CS4402-9535: Many-core Computing with CUDA

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

UWO-CS4402-CS9535
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
GPUs are massively multithreaded manycore chips:
- NVIDIA Tesla products have up to 448 scalar processors with over 12,000 concurrent threads in flight and 1030.4 GFLOPS sustained performance (single precision).
- Users across science & engineering disciplines are achieving 100x or better speedups on GPUs.
CUDA

- CUDA is a scalable parallel programming model and a software environment for parallel computing:
  - Minimal extensions to familiar C/C++ environment
  - Heterogeneous serial-parallel programming model

- GPU Computing with CUDA brings data-parallel computing to the masses
  - as of 2008, over 46,000,000 (100,000,000, as of 2009) CUDA-capable GPUs sold,
  - a developer kit costs about $400 (for 500 GFLOPS).

- Massively parallel computing has become a commodity technology!
CUDA programming and memory models in a nutshell
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
CUDA design goals

- Enable heterogeneous systems (i.e., CPU+GPU)
- Scale to 100's of cores, 1000's of parallel threads
- Use C/C++ with minimal extensions
- Let programmers focus on parallel algorithms
Heterogeneous programming (1/3)

- A CUDA program is a serial program with parallel kernels, all in C.
- The serial C code executes in a host (= CPU) thread
- The parallel kernel C code executes in many device threads across multiple GPU processing elements, called streaming processors (SP).

![Diagram showing CUDA programming model]

(Moreno Maza)
Thus, the parallel code (kernel) is launched and executed on a device by many threads.

- Threads are grouped into thread blocks (more on this soon).
- One kernel is executed at a time on the device.
- Many threads execute each kernel.
Heterogeneous programming (3/3)

- The parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in **thread and block ID variables** are used to map each thread to a specific data tile (more on this soon).

- Thus, each thread executes the same code on different data based on its thread and block ID.
IDs and dimensions (1/2)

- A kernel is a **grid** of **thread blocks**.
- Each thread block has a 2-D ID, which is unique within the grid.
- Each thread has a 2-D ID, which is unique within its thread block.
- The dimensions are set at launch time by the **host code**.
- IDs and dimension sizes are accessed via global variables in the **device code**: threadIdx, blockIdx, ..., blockDim, gridDim.
- Simplify memory addressing when processing multidimensional data.
IDs and dimensions (2/2)
Example: increment array elements (1/2)

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4  -> 4 blocks

int idx = blockDim.x * blockIdx.x + threadIdx.x;

See our example number 4 in /usr/local/cs4402/examples/4

(Moreno Maza)
Example: increment array elements (2/2)

CPU program

```c
void increment_cpu(float *a, float b, int N) {
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}

void main()
{
    .....  
    increment_cpu(a, b, N);
}
```

CUDA program

```c
__global__ void increment_gpu(float *a, float b, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}

void main()
{
    .....  
    .....  
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```
Example host code for increment array elements

```c
// allocate host memory
unsigned int numBytes = N * sizeof(float);
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```
Thread blocks (1/2)

- A **Thread block** is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

- Within a grid, **thread blocks can run in any order**:
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices
Thread blocks (2/2)

- Thus, within a grid, any possible interleaving of blocks must be valid.
- Thread blocks **may coordinate but not synchronize**
  - they may share pointers
  - they should not share locks (this can easily deadlock).
- The fact that thread blocks cannot synchronize gives **scalability**:
  - A kernel scales across any number of parallel cores
- However, within a thread bloc, threads in the same block may synchronize with barriers.
- That is, threads wait at the barrier until threads in the same block reach the barrier.
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Memory hierarchy (1/3)

Host (CPU) memory:
- Not directly accessible by CUDA threads

![Diagram showing the connection between Host memory and Device memory with cudaMemcpy() function]
Global (on the device) memory:
- Also called device memory
- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation
Shared memory:
- Each thread block has its own shared memory, which is accessible only by the threads within that block
- Data lifetime = block lifetime

