COMP 605: Introduction to Parallel Computing
Lecture : CUDA Matrix-Matrix Multiplication

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 04/25/17
Last Update: 04/25/17

Table of Contents

Matrix-Matrix Multiplication (Mat-Mat-Mult)

2D Matrix-Matrix Multiplication (Mat-Mat-Mult)

```
/* Serial_matrix_mult */
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < n; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
   printf(... )
}
```

Where:
A is an $[m \, x \, k]$ matrix
B is a $[k \, x \, n]$
C is a matrix with the dimensions $[m \, x \, n]$

2D Matrix-Matrix Multiplication (Mat-Mat-Mult)

Definition: Let A be an $[m\, x\, k]$ matrix, and B be a be an $[k\, x\, n]$, then C will be a matrix with the dimensions $[m\, x\, n]$.

Then $AB = \lfloor c_{ij} \rfloor$, and

$$c_{ij} = \sum_{t=1}^{k} a_{it} b_{tj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{k1} b_{kj}$$

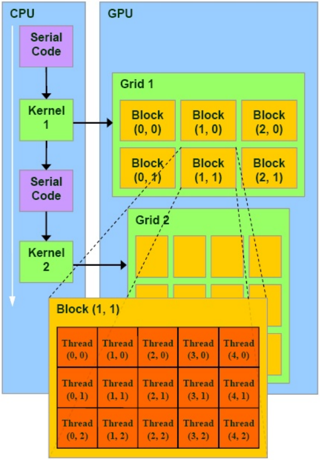
$$= \begin{bmatrix} a_{00} & \dots & a_{0j} & \dots & a_{0,k-1} \\ & & \dots & & \\ \mathbf{a_{i0}} & \dots & a_{ij} & \dots & a_{i,k-1} \\ & & \dots & & \\ a_{m-1,0} & \dots & a_{m-1,j} & \dots & a_{m-1,k-1} \end{bmatrix} \bullet \begin{bmatrix} b_{00} & \dots & \mathbf{b_{0j}} & \dots & b_{0,n-1} \\ & & \dots & & \\ b_{i0} & \dots & \mathbf{b_{ij}} & \dots & b_{i,n-1} \\ & & \dots & & \\ b_{k-1,1} & \dots & \mathbf{b_{kj}} & \dots & b_{n-1,p-1} \end{bmatrix}$$

$$= \begin{bmatrix} c_{00} & \dots & c_{1j} & \dots & c_{1,n-1} \\ & & \dots & & \\ c_{i0} & \dots & \mathbf{c_{ij}} & \dots & c_{i,n-1} \\ & & \dots & & \\ c_{m-1,0} & \dots & c_{mj} & \dots & c_{m-1,n-1} \end{bmatrix}$$

$$c_{12} = a_{11} b_{12} + a_{12} b_{22} + a_{13} b_{32}$$

Recall: Defining GPU Threads and Blocks

- Looking at Device: Nvidia Tesla C1060
- Kernels run on GPU threads
- Grid: organized as 2D array of blocks:
    - Maximum sizes of each dimension: $[gridDim.x \; x \; gridDim.y \; x \; gridDim.z]$ $= (65,536 \; x \; 65,536 \; x \; 1)$ blocks
- Block: $3D$ collection of threads
    - Max threads per block: 512
    - Max thread dimensions: (512, 512, 64) $[blockDim.x * blockDim.y * blockDim.z]$ $MaxThds/Block \leq 1024$
- threads composing a thread block must:
    - execute the same kernel
    - share data: issued to the same core
    - *Warp*: group of 32 threads; min size of data processed in SIMD fashion by CUDA multiprocessor.

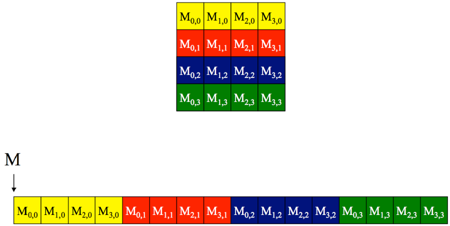


Source: http://hothardware.com/Articles/NVIDIA-GF100-Architecture-and-Feature-Preview

Matrix Mult: linear mapping of a 2D matrix in C.

