CS 410/510 General Purpose GPU Computing

Karavanic
CUDA Threads
CUDA C /OpenCL – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

```c
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n* sizeof(float);
    float* d_A, d_B, d_C;
    ...
    1. // Allocate device memory for A, B, and C
        // copy A and B to device memory

    2. // Kernel launch code – to have the device
        // to perform the actual vector addition

    3. // copy C from the device memory
        // Free device vectors
}

CPU

Host Memory

GPU

Part 1

Device Memory

Part 2

Part 3
Partial Overview of CUDA Memories

- **Device code can:**
  - R/W per-thread **registers**
  - R/W all-shared **global memory**

- **Host code can**
  - Transfer data to/from per grid **global memory**
CUDA Device Memory Management API functions

- **cudaMalloc()**
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object in terms of bytes

- **cudaFree()**
  - Frees object from device global memory
    - **Pointer** to freed object
Host-Device Data Transfer API functions

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float* d_A, d_B, d_C;

    1. // Transfer A and B to device memory
        cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Allocate device memory for
    cudaMalloc((void **) &d_C, size);

    2. // Kernel invocation code – to be shown later
        ...

    3. // Transfer C from device to host
        cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
Example: Vector Addition Kernel

Device Code

```c
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}

int vectAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);
}
```
Example: Vector Addition Kernel

// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<n) C_d[i] = A_d[i] + B_d[i];
}

int vecAdd(float* h_A, float* h_B, float* h_C, int n) {
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
More on Kernel Launch

```c
int vecAdd(float* A, float* B, float* C, int n) {
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    dim3 DimGrid(n/256, 1, 1);
    if (n%256) DimGrid.x++;
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
__host__
Void vecAdd()
{
    dim3 DimGrid = (ceil(n/256.0),1,1);
    dim3 DimBlock = (256,1,1);
    vecAddKernel<<<DimGrid,DimBlock>>>(A_d,B_d,C_d,n);
}

__global__
void vecAddKernel(float *A_d,
                      float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x
           + threadIdx.x;
    if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
More on CUDA Function Declarations

| __device__ float DeviceFunc() | Executed on the: device | Only callable from the: device |
| __global__ void KernelFunc() | Executed on the: device | Only callable from the: host |
| __host__ float HostFunc() | Executed on the: host | Only callable from the: host |

- __global__ defines a kernel function
- Each “__” consists of two underscore characters
- A kernel function must return void
- __device__ and __host__ can be used together
A Multi-Dimensional Grid Example

Kernel 1

Grid 1
- Block (0, 0)
- Block (0, 1)
- Block (1, 0)
- Block (1, 1)

Grid 2
- Block (1, 1)

- Thread (0,0,0)
- Thread (0,0,1)
- Thread (0,0,2)
- Thread (0,0,3)
- Thread (0,1,0)
- Thread (0,1,1)
- Thread (0,1,2)
- Thread (0,1,3)

Kernel 2
Processing a Picture with a 2D Grid

16×16 blocks
Row-Major Layout in C/C++

Row * Width + Col = 2 * 4 + 1 = 9
Source Code of the PictureKernel

```c
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n,int m) {

    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    // each thread computes one element of d_Pout if in range
    if ((Row < m) && (Col < n)) {
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
    }
}
```
Figure 4.5 Covering a 76×62 picture with 16×blocks.
A Simple Running Example
Matrix Multiplication

• A simple illustration of the basic features of memory and thread management in CUDA programs
  – Thread index usage
  – Memory layout
  – Register usage
  – Assume square matrix for simplicity
  – Leave shared memory usage until later
Square Matrix-Matrix Multiplication

- $P = M \times N$ of size $\text{WIDTH} \times \text{WIDTH}$
  - Each \textbf{thread} calculates one element of $P$
  - Each row of $M$ is loaded $\text{WIDTH}$ times from global memory
  - Each column of $N$ is loaded $\text{WIDTH}$ times from global memory
Row-Major Layout in C/C++

Row*Width+Col = 2*4+1 = 9
Matrix Multiplication
A Simple Host Version in C

// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
Kernel Function - A Small Example

• Have each 2D thread block to compute a \((\text{TILE}\_\text{WIDTH})^2\) sub-matrix (tile) of the result matrix
  – Each has \((\text{TILE}\_\text{WIDTH})^2\) threads
• Generate a 2D Grid of \((\text{WIDTH}/\text{TILE}\_\text{WIDTH})^2\) blocks

