location

- **Texture memory:**
  - Spatially cached
  - Optimized for 2D locality
  - Neighboring threads should read neighboring addresses
  - No need to think about coalescing

- **Constraint:**
  - These memories can only be updated from the CPU
Memory optimizations roundup

- CUDA memory handling is complex
  - And I have not covered all topics...
- Using memory correctly can lead to huge speedups
  - At least CUDA expose the memory hierarchy, unlike CPUs
- Get your algorithm up and running first, then optimize
- Use shared memory to let threads cooperate
- Be wary of “data ownership”
  - A thread does not have to read/write the data it calculates
Conflicts, Coalescing, Warps... I hate growing up.
Optimization Example: Matrix Transpose
Matrix Transpose

SDK Sample ("transpose")
Illustrates:
- Coalescing
- Avoiding SMEM bank conflicts
- Speedups for even small matrices
__global__ void transpose_naive(float *odata, float *idata, int width, int height) 
{
  1. unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  2. unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
  
  3. if (xIndex < width && yIndex < height) 
  
    4.   unsigned int index_in   = xIndex + width * yIndex;
    5.   unsigned int index_out = yIndex + height * xIndex;
    6.   odata[index_out] = idata[index_in];

}
Coalesced Transpose

- Assumption: matrix is partitioned into square tiles
- **Threadblock** \((bx, by)\):
  - Read the \((bx, by)\) input tile, store into SMEM
  - Write the SMEM data to \((by, bx)\) output tile
    - Transpose the indexing into SMEM
- **Thread** \((tx, ty)\):
  - Reads element \((tx, ty)\) from input tile
  - Writes element \((tx, ty)\) into output tile

**Coalescing is achieved if:**
- Block/tile dimensions are multiples of **16**
# Coalesced Transpose

## Reads from GMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
<td></td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
<td></td>
</tr>
</tbody>
</table>

## Writes to SMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
<td></td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
<td></td>
</tr>
</tbody>
</table>

## Reads from SMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>1,0</th>
<th>2,0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td>15,1</td>
<td></td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td>15,15</td>
<td></td>
</tr>
</tbody>
</table>

## Writes to GMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,15</td>
<td></td>
</tr>
<tr>
<td>15,0</td>
<td>15,1</td>
<td>15,2</td>
<td>15,15</td>
<td></td>
</tr>
</tbody>
</table>

---

**Example**

Friday, January 23, 2009
SMEM Optimization

Reads from SMEM

<table>
<thead>
<tr>
<th></th>
<th>0,0</th>
<th>0,1</th>
<th>0,15</th>
<th>15,0</th>
<th>15,1</th>
<th>15,15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Threads read SMEM with stride = 16
  - Bank conflicts

Solution

- Allocate an “extra” column
- Read stride = 17
- Threads read from consecutive banks
SMEM Optimization

Reads from SMEM

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
</tr>
<tr>
<td>0,1</td>
<td>1,1</td>
<td>2,1</td>
</tr>
<tr>
<td>0,15</td>
<td>1,15</td>
<td>2,15</td>
</tr>
</tbody>
</table>

- Threads read SMEM with stride = 16
  - Bank conflicts

Solution

- Allocate an “extra” column
- Read stride = 17
- Threads read from consecutive banks
__global__ void transpose(float *odata, float *idata, int width, int height)
{

1. __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];

2. unsigned int xBlock = blockDim.x * blockIdx.x;
3. unsigned int yBlock = blockDim.y * blockIdx.y;
4. unsigned int xIndex = xBlock + threadIdx.x;
5. unsigned int yIndex = yBlock + threadIdx.y;
6. unsigned int index_out, index_transpose;

7. if (xIndex < width && yIndex < height)
{
8.   unsigned int index_in = width * yIndex + xIndex;
9.   unsigned int index_block = threadIdx.y * (BLOCK_DIM+1) + threadIdx.x;
10.  block[index_block] = idata[index_in];
11.  index_transpose = threadIdx.x * (BLOCK_DIM+1) + threadIdx.y;
12.  index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
}\n
13. __syncthreads();

14. if (xIndex < width && yIndex < height)
15.   odata[index_out] = block[index_transpose];
}
Coalesced transpose: Source code

```c
__global__ void transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOC
```
```
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[index_transpose];
    }
}
```

__global__ void
transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if (xIndex < width && yIndex < height) {
        out[index_out] = block[index_transpose];
    }
}
Coalesced transpose: Source code

```c
__global__ void
transpose( float *out, float *in, int w, int h ) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, indexTranspose;

    if ( xIndex < width && yIndex < height ) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        indexTranspose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }

    __syncthreads();

    if ( xIndex < width && yIndex < height ) {
        out[index_out] = block[indexTranspose];
    }
}
```

Allocate shared memory.