- CUDA does not allow run-time allocation of a 2D matrix
- C memory mapping is *row − major* order.
- Index for accessing matrix M in the inner loop: $M[\, i \times width + k \,]$
- Need to linearize the array in *row − major* order, into a vector which can be dynamic.

Memory Layout of a Matrix in C

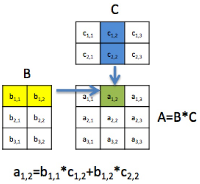


- 1 D array, where Element[row][col] is element [row*width+col]
- Thread mapping: $intx = threadIdx.x + blockIdx.x * blockDim.x;$

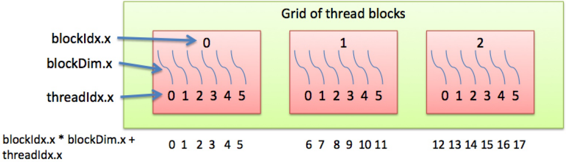
Mapping Serial Code to Threads

Map this code:

```
for (i = 0; i < n; ++i)
  for (j = 0; j < m; ++j)
    for (k = 0; k < p; ++k)
      a[i+n*j] += b[i+n*k] * c[k+p*j];
```



into this (logical) architecture:

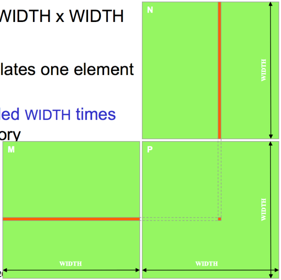


Source: `http://www.hpcwire.com/features/Compilers_and_More_Optimizing_GPU_Kernels.html`

Programming Model

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



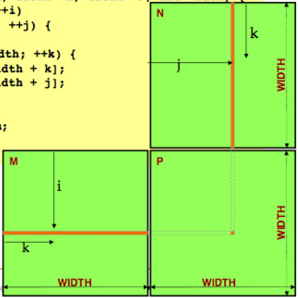
$P = M \times N$ is a dot product
Each dot product is independent of all the others.

Matrix Mult: Serial C code (K&H)

Calculating $P[i][j] = P[i][j] + M[i][k] \times N[k][j]$

A Simple Host Version in C

```
void MatrixMulOnHost( float* M, float* N, float* P, int Width) {
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
       float sum = 0;
       for (int k = 0; k < Width; ++k) {
           float a = M[i * Width + k];
           float b = N[k * Width + j];
           sum += a * b;
       }
       P[i * Width + j] = sum;
    }
}
```



Adapted From:
David Kirk/NVIDIA and Wen-mei W. Hwu, UIUC

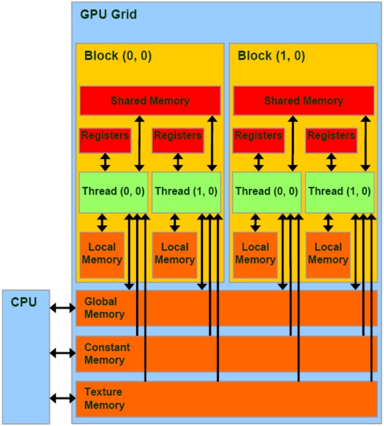
Matrix-Multiplication Algorithm for GPU/CUDA Host.

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

Host uses CudaMalloc to allocate memory on the device *globalmemory* space.

Cuda Device Memory Model

- Host and device have separate memories.
- Host can only copy to/from *global memory* and *constant memory*
  - *cudaMalloc*()
  - *cudaFree*()
  - *cudaMemcpy*()



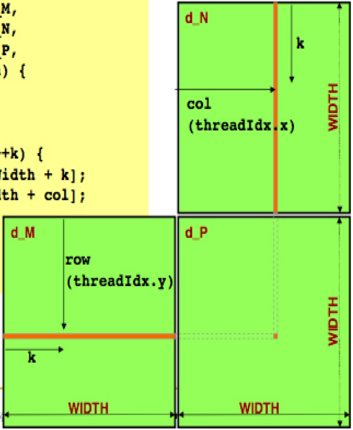
Matrix-Multiplication Algorithm for GPU/CUDA Host.