WIDTH = 4; TILE\_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE\_WIDTH = 2
Use 2*2 = 4 blocks
**A Slightly Bigger Example**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀₀</td>
<td>P₀₁</td>
<td>P₀₂</td>
<td>P₀₃</td>
<td>P₀₄</td>
<td>P₀₅</td>
<td>P₀₆</td>
<td>P₀₇</td>
</tr>
<tr>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>P₁₃</td>
<td>P₁₄</td>
<td>P₁₅</td>
<td>P₁₆</td>
<td>P₁₇</td>
</tr>
<tr>
<td>P₂₀</td>
<td>P₂₁</td>
<td>P₂₂</td>
<td>P₂₃</td>
<td>P₂₄</td>
<td>P₂₅</td>
<td>P₂₆</td>
<td>P₂₇</td>
</tr>
<tr>
<td>P₃₀</td>
<td>P₃₁</td>
<td>P₃₂</td>
<td>P₃₃</td>
<td>P₃₄</td>
<td>P₃₅</td>
<td>P₃₆</td>
<td>P₃₇</td>
</tr>
<tr>
<td>P₄₀</td>
<td>P₄₁</td>
<td>P₄₂</td>
<td>P₄₃</td>
<td>P₄₄</td>
<td>P₄₅</td>
<td>P₄₆</td>
<td>P₄₇</td>
</tr>
<tr>
<td>P₅₀</td>
<td>P₅₁</td>
<td>P₅₂</td>
<td>P₅₃</td>
<td>P₅₄</td>
<td>P₅₅</td>
<td>P₅₆</td>
<td>P₅₇</td>
</tr>
<tr>
<td>P₆₀</td>
<td>P₆₁</td>
<td>P₆₂</td>
<td>P₆₃</td>
<td>P₆₄</td>
<td>P₆₅</td>
<td>P₆₆</td>
<td>P₆₇</td>
</tr>
<tr>
<td>P₇₀</td>
<td>P₇₁</td>
<td>P₇₂</td>
<td>P₇₃</td>
<td>P₇₄</td>
<td>P₇₅</td>
<td>P₇₆</td>
<td>P₇₇</td>
</tr>
</tbody>
</table>

WIDTH = 8; TILE_WIDTH = 2
Each block has 2*2 = 4 threads

WIDTH/TILE_WIDTH = 4
Use 4* 4 = 16 blocks
A Slightly Bigger Example (cont.)

<table>
<thead>
<tr>
<th></th>
<th>P₀₀</th>
<th>P₀₁</th>
<th>P₀₂</th>
<th>P₀₃</th>
<th>P₀₄</th>
<th>P₀₅</th>
<th>P₀₆</th>
<th>P₀₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁₀</td>
<td>P₁₁</td>
<td>P₁₂</td>
<td>P₁₃</td>
<td>P₁₄</td>
<td>P₁₅</td>
<td>P₁₆</td>
<td>P₁₇</td>
<td></td>
</tr>
<tr>
<td>P₂₀</td>
<td>P₂₁</td>
<td>P₂₂</td>
<td>P₂₃</td>
<td>P₂₄</td>
<td>P₂₅</td>
<td>P₂₆</td>
<td>P₂₇</td>
<td></td>
</tr>
<tr>
<td>P₃₀</td>
<td>P₃₁</td>
<td>P₃₂</td>
<td>P₃₃</td>
<td>P₃₄</td>
<td>P₃₅</td>
<td>P₃₆</td>
<td>P₃₇</td>
<td></td>
</tr>
<tr>
<td>P₄₀</td>
<td>P₄₁</td>
<td>P₄₂</td>
<td>P₄₃</td>
<td>P₄₄</td>
<td>P₄₅</td>
<td>P₄₆</td>
<td>P₄₇</td>
<td></td>
</tr>
<tr>
<td>P₅₀</td>
<td>P₅₁</td>
<td>P₅₂</td>
<td>P₅₃</td>
<td>P₅₄</td>
<td>P₅₅</td>
<td>P₅₆</td>
<td>P₅₇</td>
<td></td>
</tr>
<tr>
<td>P₆₀</td>
<td>P₆₁</td>
<td>P₆₂</td>
<td>P₆₃</td>
<td>P₆₄</td>
<td>P₆₅</td>
<td>P₆₆</td>
<td>P₆₇</td>
<td></td>
</tr>
<tr>
<td>P₇₀</td>
<td>P₇₁</td>
<td>P₇₂</td>
<td>P₇₃</td>
<td>P₇₄</td>
<td>P₇₅</td>
<td>P₇₆</td>
<td>P₇₇</td>
<td></td>
</tr>
</tbody>
</table>

WIDTH = 8; TILE_WIDTH = 4
Each block has 4*4 = 16 threads

WIDTH / TILE_WIDTH = 2
Use 2* 2 = 4 blocks
// Setup the execution configuration
// TILE_WIDTH is a #define constant
   dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

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

// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{

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

Work for Block (0,0)
in a TILE_WIDTH = 2 Configuration

Col = 0 \times 2 + threadIdx.x
Row = 0 \times 2 + threadIdx.y

Row = 0
Row = 1
Work for Block (0,1)

Col = 1 \times 2 + threadIdx.x
Row = 0 \times 2 + threadIdx.y

Row = 0
Row = 1
A Simple Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **1024** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have **thread index** numbers within block
  - Kernel code uses **thread index and block index** to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

Courtesy: John Nickolls, NVIDIA
History of parallelism

• 1\textsuperscript{st} gen - Instructions are executed sequentially in program order, one at a time.

