Programming Massively Parallel Processors

Fall 2007 Illinois ECE 498AL1

Wen-mei Hwu, Prof. of ECE
David Kirk, Chief Scientist, NVIDIA and Professor of ECE

http://courses.ece.uiuc.edu/ece498AL1

Excerpts by Mark D. Hill, March 2008

Why Massively Parallel Processor

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API

- GPU in every PC and workstation - massive volume and potential impact

An Example of Physical Reality
Behind CUDA

GeForce 8800

16 highly threaded SM’s, >128 FPU’s, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU

Previous Projects

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Source</th>
<th>Kernel</th>
<th>% time</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264</td>
<td>H.264 v1.0, change in guess vector</td>
<td>34,811</td>
<td>194</td>
<td>35%</td>
</tr>
<tr>
<td>LBM</td>
<td>LBM v1.0, change to single-precision vector source update</td>
<td>1,481</td>
<td>285</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>RCS-72</td>
<td>RCS-72 challenge-circuit test</td>
<td>1,979</td>
<td>218</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite-element modeling, simulation of 3D graded materials</td>
<td>1,874</td>
<td>281</td>
<td>99%</td>
</tr>
<tr>
<td>RPES</td>
<td>RPES - Parallel Equation Solver; quantum chem, 2-electron interaction</td>
<td>1,104</td>
<td>146</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>PNS</td>
<td>PNS - Pet Net simulation of a distributed system</td>
<td>322</td>
<td>281</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Single-precision implementation of saxpy, used in Linpack's Gaussian elim. routine</td>
<td>952</td>
<td>31</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>TRACF</td>
<td>The Truncation Angular Correlation Function</td>
<td>536</td>
<td>98</td>
<td>96%</td>
</tr>
<tr>
<td>FDTD</td>
<td>Finite-Difference Time Domain analysis of 3D electromagnetic wave propagation</td>
<td>1,365</td>
<td>93</td>
<td>16%</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>Comparing a mri-Q of a rabbit’s brain to the brain construction</td>
<td>490</td>
<td>33</td>
<td>&gt;99%</td>
</tr>
</tbody>
</table>

Speedup of Applications

- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10x speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25x to 400x speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
- Keep in mind that the speedup also reflects how suitable the CPU is for executing the kernel
ECE 498AL

Lectures 7:
Threading Hardware in G80

---

**Single-Program Multiple-Data (SPMD)**

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks

---

**Grids and Blocks: CUDA Review**

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate

---

**CUDA Thread Block: Review**

- Programmer declares (Thread) Block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads have thread id numbers within Block
- Threads share data and synchronize while doing their share of the work
- Thread program uses thread id to select work and address shared data

---

**GeForce-8 Series HW Overview**

- Streaming Processor Array
  - SPA
  - Texture Processor Cluster (2 SM + TEX)
  - SM
  - Multi-threaded processor core
  - Functional processing unit for CUDA thread block
- SP
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

---

**CUDA Processor Terminology**

- SPA
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- TPC
  - Texture Processor Cluster (2 SM + TEX)
- SM
  - Streaming Multiprocessor (8 SP)
- Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block
- SP
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and memory access

Thread Life Cycle in HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SM’s
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - Eligible parallelism
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks

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
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

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/cycle
  - 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

- 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 scoreboard prevents issue
  - Allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

Decoupled Memory/Processor pipelines

- Any thread can continue to issue instructions until scoreboard prevents issue
- Allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories

Global, Constant, and Texture Memories

- Global memory:
  - Main means of communicating R/W data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

Parallel Memory Sharing

- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

SM Register File

- Register File (RF)
  - 32 KB
  - Provides 4 operands/clock
- TEX pipe can also read/write RF
  - 2 SMs share 1 TEX
- Load/Store pipe can also read/write RF
Programmer View of Register File

- There are 8192 registers in each SM in G80
  - 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, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself

Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
  - 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!

Shared Memory

- Each SM has 16 KB of Shared Memory
  - 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - read and write access
- Not used explicitly for pixel shader programs
  - we dislike pixels talking to each other 😞

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
- No Bank Conflicts
  - Random 1:1 Permutation

ECE 498AL
Lecture 2: The CUDA Programming Model
What is GPGPU?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Previous GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
  - Instruction sets
    - Lack of integer & bit ops
- Communication limited
  - Between pixels
  - Scatter $a[i]=p$

CUDA Programming Model:
A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
  - Very little creation overhead
  - GPU needs 1000x of threads for full efficiency
  - Multi-core CPU needs only a few

Extended C

- Declspecs
  - _device_ float filter[N];
  - __global__ void convolve (float *image) {
  - __shared_ float region[N];
  - ...}
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads();
  - image[] = cerasin;
- Runtime API
  - Allocating device memory
    void *pyImage = cudaMalloc(bytes);
  - 100 blocks, 10 threads per block
    convolve<<<100, 10>>>(pyImage);

Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

© David Kirk/NVIDIA and Wen-mei H. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes

CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories

Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Contents initialized by host
  - Visible to all threads

A Simple Running Example

Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device

Programming Model: Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory
- Step 1: Matrix Data Transfers

```cpp
// Allocate the device memory where we will copy M to Matrix Md;
Md.width = WIDTH;
Md.height = WIDTH;
Md.pitch = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(Md.elements);
```
Step 2: Matrix Multiplication
A Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P) {
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}

Multiply Using One Thread Block

• One Block of threads compute matrix P
  – Each thread computes one element of P
• Each thread
  – Loads a row of matrix M
  – Loads a column of matrix N
  – Performs one multiply and addition for each pair of M and N elements
  – Compute to off-chip memory access ratio close to 1:1 (not very high)
• Size of matrix limited by the number of threads allowed in a thread block

Step 3: Matrix Multiplication Host-side
Main Program Code

int main(void) {
    // Allocate and initialize the matrices
    Matrix M = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix N = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix P = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
    return 0;
}

Step 3: Matrix Multiplication Host-side (cont.)

// Setup the execution configuration
dim3 dimBlock(WIDTH, WIDTH);
dim3 dimGrid(1, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

// Read P from the device
CopyFromDeviceMatrix(Pd, P);

// Free device matrices
FreeDeviceMatrix(Md);
FreeDeviceMatrix(Nd);
FreeDeviceMatrix(Pd);
}

Step 4: Matrix Multiplication
Device-side Kernel Function

// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
}
Step 4: Matrix Multiplication
Device-Side Kernel Function (cont.)

```c
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * P.pitch + tx] = Pvalue;
```

Step 5: Some Loose Ends

```c
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}
// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}
void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 5: Some Loose Ends (cont.)

```c
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}
// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}
```

Granularity Considerations

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, it can take up to 12 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 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!