Host code showing cudaMalloc calls.

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

1. // Transfer M and N to device memory
   cudaMalloc((void**) &Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer P from device to host
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

GPU Kernel Function

```
__global__
void MatrixMulKernel(float* d_M,
                     float* d_N,
                     float* d_P,
                     int Width) {
  int row = threadIdx.y;
  int col = threadIdx.x;
  float P_val = 0;
  for (int k = 0; k < Width; ++k) {
    float M_elem = d_M[row * Width + k];
    float N_elem = d_N[k * Width + col];
    P_val += M_elem * N_elem;
  }
  d_p[row*Width+col] = P_val;
}
```



Adapted From:
David Kirk/NVIDIA and Wen-mei W. Hwu, UIUC

Current code only uses threadIdnx, so can only use 1 block.

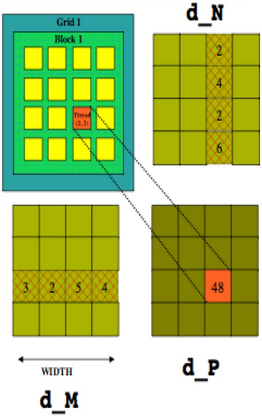
Kernel Invocation and Copy Results

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


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N, d_P,
Width);

// Copy back the results from device to host
cudaMemcpy(P, d_P, matrix_size, cudaMemcpyDeviceToHost);

// Free up the device memory matrices
cudaFree(d_P);
cudaFree(d_M);
cudaFree(d_N);
```

Only One Thread Block Used

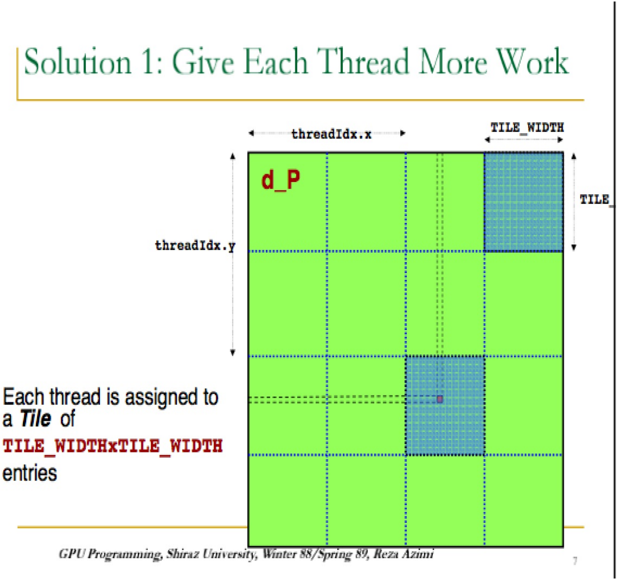
- One Block of threads compute matrix d_P
- Each thread
  - Loads a row of matrix d_M
  - Loads a column of matrix d_N
  - Perform one multiply and addition for each pair of d_M and d_N elements
  - Computes one element of d_P

Size of matrix limited by the number of threads allowed in a thread block



Adapted From:
David Kirk/NVIDIA and Wen-mei W. Hwu, UIUC *sity, Winter 88/Spring 89, Reza Azimi*

Handling Arbitrary Sized Square Matrices

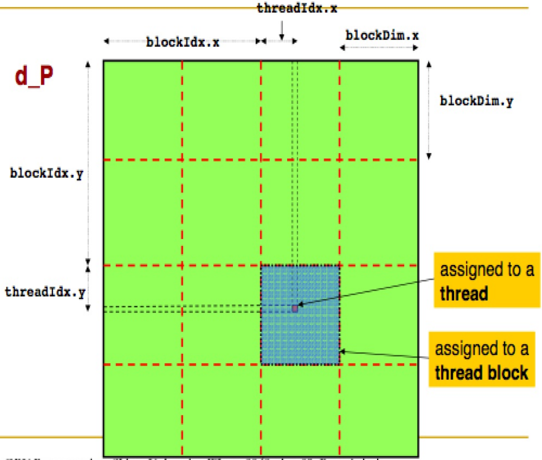


Solution 1: Give Each Thread More Work