• Example:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
</tr>
</tbody>
</table>
History - Cont’d

• 2\textsuperscript{nd} gen - Instructions are executed sequentially, in program order, in an assembly line fashion. (pipeline)

• Example:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction3</td>
<td></td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
</tbody>
</table>
History –
Instruction Level Parallelism

- 3\textsuperscript{rd} gen - Instructions are executed in parallel
- Example code 1:
\[
\begin{align*}
  c &= b + a; \\
  d &= c + e;
\end{align*}
\]
\text{Non-parallelizable}
- Example code 2:
\[
\begin{align*}
  a &= b + c; \\
  d &= e + f;
\end{align*}
\]
\text{Parallelizable}
Instruction Level Parallelism (Cont.)

- Two forms of ILP:
  - Superscalar: At runtime, fetch, decode, and execute multiple instructions at a time. Execution may be out of order
  
<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction1</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>Instruction3</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
<tr>
<td>Instruction4</td>
<td></td>
<td>Fetch</td>
<td>Decode</td>
<td>Execute</td>
<td>Memory</td>
</tr>
</tbody>
</table>

  - VLIW: At compile time, pack multiple, independent instructions in one large instruction and process the large instructions as the atomic units.
History – Cont’d

- **4th gen** – Multi-threading: multiple threads are executed in an alternating or simultaneous manner on the same processor/core.
- **5th gen** - Multi-Core: Multiple threads are executed simultaneously on multiple processors
Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
Example: Executing Thread Blocks

- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to 8 blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.

- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution
The Von-Neumann Model
The Von-Neumann Model with SIMD units

- Memory
- Control Unit
- Processing Unit
  - ALU
  - Reg File
- I/O
- PC
- IR
Example: Thread Scheduling

• Each Block is executed as 32-thread Warps
  – An implementation decision, not part of the CUDA programming model
  – Warps are scheduling units in SM
• If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  – Each Block is divided into 256/32 = 8 Warps
  – There are 8 * 3 = 24 Warps
Going back to the program

• Every instruction needs to be fetched from memory, decoded, then executed.

• Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.

• An example instruction cycle is the following:

  Fetch | Decode | Execute | Memory
Operate Instructions

• Example of an operate instruction:
  ADD R1, R2, R3

• Instruction cycle for an operate instruction:
  Fetch | Decode | Execute | Memory
Data Transfer Instructions

• Examples of data transfer instruction:
  LDR R1, R2, #2
  STR R1, R2, #2

• Instruction cycle for an operate instruction:
  Fetch | Decode | Execute | Memory
Control Flow Operations

• Example of control flow instruction:
  
  BRp #-4
  
  if the condition is positive, jump back four instructions

• Instruction cycle for an arithmetic instruction:
  
  Fetch | Decode | Execute | Memory
How thread blocks are partitioned

• Thread blocks are partitioned into warps
  – Thread IDs within a warp are consecutive and increasing
  – Warp 0 starts with Thread ID 0

• Partitioning is always the same
  – Thus you can use this knowledge in control flow
  – However, the exact size of warps may change from generation to generation
  – (Covered next)

• However, DO NOT rely on any ordering between warps
  – If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).
Main performance concern with branching is divergence
- Threads within a single warp take different paths
- Different execution paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.

A common case: avoid divergence when branch condition is a function of thread ID
- Example with divergence:
  - If (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (threadIdx.x / WARP_SIZE > 2) { }
  - Also creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - At any time, 1 or 2 of the warps is executed by SM
  - 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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
<th>TB1 W1</th>
<th>TB2 W1</th>
<th>TB3 W1</th>
<th>TB1 W2</th>
<th>TB2 W2</th>
<th>TB3 W2</th>
<th>TB1 W3</th>
<th>TB1 W2</th>
<th>TB3 W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TB = Thread Block, W = Warp
Block Granularity Considerations

For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?

- For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, there are 24 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 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.

- For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM. Also, this works for CUDA 3.0 and beyond but too large for some early CUDA versions.
These slides include materials related to the Course Textbook and © David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2012  ECE408/CS483, University of Illinois, Urbana-Champaign