ME964
High Performance Computing for Engineering Applications

Memory Layout in CUDA
Execution Scheduling in CUDA
February 15, 2011

“They have computers, and they may have other weapons of mass destruction.”
Janet Reno, former Attorney General of the United States
Before We Get Started…

- **Last time**
  - Wrapped up CUDA API short overview
  - Started discussion on memory ecosystem on the GPU card
  - Started example of tiled matrix-matrix multiplication
    - Vehicle for introducing the concept of shared memory and thread synchronization

- **Today**
  - Wrap up tiled matrix-matrix multiplication
  - Discuss thread scheduling for execution on the GPU

- **HW**
  - HW4 has been posted. Due date: 02/17, 11:59 PM
  - Please indicate your preference for midterm project on the forum
Here’s Euler, in Diapers…

- Andrew and Hammad the delivery doctors on duty
- 32 Fermi GPUs
- Eight compute nodes, each with two quad core Intel Xeon 5520
- Hopefully operational upon your return from Spring break
- Hopefully you’ll be able to use authentication credentials from Newton to log into Euler
Multiply Using Several Blocks

- One block computes one square sub-matrix $C_{\text{sub}}$ of size $\text{Block}_\text{Size}$

- One thread computes one element of $C_{\text{sub}}$

- Assume that the dimensions of $A$ and $B$ are multiples of $\text{Block}_\text{Size}$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest

NOTE: Similar example provided in the CUDA Programming Guide 3.2
- Available on the class website
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
   int size;

   // Load A and B to the device
   float* Ad; size = hA * wA * sizeof(float);
   cudaMalloc((void**)&Ad, size); cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
   float* Bd; size = wA * wB * sizeof(float); cudaMalloc((void**)&Bd, size); cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

   // Allocate C on the device
   float* Cd; size = hA * wB * sizeof(float); cudaMalloc((void**)&Cd, size);

   // Compute the execution configuration assuming
   // the matrix dimensions are multiples of BLOCK_SIZE
   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
   dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

   // Launch the device computation
   Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

   // Read C from the device
   cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

   // Free device memory
   cudaFree(Ad); cudaFree(Bd); cudaFree(Cd);
}

(continues with next block…)

(continues below…)
__global__ void Muld(float* A, float* B, int wA, int wB, float* C) {
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
Synchronization Function

- It’s a device lightweight runtime API function
  - void __syncthreads();

- Synchronizes all threads in a block (acts as a barrier for all threads of a block)

- Once all threads have reached this point, execution resumes normally

- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory

- Allowed in conditional constructs only if the conditional is uniform across the entire thread block
The Shared Memory in the Context of the SM Memory Architecture [NVIDIA G80]

- Threads in a Block:
  - Cooperate through data accessible to all of them both in Global Memory and Shared Memory
  - Synchronize at barrier instruction

- Shared Memory is very good
  - Keeps data close to processor (low latency)
  - Minimize trips to global memory
  - Dynamically allocated at the SM level to each Block
  - One of the limiting resources

Courtesy: John Nickolls, NVIDIA
The Three Most Important Parallel Memory Spaces

- **Register**: per-thread basis
  - Private per thread
  - Can spill into local memory (perf. hit)
- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- **Global Memory**: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication
SM Register File (RF) [Tesla C1060]

- Register File (RF)
  - 64 KB (16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core

- TEX pipe can also read/write RF
  - 3 SMs share 1 TEX

- Global Memory Load/Store pipe can also read/write RF
Programmer View of Register File

- Number of 32 bit registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla C1060
  - 32K on Tesla C2050

- This is an implementation decision, not part of CUDA

- Registers are dynamically partitioned across all Blocks assigned to the SM

- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks

- Each thread in the same Block only access registers assigned to itself

Possible per-block partitioning scenarios of the RF available on the SM
Matrix Multiplication Example

[Tesla C1060]

- If each Block has 16x16 threads and each thread uses 20 registers, how many threads can run on each SM?
  - Each Block requires 20*256 = 5120 registers
  - 16,384 = 3 * 5120 + change
  - So, three blocks can run on an SM as far as registers are concerned

- What if each thread increases the use of registers by 2?
  - Each Block now requires 22*256 = 5632 registers
  - 16,384 < 16896= 5632 *3
  - Only two Blocks can run on an SM, about 33% reduction of parallelism!!!

- Example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism
More on Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism
Constant Memory

- This comes handy when all threads use the same *constant* value in their computation
  - Example: \( \pi \), some spring force constant, \( e=2.7173 \), etc.

- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)

- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
  - When all threads in a warp read the same constant memory address this is as fast as a register
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of "printf"]

#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;

    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>();
    cudaThreadSynchronize();
    return 0;
}

Hello thread 0, Circ= 0.00
Hello thread 1, Circ= 6.28
Hello thread 2, Circ=12.57
Hello thread 3, Circ=18.85
Hello thread 4, Circ=25.13
Hello thread 5, Circ=31.42
Hello thread 6, Circ=37.70
Hello thread 7, Circ=43.98
Hello thread 8, Circ=50.27
Hello thread 9, Circ=56.55
Hello thread 10, Circ=62.83
Hello thread 11, Circ=69.11
Hello thread 12, Circ=75.40
Hello thread 13, Circ=81.68
Hello thread 14, Circ=87.96
Hello thread 15, Circ=94.25
Memory Issues Not Addressed Yet...