Set up indexing.

Check that we are within domain, calculate more indices.
__global__ void
transpose( float *out, float *in, int w, int h ) {
__shared__ float block[BLOCK_DIM*BLOCK_DIM];

unsigned int xBlock = blockDim.x * blockIdx.x;
unsigned int yBlock = blockDim.y * blockIdx.y;

unsigned int xIndex = xBlock + threadIdx.x;
unsigned int yIndex = yBlock + threadIdx.y;

unsigned int index_out, index_transpose;

if ( xIndex < width && yIndex < height ) {
    unsigned int index_in = width * yIndex + xIndex;
    unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;
    block[index_block] = in[index_in];

    index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
    index_out = height * (xBlock + threadIdx.x) + yBlock + threadIdx.x;
}
__syncthreads();

if ( xIndex < width && yIndex < height ) {
    out[index_out] = block[index_transpose];
}
}
Coalesced transpose: Source code

```c
__global__ void
transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;

    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];

        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if (xIndex < width && yIndex < height) {
        out[index_out] = block[index_transpose];
    }
}
```

- Allocate shared memory.
- Set up indexing.
- Check that we are within domain, calculate more indices.
- Write to shared memory.
- Calculate output indices.
Coalesced transpose: Source code

```c
__global__ void transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOCK_DIM];

    unsigned int xBlock = blockDim.x * blockIdx.x;
    unsigned int yBlock = blockDim.y * blockIdx.y;

    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = width * yIndex + xIndex;
        unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

        block[index_block] = in[index_in];
        index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
        index_out = height * (xBlock + threadIdx.y) + yBlock + threadIdx.x;

        out[index_out] = block[index_transpose];
    }
}
```

Allocate shared memory.
Set up indexing.
Check that we are within domain, calculate more indices.
Write to shared memory.
Calculate output indices.
Synchronize.

NB: outside if-clause

Example

Example

Friday, January 23, 2009
Coalesced transpose: Source code

```c
__global__ void
transpose(float *out, float *in, int w, int h) {
    __shared__ float block[BLOCK_DIM*BLOC
```
 transpose timings

Was it worth the trouble?

<table>
<thead>
<tr>
<th>Grid Size</th>
<th>Coalesced</th>
<th>Non-coalesced</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 × 128</td>
<td>0.011 ms</td>
<td>0.022 ms</td>
<td>2.0×</td>
</tr>
<tr>
<td>512 × 512</td>
<td>0.07 ms</td>
<td>0.33 ms</td>
<td>4.5×</td>
</tr>
<tr>
<td>1024 × 1024</td>
<td>0.30 ms</td>
<td>1.92 ms</td>
<td>6.4×</td>
</tr>
<tr>
<td>1024 × 2048</td>
<td>0.79 ms</td>
<td>6.6 ms</td>
<td>8.4×</td>
</tr>
</tbody>
</table>

For me, this is a clear yes.
Execution Optimizations
Know the arithmetic cost of operations

- 4 clock cycles:
  - Floating point: add, multiply, fused multiply-add
  - Integer add, bitwise operations, compare, min, max

- 16 clock cycles:
  - Reciprocal, reciprocal square root, \( \log(x) \), 32-bit integer multiplication

- 32 clock cycles:
  - \( \sin(x) \), \( \cos(x) \) and \( \exp(x) \)

- 36 clock cycles:
  - Floating point division (24-bit version in 20 cycles)

- Particularly costly:
  - Integer division, modulo
  - Remedy: Replace with shifting whenever possible

- Double precision (when available) will perform at half the speed
Occupancy

Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently

Limited by resource usage:
- Registers
- Shared memory
Grid/Block Size Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute

- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren’t waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Register Dependency

Read-after-write register dependency
- Instruction’s result can be read ~22 cycles later

Scenarios: CUDA: PTX:

\[
\begin{align*}
\text{x} &= \text{y} + 5; \\
\text{z} &= \text{x} + 3; \\
\text{s}\_\text{data}[0] &=+ 3; \\
\end{align*}
\]

\[
\begin{align*}
\text{add.f32} &\quad \text{f3, f1, f2} \\
\text{add.f32} &\quad \text{f5, f3, f4} \\
\text{ld.shared.f32} &\quad \text{f3, [r31+0]} \\
\text{add.f32} &\quad \text{f3, f3, f4} \\
\end{align*}
\]

To completely hide the latency:
- Run at least 192 threads (6 warps) per multiprocessor
- At least 25% occupancy
- Threads do not have to belong to the same thread block

Friday, January 23, 2009
Register Pressure

- Hide latency by using more threads per SM

Limiting Factors:
- Number of registers per kernel
  - 8192 per SM, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per SM, partitioned among concurrent threadblocks

- Check .cubin file for # registers / kernel

- Use \texttt{-maxrregcount=N} flag to NVCC
  - \( N \) = desired maximum registers / kernel
  - At some point “spilling” into LMEM may occur
    - Reduces performance – LMEM is slow
    - Check .cubin file for LMEM usage
Determining resource usage

- Use "--ptxoptions=-v" option to nvcc
- Or, compile the kernel code with the -cubin flag to determine register usage.
- Open the .cubin file with a text editor and look for the "code" section.

```c
architecture {sm_10}
abiversion {0}
modname {cubin}
code {
  name = BlackScholesGPU
  lmem = 0
  smem = 68
  reg = 20
  bar = 0
  bincode {
    0xa0004205 0x04200780 0x40024c09 0x00200780
    ...
  }
```

- per thread local memory
- per thread block shared memory
- per thread registers
CUCKO GPU Occupancy Calculator

1) Select a GPU from the list (click) G80

2) Enter your resource usage:
   - Threads Per Block
   - Registers Per Thread
   - Shared Memory Per Block (bytes)

3) GPU Occupancy Data is displayed here and in the graphs:
   - Active Threads per Multiprocessor
   - Active Warps per Multiprocessor
   - Active Threads Blocks per Multiprocessor
   - Occupancy of each Multiprocessor
   - Maximum Simultaneous Blocks per GPU

4) Physical Limits for GPU
   - G80
   - Multiprocessors per GPU
   - Threads Per Warp
   - Warps Per Multiprocessor
   - Threads Per Multiprocessor
   - Thread Decays Per Multiprocessor
   - Total # of 32-bit registers / Multiprocessor
   - Shared Memory (Multiprocessor bytes)

5) Allocation Per Thread Block
   - Warp
   - Registers
   - Shared Memory

6) These data are used in computing the occupancy data in blue:
   - Maximum Thread Blocks Per Multiprocessor
     - Blocks
   - Linked by Max Warps Per Multiprocessor
   - Linked by Registers Per Multiprocessor
   - Linked by Shared Memory Per Multiprocessor

7) Thread Block Limit Per Multiprocessor is the minimum of these 3

8) CUDA Occupancy Calculator

9) Version

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

![Graphs showing occupancy](attachment://graph.png)
Optimizing threads per block

Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps

More threads per block == better memory latency hiding

But, more threads per block == fewer registers per thread
  - Kernel invocations can fail if too many registers are used

Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!
Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

**BUT...**

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)
Parameterize Your Application

- Parameterization helps adaptation to different GPUs

- GPUs vary in many ways
  - # of multiprocessors
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Threads per block

- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration
Sometimes we know some kernel parameters at compile time:
- # of loop iterations
- Degrees of polynomials
- Number of data elements

If we could “tell” this to the compiler, it can unroll loops and optimize register usage

We need to be generic
- Avoid code duplication, sizes unknown at compile time

Templates to rescue
- The same trick can be used for regular C++ sources
Example: de Casteljau algorithm