```
__global__ void MatrixMulKernel(float* d_M,
                                float* d_N,
                                float* d_P,
                                int Width) {
  int start_row = threadIdx.y * TILE_WIDTH;
  int end_row = start_row + TILE_WIDTH;
  int start_col = threadIdx.x * TILE_WIDTH;
  int end_col = start_col + TILE_WIDTH;

  for (int row = start_row; row < end_row; row++) {
     for(int col = start_col; col < end_col; col++) {
        float P_val = 0;
        for (int k = 0; k < Width; ++k) {
           float M_elem = d_M[row * Width + k];
           float N_elem = d_N[k * Width + col];
           P_val += M_elem * N_elem;
        }
        d_p[row*Width+col] = P_val;
     }
  }
```

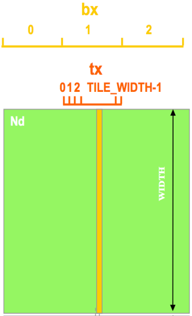
With one block we utilize only one multiprocessor!

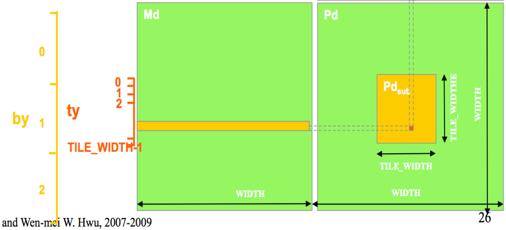
Solution 2: Use Multiple Thread Blocks



Matrix Multiplication Using Multiple Blocks

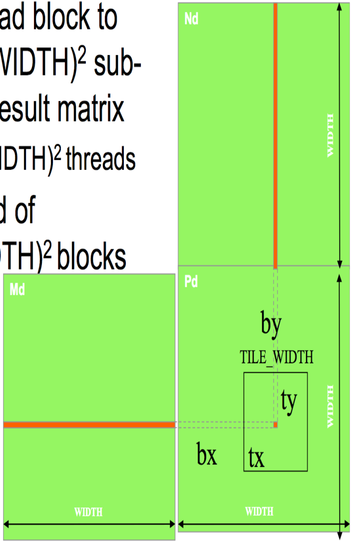
- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



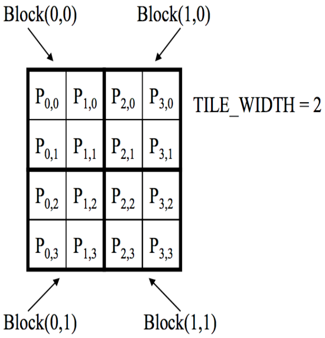


- Have each 2D thread block to compute a $(\text{TILE\_WIDTH})^2$ sub-matrix (tile) of the result matrix
  - Each has $(\text{TILE\_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{TILE\_WIDTH})^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!

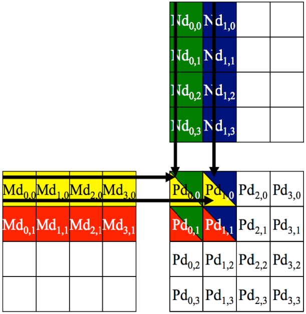


Matrix-Multiplication Using 2x2 Block Grid



Matrix-Multiplication - Thread Actions

A Small Example: Multiplication



Matrix-Multiplication using multiple thread blocks

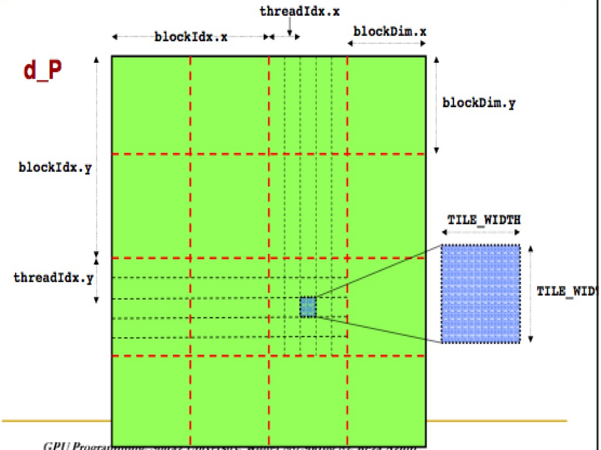
```
int block_size = 64;