- Not all global memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question

- Not all shared memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions

- To do justice to these topics we’ll need to talk first about scheduling threads for execution
  - Coming up next…
Thread Execution Scheduling

- GPU Architecture Paradigm: Single Instruction Multiple Data (SIMD)
  - CUDA perspective: Single Program Multiple Threads

- What’s the overall software (application) development model?
  - CUDA integrated CPU + GPU application C program
    - Serial C code executes on CPU
    - Parallel Kernel C code executes on GPU thread blocks

```
GPU Parallel Kernel
KernelA<<< nBlkA, nTidA >>>(args);

CPU Serial Code

GPU Parallel Kernel
KernelB<<< nBlkB, nTidB >>>(args);
```

Grid 0

Grid 1

17
In relation to a Block, the programmer decides:
- Block size: from 1 to 512 concurrent threads
- Block dimension (shape): 1D, 2D, or 3D
- # of threads in each dimension

Threads have thread idx numbers within Block
Threads within Block share data and may synchronize while each is doing its work
Thread program uses thread idx to select work and address shared data
Beyond the concept of thread idx we brought into the picture the concept of thread id and how to compute a thread id based on the thread index
GeForce-8 Series HW Overview
Scheduling on the HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can launch next Block[s] in line
- NOTE: Two levels of scheduling:
  - For running [desirably] a large number of blocks on a small number of SMs (30/16/14/etc.)
  - For running up to 24 (or 32, on Tesla C1060) warps of threads on the 8 SPs available on each SM
Thread Scheduling/Execution

- Each Thread Block is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling units in SM

- If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only *one* of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
- How is this relevant?
  - Suppose your code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:
    
    \[
    \frac{400}{(6 \times 4)} = 16.6667 \Rightarrow 17 \text{ Warps}
    \]
SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - From instruction L1 cache
  - Into any instruction buffer slot

- Issue one “ready-to-go” warp instruction per 4 cycles
  - From any warp - instruction buffer slot
  - Operand scoreboard used to prevent hazards

- Issue selection based on round-robin/age of warp

- SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding

- Used to determine whether a thread is ready to execute

- A **scoreboard** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

- Old concept from CDC 6600 (1960s) to separate memory and computation
Scoreboarding from Example

Consider three separate instruction streams: warp1, warp3 and warp8.

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
</tbody>
</table>

Schedule at time \( k \):

- \( t = k \): warp 8 instruction 11
- \( t = k+1 \): warp 1 instruction 42
- \( t = k+2 \): warp 3 instruction 95
- \( t > k \): warp 8 instruction 12
- \( t = l+1 \): warp 3 instruction 96

Mary Hall, U-Utah
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Ready to write result</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Computing</td>
</tr>
</tbody>
</table>

Schedule at time k+1

Mary Hall, U-Utah
Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes “ready” after the needed values are deposited
  - Prevents hazards
  - Cleared instructions are eligible for issue

- Decoupled Memory/Processor pipelines
  - Any thread can continue to issue instructions until scoreboarding prevents issue

TB = Thread Block, W = Warp
Granularity Considerations

[NOTE: Specific to Tesla C1060]

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  - For 8X8, we have 64 threads per Block. Since each Tesla C1060 SM can manage up to 1024 threads, it could take up to 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. This is not an option anyway (we need less then 512 per block)

- NOTE: this type of thinking should be invoked for your target hardware (from where the need for auto-tuning software…)
ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers.

- Also, assume global loads have 400 cycles.
  - 3 Blocks can run on each SM.

- If a Compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load.
  - Only two blocks can now run on each SM.
  - However, one only needs 400 cycles/(8 instructions * 4 cycles/instruction) \(\approx 13\) Warps to tolerate the memory latency.
  - Two Blocks have 16 Warps. The performance can be actually higher!
Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.

- The threads of a block execute concurrently on one SM, and multiple blocks (up to 8) can execute concurrently on one SM.

- When a thread block finishes, a new block is launched on the vacated SM.
A Word on HTT

The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency.

Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations (doing multiple tasks at once).

For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.

HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.

- This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.

Similar to the use of multiple warps on the GPU to hide latency:
- The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060).
Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow!

- SSE contains 70 new instructions

**Example**

- Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

\[
\begin{align*}
\text{vec\_res.x} &= v1.x + v2.x; \\
\text{vec\_res.y} &= v1.y + v2.y; \\
\text{vec\_res.z} &= v1.z + v2.z; \\
\text{vec\_res.w} &= v1.w + v2.w;
\end{align*}
\]

- SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

```
movaps xmm0,address-of-v1 ; xmm0=v1.w | v1.y | v1.x
addps xmm0,address-of-v2 ; xmm0=v1.w+v2.w | v1.y+v2.y | v1.x+v2.x
movaps address-of-vec\_res,xmm0
```