A standard algorithm for evaluating polynomials in Bernstein form

Recursively defined:

\[ f(x) = b_{00}^d \]

\[ b_{i,j}^k = xb_{i+1,j}^{k-1} + (1 - x)b_{i,j+1}^{k-1} \]

\[ b_{i,j}^0 \text{ are coefficients} \]
The de Casteljau algorithm is usually implemented as nested for-loops

- Coefficients are overwritten for each iteration

```c
float deCasteljau(float *c, float x, int d) {
    for (uint i = 1; i <= d; ++i) {
        for (uint j = 0; j <= d-i; ++j) {
            c[j] = (1.0f-x)*c[j] + x*c[j+1];
        }
    }
    return c[0];
}
```

\[ f(x) = c_0^d \]
Template loop unrolling

- We make \( d \) a template parameter

```cpp
template<int d>
float deCasteljau(float* c, float x, int d)
{
    for (uint i = 1; i <= d; ++i)
    {
        for (uint j = 0; j <= d-i; ++j)
            c[j] = (1.0f-x)*c[j] + x*c[j+1];
    }
    return c[0];
}
```

- Kernel is called as

```cpp
switch (d) {
    case 1:
        deCasteljau<1>(c, x); break;
    case 2:
        deCasteljau<2>(c, x); break;
    ...
    case MAXD:
        deCasteljau<MAXD>(c, x); break;
}
```
Results

- For the de Casteljau algorithm we see a relatively small speedup
  - $\approx 1.2 \times (20\%...)$
- Very easy to implement
- Can lead to long compile times

Conclusion:
- Probably worth it near end of development cycle
Conclusion

- Understand CUDA performance characteristics
  - Memory coalescing
  - Divergent branching
  - Bank conflicts
  - Latency hiding

- Use peak performance metrics to guide optimization

- Understand parallel algorithm complexity theory

- Know how to identify type of bottleneck
  - e.g. memory, core computation, or instruction overhead

- Optimize your algorithm, *then* unroll loops

- Use template parameters to generate optimal code
The CUDA Visual Profiler

- Helps measure and find potential performance problem
  - GPU and CPU timing for all kernel invocations and memcpys
  - Time stamps

- Access to hardware performance counters
Events are tracked with hardware counters on signals in the chip:

- **timestamp**
- **gld_incoherent**
- **gld_coherent**
- **gst_incoherent**
- **gst_coherent**
- **local_load**
- **local_store**
- **branch**
- **divergent_branch**
- **instructions** – instruction count
- **warp_serialize** – thread warps that serialize on address conflicts to shared or constant memory
- **cta_launched** – executed thread blocks

Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)

Local loads/stores

Total branches and divergent branches taken by threads
Interpreting profiler counters

- Values represent events within a thread warp

- Only targets one multiprocessor
  - Values will not correspond to the total number of warps launched for a particular kernel.
  - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.

- Values are best used to identify relative performance differences between unoptimized and optimized code
  - In other words, try to reduce the magnitudes of `gld/gst_incoherent`, `divergent_branch`, and `warpวี serialize`
## Performance for 4M element reduction

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Description</th>
<th>Time ((2^{22}) ints)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Cumulative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel 1:</td>
<td>interleaved addressing with divergent branching</td>
<td>8.054 ms</td>
<td>2.083 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel 2:</td>
<td>interleaved addressing with bank conflicts</td>
<td>3.456 ms</td>
<td>4.854 GB/s</td>
<td>2.33x</td>
<td>2.33x</td>
</tr>
<tr>
<td>Kernel 3:</td>
<td>sequential addressing</td>
<td>1.722 ms</td>
<td>9.741 GB/s</td>
<td>2.01x</td>
<td>4.68x</td>
</tr>
<tr>
<td>Kernel 4:</td>
<td>first add during global load</td>
<td>0.965 ms</td>
<td>17.377 GB/s</td>
<td>1.78x</td>
<td>8.34x</td>
</tr>
<tr>
<td>Kernel 5:</td>
<td>unroll last warp</td>
<td>0.536 ms</td>
<td>31.289 GB/s</td>
<td>1.8x</td>
<td>15.01x</td>
</tr>
<tr>
<td>Kernel 6:</td>
<td>completely unrolled</td>
<td>0.381 ms</td>
<td>43.996 GB/s</td>
<td>1.41x</td>
<td>21.16x</td>
</tr>
<tr>
<td>Kernel 7:</td>
<td>multiple elements per thread</td>
<td>0.268 ms</td>
<td>62.671 GB/s</td>
<td>1.42x</td>
<td>30.04x</td>
</tr>
</tbody>
</table>

Kernel 7 on 32M elements: 72 GB/s!
Build your own!

