Basic Example: Matrix Multiplication using CUDA

General-purpose Programming of Massively Parallel Graphics Processors
Shiraz University, Spring 2010
Instructor: Reza Azimi

Some materials/slides are adapted from:
Andreas Moshovos’ Course at the University of Toronto
UIUC course by Wen-Mei Hwu and David Kirk

A Simple Host Version in C

```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
__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;
}

void MatrixMulOnDevice(float* M,
float* N,
float* P,
int Width)
{
    int matrix_size = Width * Width * sizeof(float);
    float *d_M, *d_N, *d_P;

    // Allocate and Load M and N to device memory
    cudaMalloc(&d_M, matrix_size);
    cudaMemcpy(d_M, M, matrix_size, cudaMemcpyHostToDevice);
    cudaMalloc(&d_N, matrix_size);
    cudaMemcpy(d_N, N, matrix_size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&d_P, matrix_size);
Kernel Invocation and Copy Results

```c
// 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 \( \text{d\_P} \)
- Each thread
  - Loads a row of matrix \( \text{d\_M} \)
  - Loads a column of matrix \( \text{d\_N} \)
  - Perform one multiply and addition for each pair of \( \text{d\_M} \) and \( \text{d\_N} \) elements
  - Computes one element of \( \text{d\_P} \)

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
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

```c
__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;
}
```
**Kernel Invocation and Copy Results**

```c
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

---

**GTX 280 Thread Limitations**

- Max Number of Threads per Block: 512
- Max Number of Blocks per Streaming Multiprocessor: 8
- Number of Streaming Multiprocessors: 30
- Total Number of Threads Available =
  
  \[30 \times 8 \times 512 = 122880\]

Let me double-check this!
Combining the Two Solutions

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