Local storage:
- Each thread has its own local storage
- Data lifetime = thread lifetime
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Vector addition on GPU (1/4)

Device Code

```c
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
}
```
Vector addition on GPU (2/4)

```c
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>> (d_A, d_B, d_C);
}
```
Vector addition on GPU (3/4)

// allocate and initialize host (CPU) memory
float *h_A = ...;  *h_B = ...;  *h_C = ... (empty)

// allocate device (GPU) memory
float *d_A, *d_B, *d_C;
cudaMalloc((void**)&d_A, N * sizeof(float));
cudaMalloc((void**)&d_B, N * sizeof(float));
cudaMalloc((void**)&d_C, N * sizeof(float));

// copy host memory to device
cudaMemcpy(d_A, h_A, N * sizeof(float),
            cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, N * sizeof(float),
            cudaMemcpyHostToDevice);

// execute grid of N/256 blocks of 256 threads each
dvecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// copy result back to host memory
cudaMemcpy(h_C, d_C, N * sizeof(float),
ciaMemcpyDeviceToHost);

// do something with the result...

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
Code executed on the GPU

- The GPU code defines and calls C function with some restrictions:
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
  - No dynamic polymorphism

- GPU functions must be declared with a qualifier:
  - `__global__`: launched by CPU, cannot be called from GPU, must return void
  - `__device__`: called from other GPU functions, cannot be launched by the CPU
  - `__host__`: can be executed by CPU

- Qualifiers can be combined.

- Built-in variables: `gridDim`, `blockDim`, `blockIdx`, `threadIdx`
Variable Qualifiers (GPU code)

__device__:
- stored in global memory (not cached, high latency)
- accessible by all threads
- lifetime: application

__constant__:
- stored in global memory (cached)
- read-only for threads, written by host
- Lifetime: application

__shared__:
- stored in shared memory (latency comparable to registers)
- accessible by all threads in the same threadblock
- lifetime: block lifetime

Unqualified variables:
- scalars and built-in vector types are stored in registers
- arrays are stored in device (＝ global) memory
Launching kernels on GPU

Launch parameters:

- grid dimensions (up to 2D)
- thread-block dimensions (up to 3D)
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID:
  - Optional, 0 by default

```cpp
dim3 grid(16, 16);
dim3 block(16, 16);
kernel<<<grid, block, 0, 0>>>(...);
kernell<<<32, 512>>>(...);
```
GPU Memory Allocation / Release

Host (CPU) manages GPU memory:

- `cudaMalloc (void ** pointer, size_t nbytes)`
- `cudaMemset (void * pointer, int value, size_t count)`
- `cudaFree (void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```
CUDA Programming Basics

Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - returns after the copy is complete,
  - blocks the CPU thread,
  - doesn’t start copying until previous CUDA calls complete.

- `enum cudaMemcpyKind`
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

- Non-blocking memcopies are provided (more on this later)
```
__global__ void sum_kernel(int *g_input, int *g_output)
{
    extern __shared__ int s_data[ ]; // allocated during kernel launch

    // read input into shared memory
    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    s_data[ threadIdx.x ] = g_input[ idx ];
    __syncthreads( );

    // compute sum for the threadblock
    for ( int dist = blockDim.x/2; dist > 0; dist /= 2 )
    {
        if ( threadIdx.x < dist )
            s_data[ threadIdx.x ] += s_data[ threadIdx.x + dist ];
        __syncthreads( );
    }

    // write the block's sum to global memory
    if ( threadIdx.x == 0 )
        g_output[ blockIdx.x ] = s_data[0];
}
```
Kernel variations and output: what is in a?

