ECE498AL

Lecture 3: A Simple Example, Tools, and CUDA Threads
A Simple Running Example
Matrix Multiplication

• A simple 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
  – Assume square matrix for simplicity
Programming Model:
Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory
Memory Layout of a Matrix in C

\[
\begin{array}{cccc}
M_{0,0} & M_{1,0} & M_{2,0} & M_{3,0} \\
M_{0,1} & M_{1,1} & M_{2,1} & M_{3,1} \\
M_{0,2} & M_{1,2} & M_{2,2} & M_{3,2} \\
M_{0,3} & M_{1,3} & M_{2,3} & M_{3,3}
\end{array}
\]
Step 1: 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;
        }
}
Step 2: Input Matrix Data Transfer
(Host-side Code)

```c
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;

    // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
    cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);
}
```
Step 3: Output Matrix Data Transfer

(Host-side Code)

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

3. // Read P from the device
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
Step 4: Kernel Function

// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {

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

Step 4: Kernel Function (cont.)

for (int k = 0; k < Width; ++k) {
    float Melement = Md[threadIdx.y*Width+k];
    float Nelement = Nd[k*Width+threadIdx.x];
    Pvalue += Melement * Nelement;
}

Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
Step 5: Kernel Invocation
(Host-side Code)

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

// Launch the device computation threads!
MatrixMulKernel<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
Only One Thread Block Used

- One Block of threads compute matrix $P_d$
  - Each thread computes one element of $P_d$
- Each thread
  - Loads a row of matrix $M_d$
  - Loads a column of matrix $N_d$
  - Perform one multiply and addition for each pair of $M_d$ and $N_d$ 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign
Step 7: Handling Arbitrary Sized Square Matrices (will cover later)

- Have each 2D thread block to compute a \((TILE \_ WIDTH)^2\) sub-matrix (tile) of the result matrix
  - Each has \((TILE \_ WIDTH)^2\) threads

- Generate a 2D Grid of \((WIDTH/TILE \_ WIDTH)^2\) blocks

You still need to put a loop around the kernel call for cases where \(WIDTH/TILE \_ WIDTH\) is greater than max grid size (64K)!
Some Useful Information on Tools
Compiling a CUDA Program

C/C++ CUDA Application

NVCC

CPU Code

PTX Code

Virtual

Physical

PTX to Target Compiler

G80

GPU

Target code

• Parallel Thread eXecution (PTX)
  – Virtual Machine and ISA
  – Programming model
  – Execution resources and state

float4 me = gx[gtid];
me.x += me.y * me.z;

ld.global.v4.f32 {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32 $f1, $f5, $f3, $f1;
Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime
Linking

• Any executable with CUDA code requires two dynamic libraries:
  – The CUDA runtime library (**cudart**)
  – The CUDA core library (**cuda**)
Debugging Using the Device Emulation Mode

• An executable compiled in device emulation mode (\texttt{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

• Running in device emulation mode, one can:
  – Use host native debug support (breakpoints, inspection, etc.)
  – Access any device-specific data from host code and vice-versa
  – Call any host function from device code (e.g. \texttt{printf}) and vice-versa
  – Detect deadlock situations caused by improper usage of \texttt{__syncthreads}
Device Emulation Mode Pitfalls

• Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.

• Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.
Floating Point

• Results of floating-point computations will slightly differ because of:
  – Different compiler outputs, instruction sets
  – Use of extended precision for intermediate results
    • There are various options to force strict single precision on the host
CUDA Threads
Block IDs and Thread IDs

- Each thread uses IDs to 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
  - …
Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size
A Small Example

Block(0,0)  Block(1,0)

\[
\begin{array}{|c|c|}
\hline
P_{0,0} & P_{1,0} \\
\hline
P_{0,1} & P_{1,1} \\
\hline
P_{0,2} & P_{1,2} \\
\hline
P_{0,3} & P_{1,3} \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
P_{2,0} & P_{3,0} \\
\hline
P_{2,1} & P_{3,1} \\
\hline
P_{2,2} & P_{3,2} \\
\hline
P_{2,3} & P_{3,3} \\
\hline
\end{array}
\]

TILE_WIDTH = 2

Block(0,1)  Block(1,1)
A Small Example: Multiplication
Revised Matrix Multiplication Kernel using Multiple Blocks

```c
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width) {
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[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 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id 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
Transparent Scalability

- Hardware is free to assigns 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.
G80 Example: Executing Thread Blocks

• Threads are assigned to Streaming Multiprocessors in block granularity
  – Up to 8 blocks to each SM as resource allows
  – SM in G80 can take up to 768 threads
    • Could be 256 (threads/block) * 3 blocks
    • Or 128 (threads/block) * 6 blocks, etc.

• Threads run concurrently
  – SM maintains thread/block id #s
  – SM manages/schedules thread execution
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
G80 Example: Thread Scheduling (Cont.)

• SM implements zero-overhead warp scheduling
  – At any time, only one 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign
G80 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 768 threads, there are 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!
Some Additional API Features
Application Programming Interface

- The API is an **extension to the C programming language**
- It consists of:
  - **Language extensions**
    - To target portions of the code for execution on the device
  - **A runtime library** split into:
    - A **common component** providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A **host component** to control and access one or more devices from the host
    - A **device component** providing device-specific functions
Language Extensions: Built-in Variables

• **dim3** `gridDim`;
  – Dimensions of the grid in blocks (`gridDim.z` unused)

• **dim3** `blockDim`;
  – Dimensions of the block in threads

• **dim3** `blockIdx`;
  – Block index within the grid

• **dim3** `threadIdx`;
  – Thread index within the block
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. $\_\_\sin(x)$)
  - $\_\_\text{pow}$
  - $\_\_\log, \_\_\log2, \_\_\log10$
  - $\_\_\exp$
  - $\_\_\sin, \_\_\cos, \_\_\tan$
Host Runtime Component

• Provides functions to deal with:
  – Device management (including multi-device systems)
  – Memory management
  – Error handling

• Initializes the first time a runtime function is called

• A host thread can invoke device code on only one device
  – Multiple host threads required to run on multiple devices
Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in 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