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


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(d_M, d_N, d_P,
Width);

...
```

Size of matrix limited by the number of threads allowed on a device

Matrix-Multiplication using multiple thread blocks

```
__global__
void MatrixMulKernel(float* d_M,
                     float* d_N,
                     float* d_P,
                     int Width) {
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float P_val = 0;

  for (int k = 0; k < Width; ++k) {
    float M_elem = d_M[row * Width + k];
    float N_elem = d_N[k * Width + col];
    P_val += M_elem * N_elem;
  }
  d_p[row*Width+col] = P_val;
}
```

Combining the Two Solutions

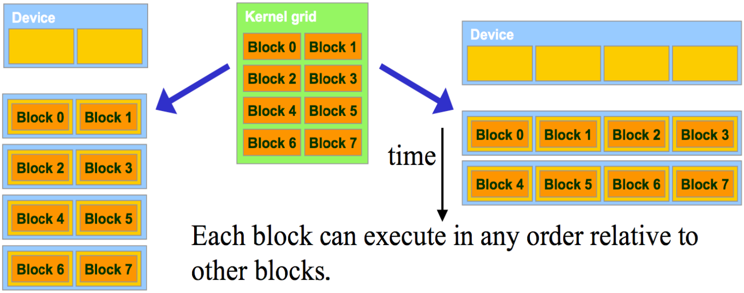


Combining the Two Solutions

```
__global__ void MatrixMulKernel(float* d_M,
                               float* d_N,
                               float* d_P,
                               int Width) {
  int start_row = blockDim.y * blockIdx.y + threadIdx.y * TILE_WIDTH;
  int end_row = start_row + TILE_WIDTH;
  int start_col = blockDim.x * blockIdx.x + threadIdx.x * TILE_WIDTH;
  int end_col = start_col + TILE_WIDTH;

  for (int row = start_row; row < end_row; row++) {
     for(int col = start_col; col < end_col; col++) {
        float P_val = 0;
        for (int k = 0; k < Width; ++k) {
           float M_elem = d_M[row * Width + k];
           float N_elem = d_N[k * Width + col];
           P_val += M_elem * N_elem;
        }
        d_p[row*Width+col] = P_val;
     }
  }
}
```

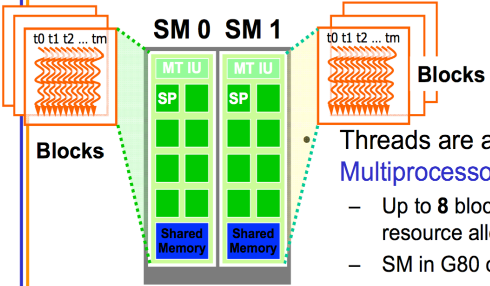
Transparent Scalability

- Hardware is free to assign blocks to any processor at any time, given the resources
  - A kernel scales across any number of parallel processors
  - When less resources are available, hardware will reduce the number of blocks run in parallel (compare right with left block assignment below)



Each block can execute in any order relative to other blocks.

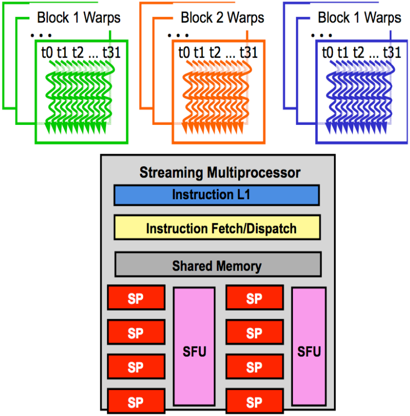
G80 Example: Executing Thread Blocks



- 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

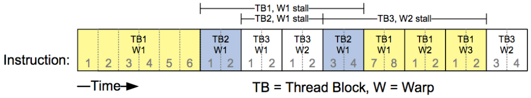
G80 Example: Thread Scheduling

- 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



G80 Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
  - Effectively provides for *latency hiding* (memory waits, etc.)
  - 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



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! This will lead to under-utilization (bad for latency hiding).
  - 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!

CONTENT