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```
Kernel variations and output: answers

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 3 3 3 3

```c
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a = 0, *h_a = 0; // device and host pointers
```c
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes);

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes,
               cudaMemcpyDeviceToHost );
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);
    
    int *d_a=0, *h_a=0; // device and host pointers
    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );
    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
    cudaMemcpy( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
    for(int i=0; i<dimx; i++)
    {
        printf("%d ", h_a[i] );
        printf("\n");
    }
    free( h_a );
    cudaFree( d_a );
    return 0;
}
Example: Shuffling Data

```c
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
    new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    shuffle<<<N/256, 256>>>(d_old, d_new, d_ind);
}
```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx]+1;
}
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix = blockIdx.x*blockDim.x + threadIdx.x;
    int iy = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx] = a[idx] + 1;
}

int main()
{
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a = 0; *h_a = 0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMemcpy( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )
    {
        printf("couldn't allocate memory in");
        return 1;
    }

    cudaMemcpy( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x = dimx / block.x;
    grid.y = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++)
    {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col]);
        printf(\n"n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Blocks Run on Multiprocessors

Kernel launched by host

Device processor array

Device Memory
Streaming processors and multiprocessors
G80 (launched Nov 2006)
128 Thread Processors execute kernel threads
Up to 12,288 parallel threads active
Streaming Multiprocessor (1/2)

- **Processing elements:**
  - 8 scalar thread processors (SP)
  - SM 32 GFLOPS peak at 1.35 GHz
  - 8192 32-bit registers (32KB)
  - usual ops: float, int, branch, ...
Streaming Multiprocessor (2/2)

- **Hardware multithreading:**
  - up to 8 blocks resident at once
  - up to 768 active threads in total

- **16KB on-chip memory:**
  - low latency storage
  - shared among threads of a block
  - supports thread communication
Hardware Multithreading

- **Hardware allocates resources to blocks:**
  - blocks need: thread slots, registers, shared memory
  - blocks don’t run until resources are available

- **Hardware schedules threads:**
  - threads have their own registers
  - any thread not waiting for something can run
  - context switching is free every cycle

- **Hardware relies on threads to hide latency:**
  - thus high parallelism is necessary for performance.
SIMT Thread Execution (1/3)

- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a **warp**
  - The number of threads in a warp is the **warp size** (32 on G80)
  - A half-warp is the first or second half of a warp.
- Within a warp, threads
  - share instruction fetch/dispatch
  - some become inactive when code path diverges
  - hardware automatically handles divergence
- **Warps are the primitive unit of scheduling:**
  - each active block is split into warps in a well-defined way
  - threads within a warp are executed physically in parallel while warps and blocks are executed logically in parallel.
SIMT execution is an implementation choice:
- sharing control logic leaves more space for ALUs
- largely invisible to programmer
- must be understood for performance, not correctness

As already mentioned, each multiprocessor processes batches of blocks, one batch after the other:
- **Active blocks** = the blocks processed by one multiprocessor in one batch
- **Active threads** = all the threads from the active blocks
SIMT Thread Execution (3/3)

- The multiprocessor’s registers and shared memory are split among the active threads.
- Therefore, for a given kernel, the number of active blocks depends on:
  - The number of registers the kernel compiles to
  - How much shared memory the kernel requires
- If there cannot be at least one active block, the kernel fails to launch.
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Thread Synchronization Function

- **void __syncthreads();**

- Synchronizes all threads in a block:
  - once all threads have reached this point, execution resumes normally.
  - this is used to avoid hazards when accessing shared memory.

- Should be used in conditional code only if the condition is uniform across the entire thread block.
GPU Atomic Integer Operations

- Atomic operations on integers in global memory:
  - associative operations on signed/unsigned ints, such as add, min, max, and, or, xor.
  - they have names like atomicAdd, atomicMin, atomicAnd, ...