Our 16-GPU Monster-Class Supercomputer
the world’s most compact (18”x18”x18”) and inexpensive ($3000) supercomputer
Thank you!
Back Pocket Slides
Processor Parallelism

CPUs
Multiple cores driving performance increases

Emerging Intersection

GPUs
Increasingly general purpose data-parallel computing
Improving numerical precision

OpenCL - Heterogenous Computing

OpenCL - Open Computing Language
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors
Misc
## Tesla C1060 Computing Processor

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td>1x Tesla T10P</td>
</tr>
<tr>
<td><strong>Core GHz</strong></td>
<td>1.33 GHz</td>
</tr>
<tr>
<td><strong>Form factor</strong></td>
<td>Full ATX: 4.736” (H) x 10.5” (L) Dual slot wide</td>
</tr>
<tr>
<td><strong>On-board memory</strong></td>
<td>4 GB</td>
</tr>
<tr>
<td><strong>System I/O</strong></td>
<td>PCIe x16 gen2</td>
</tr>
<tr>
<td><strong>Memory I/O</strong></td>
<td>512-bit, 800MHz DDR 102 GB/s peak bandwidth</td>
</tr>
<tr>
<td><strong>Display outputs</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Typical power</strong></td>
<td>160 W</td>
</tr>
</tbody>
</table>
# Tesla S1070 1U System

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processors</strong></td>
<td>4 x Tesla T10P</td>
</tr>
<tr>
<td><strong>Core GHz</strong></td>
<td>1.5 GHz</td>
</tr>
<tr>
<td><strong>Form factor</strong></td>
<td>1U for an EIA 19” 4-post rack</td>
</tr>
<tr>
<td><strong>Total 1U system memory</strong></td>
<td>16 GB (4.0GB per GPU)</td>
</tr>
<tr>
<td><strong>System I/O</strong></td>
<td>2 PCIe x16</td>
</tr>
<tr>
<td><strong>Memory I/O per processor</strong></td>
<td>512-bit, 800MHz GDDR 102 GB/s peak bandwidth</td>
</tr>
<tr>
<td><strong>Display outputs</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Typical power</strong></td>
<td>700 W</td>
</tr>
<tr>
<td><strong>Chassis dimensions</strong></td>
<td>1.73” H × 17.5” W × 28.5” D</td>
</tr>
</tbody>
</table>
## Double Precision Floating Point

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA GPU</th>
<th>SSE2</th>
<th>Cell SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precision</strong></td>
<td>IEEE 754</td>
<td>IEEE 754</td>
<td>IEEE 754</td>
</tr>
<tr>
<td><strong>Rounding modes for FADD and FMUL</strong></td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>All 4 IEEE, round to nearest, zero, inf, -inf</td>
<td>Round to zero/truncate only</td>
</tr>
<tr>
<td><strong>Denormal handling</strong></td>
<td>Full speed</td>
<td>Supported, costs 1000’s of cycles</td>
<td>Flush to zero</td>
</tr>
<tr>
<td><strong>NaN support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Overflow and Infinity support</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No infinity, clamps to max norm</td>
</tr>
<tr>
<td><strong>Flags</strong></td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
</tr>
<tr>
<td><strong>FMA</strong></td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Square root</strong></td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Division</strong></td>
<td>Software with low-latency FMA-based convergence</td>
<td>Hardware</td>
<td>Software only</td>
</tr>
<tr>
<td><strong>Reciprocal estimate accuracy</strong></td>
<td>24 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>Reciprocal sqrt estimate accuracy</strong></td>
<td>23 bit</td>
<td>12 bit</td>
<td>12 bit</td>
</tr>
<tr>
<td><strong>log2(x) and 2^x estimates accuracy</strong></td>
<td>23 bit</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>