- Requires hardware with 1.1 compute capability

- Should be used only when strictly necessary: non-locking mechanisms should be preferred for performance consideration.
Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- `cudaThreadSynchronize()`
  - host code execution resumes when all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies (more on this later)
  - ability to overlap memcopies and kernel execution
Example host code (recall)

```c
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**) &d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>> (d_A, b, N);

// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```
Device Management

- CPU can query and select GPU devices:
  - `cudaGetDeviceCount( int* count )`
  - `cudaSetDevice( int device )`
  - `cudaGetDevice( int *current_device )`
  - `cudaGetDeviceProperties( cudaDeviceProp* prop, int device )`
  - `cudaChooseDevice( int *device, cudaDeviceProp* prop )`

- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control one GPU
  - multiple CPU threads can control the same GPU but their calls are serialized by the driver.
  - CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread.
CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except for kernel launches
  - the error code type is `cudaError_t`
- `cudaError_t cudaGetLastError(void):`
  - returns the code for the last error (no error has also a code)
- `char* cudaGetErrorString(cudaError_t code):`
  - returns a null-terminated character string describing the error

```c
printf("%s\n, cudaGetErrorString( cudaGetLastError() ) );
```
CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed

```c
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernelf<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start); cudaEventDestroy(stop);
```
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
The `nvcc` compiler

- Any source file containing CUDA language extensions must be compiled with `nvcc`:
  - NVCC separates code running on the host from code running on the device.

- **Two-stage compilation:**
  - First generates *Parallel Thread eXecution code* (PTX)
  - Then produces Device-specific binary object

- NVCC is a **compiler driver:**
  - Works by invoking all the necessary tools and compilers like `cudacc`, `g++`,

- An executable with CUDA code requires:
  - the CUDA core library (cuda)
  - the CUDA runtime library (cudart)
Compiling CUDA code

1. C/C++ CUDA Application
2. NVCC
3. PTX Code
4. PTX to Target Compiler
5. Target code
6. Physical
7. GPU
8. G80

Virtual

CPU Code
PTX Example (SAXPY code)

cvt.u32.u16 $blockid, %ctaid.x;    // Calculate i from thread/block IDs
cvt.u32.u16 $blocksize, %ntid.x;
cvt.u32.u16 $tid, %tid.x;
mad24.lo.u32 $i, $blockid, $blocksize, $tid;
ld.param.u32 $n, [N];              // Nothing to do if n ≤ i
setp.le.u32 $p1, $n, $i;
@$p1 bra $L_finish;

mul.lo.u32 $offset, $i, 4;        // Load y[i]
ld.param.u32 $yaddr, [Y];
add.u32 $yaddr, $yaddr, $offset;
ld.global.f32 $y_i, [$yaddr+0];
ld.param.u32 $xaddr, [X];         // Load x[i]
add.u32 $xaddr, $xaddr, $offset;
ld.global.f32 $x_i, [$xaddr+0];

ld.param.f32 $alpha, [ALPHA];    // Compute and store alpha*x[i] + y[i]
mad.f32 $y_i, $alpha, $x_i, $y_i;
st.global.f32 [$yaddr+0], $y_i;

$L_finish: exit;
Debugging CUDA code

- An executable compiled in **device emulation mode** (nvcc -deviceemu) runs completely on the host using the CUDA runtime:
  - no need of any device and CUDA driver
  - each device thread is emulated with a host thread

- However, the device emulation mode has several pitfalls:
  - emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads potentially produce different results.
  - results of floating-point computations will slightly differ because of different compiler outputs, different instruction sets, etc.
  - dereferencing device pointers on the host may produce correct results in device emulation mode while generating errors in device execution mode

- In fact in the latest version of nvcc the device emulation mode is **no longer supported**!
Developing a CUDA program

1. Decompose the targeted application according to the many-core programming model of CUDA:
   - such a program alternates serial code and vectorized code
   - such that the parallel code has enough work and enough parallelism

2. Write serial C code for each targeted CUDA kernel

3. For each targeted CUDA kernel, carefully decompose the work into thread blocks:
   - this implies mapping the thread blocks to the data
   - leading to potentially delicate index calculation:
     proving them mathematically often prevents from painful debugging!

4. Verify each kernel against its C counterpart

5. Debugging may lead to further decompose a kernel into smaller kernels.
Plan

1. GPUs and CUDA: a Brief Introduction
2. CUDA Programming Model
3. CUDA Memory Model
4. CUDA Programming Basics
5. CUDA Hardware Implementation
6. CUDA Programming: Scheduling and Synchronization
7. CUDA Tools
8. Sample Programs
Matrix multiplication (1/16)

- The goals of this example are:
  - Understanding how to write a kernel for a non-toy example
  - Understanding how to map work (and data) to the thread blocks
  - Understanding the importance of using shared memory

- We start by writing a naive kernel for matrix multiplication which does not use shared memory.

- Then we analyze the performance of this kernel and realize that it is limited by the global memory latency.

- Finally, we present a more efficient kernel, which takes advantage of a tile decomposition and makes use of shared memory.
Consider multiplying two rectangular matrices $A$ and $B$ with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.

Principle: each thread computes an element of $C$ through a 2D kernel.
Matrix multiplication (3/16)

```c
__global__ void mat_mul(float *a, float *b, 
                        float *ab, int width)
{
    // calculate the row & col index of the element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    float result = 0;
    // do dot product between row of a and col of b
    for(int k = 0; k < width; ++k)
        result += a[row*width+k] * b[k*width+col];
    ab[row*width+col] = result;
}
```
Sample Programs

Matrix multiplication (4/16)

- Analyze the previous CUDA kernel for multiplying two rectangular matrices $A$ and $B$ with respective formats $m \times n$ and $n \times p$. Define $C = A \times B$.
- Each element of $C$ is computed by one thread:
  - then each row of $A$ is read $p$ times and
  - each column of $B$ is read $m$ times, thus
  - **2 m n p reads in total for 2 m n p flops.**
- Let $t$ be an integer dividing $m$ and $p$. We decompose $C$ into $t \times t$ tiles. If tiles are computed one after another, then:
  - $(m/t)(t n)(p/t)$ slots are read in $A$
  - $(p/t)(t n)(m/t)$ slots are read in $A$, thus
  - **2m n p/t reads in total for 2 m n p flops.**
- For a CUDA implementation, $t = 16$ such that each tile is computed by one thread block.
Matrix multiplication (5/16)

- The previous explanation can be adapted to a particular GPU architecture, so as to estimate the performance of the first (naive) kernel.

- The first kernel has a **global memory access to flop ratio** (GMAC) of 8 Bytes / 2 ops, that is, 4 B/op.

- Suppose using a GeForce GTX 260, which has 805 GFLOPS peak performance.

- In order to reach **peak fp performance** we would need a memory bandwidth of $\text{GMAC} \times \text{Peak FLOPS} = 3.2 \text{ TB/s}$.

- Unfortunately, we only have 112 GB/s of actual **memory bandwidth** (BW) on a GeForce GTX 260.

- Therefore an upper bound on the performance of our implementation is $\text{BW} / \text{GMAC} = 28 \text{ GFLOPS}$. 
Matrix multiplication (6/16)

- The picture below illustrates our second kernel.
- Each thread block computes a tile in $C$, which is obtained as a dot product of tile-vector of $A$ by a tile-vector of $B$.
- Tile size is chosen in order to maximize data locality.
Matrix multiplication (7/16)

- So a thread block computes a $t \times t$ tile of $C$.
- Each element in that tile is a dot-product of a row from $A$ and a column from $B$.
- We view each of these dot-products as a sum of small dot products:

$$c_{i,j} = \sum_{k=0}^{t-1} a_{i,k} b_{k,j} + \sum_{k=t}^{2t-1} a_{i,k} b_{k,j} + \cdots + \sum_{k=n-1-t}^{n-1} a_{i,k} b_{k,j}$$

- Therefore we fix $\ell$ and then compute $\sum_{k=\ell t}^{(\ell+1)t-1} a_{i,k} b_{k,j}$ for all $i,j$ in the working thread block.
- We do this for $\ell = 0, 1, \ldots, (n/t - 1)$.
- This allows us to store the working tiles of $A$ and $B$ in shared memory.
Matrix multiplication (8/16)

- We assume that $A$, $B$, $C$ are stored in row-major layout.
- Observe that for computing a tile in $C$ our kernel code does need to know the number of rows in $A$.
- It just needs to know the width (number of columns) of $A$ and $B$.
- The following code fragments are taken from Example 2.

```c
#define BLOCK_SIZE 16

template <typename T>
__global__ void matrix_mul_ker(T* C, const T *A, const T *B,
   size_t wa, size_t wb)

   // Block index; WARNING: should be at most 2^16 - 1
   int bx = blockIdx.x; int by = blockIdx.y;

   // Thread index
   int tx = threadIdx.x; int ty = threadIdx.y;
```
We need the position in A of the first element of the first working tile from A; we call it aBegin.

We will need also the position in A of the last element of the last working tile from A; we call it aEnd.

Moreover, we will need the offset between two consecutive working tiles of A; we call it aStep.

```c
int aBegin = wa * BLOCK_SIZE * by;
int aEnd = aBegin + wa - 1;
int aStep = BLOCK_SIZE;
```
Matrix multiplication (10/16)

- Similarly for $B$ we have $b\text{Begin}$ and $b\text{Step}$.
- We will not need a $b\text{End}$ since once we are done with a row of $A$, we are also done with a column of $B$.
- Finally, we initially the accumulator of the working thread; we call it $C_{\text{sub}}$.

```c
int bBegin = BLOCK_SIZE * bx;

int bStep = BLOCK_SIZE * wb;

int Csub = 0;
```
Matrix multiplication (11/16)

- The main loop starts by copying the working tiles of $A$ and $B$ to shared memory.

```c
for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
    // shared memory for the tile of A
    __shared__ int As[BLOCK_SIZE][BLOCK_SIZE];

    // shared memory for the tile of B
    __shared__ int Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the tiles from global memory to shared memory
    // each thread loads one element of each tile
    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();
}
```
Compute a small “dot-product” for each element in the working tile of C.

```c
// Multiply the two tiles together
// each thread computes one element of the tile of C
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 tiles of A and B in the next iteration
__syncthreads();
```
Once computed, the working tile of $C$ is written to global memory.

```c
// Write the working tile of $C$ to global memory;
// each thread writes one element
int c = wb * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wb * ty + tx] = Csub;
```
Matrix multiplication (14/16)

- Each thread block should have many threads:
  - TILE_WIDTH = 16 implies $16 \times 16 = 256$ threads

- There should be many thread blocks:
  - A $1024 \times 1024$ matrix would require 4096 thread blocks.
  - Since one streaming multiprocessor (SM) can handle 768 threads, each SM will process 3 thread blocks, leading it full occupancy.

- Each thread block performs $2 \times 256$ reads of a 4-byte float while performing $256 \times (2 \times 16) = 8,192$ fp ops:
  - Memory bandwidth is no longer limiting factor
Matrix multiplication (15/16)

- Experimentation performed on a GT200.
- **Tiling** and using **shared memory** were clearly worth the effort.

![Graph showing GFLOPS vs TILE_SIZE](image-url)
Matrix multiplication (16/16)

- Effective use of different memory resources reduces the number of accesses to global memory.
- But these resources are finite!
- The more memory locations each thread requires, the fewer threads an SM can accommodate.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per GT200 SM</th>
<th>Full Occupancy on GT200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>16384</td>
<td>&lt;= 16384 / 768 threads = 21 per thread</td>
</tr>
<tr>
<td><strong>shared</strong> Memory</td>
<td>16KB</td>
<td>&lt;= 16KB / 8 blocks = 2KB per block</td>
</tr>
</tbody>
</table>