ECE408/CS483/CSE408 Spring 2018Applied Parallel Programming

Lectures 6:
More on Tiling

Objective

- To learn to assess the benefit of tiling
- To learn to handle boundary conditions in tiled algorithms

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    // The code assumes that the Width is a multiple of TILE WIDTH!
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
       // Collaborative loading of M and N tiles into shared memory
9.     subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.    subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
11.    __syncthreads();
12.    for (int k = 0; k < TILE_WIDTH; ++k)
13.        Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.    __syncthreads();
15. }
16. P[Row*Width+Col] = Pvalue;
}
```


Shared Memory and Threading

- Each SM in Maxwell has 64KB shared memory (48KB max per block)
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
    - Shared memory can potentially support up to 32 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 8
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block,
    - Shared memory can potentially support up to 8 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 2
    - This allows up to 2*2,048 = 4,096 pending loads (2 per thread, 1024 threads per block)

Memory Bandwidth Consumption

- Using 16x16 tiling, we reduce the global memory by a factor of 16
  - Each operand is now used by 16 floating-point operations
  - The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!
- Using 32x32 tiling, we reduce the global memory accesses by a factor of 32
  - Each operand is no used by 32 floating-point operations
  - The 150 GB/s bandwidth can now support (150/4)*32 = 1,200 GFLOPS!
  - The memory bandwidth is no longer a limiting factor for performance!

Device Query

- Number of devices in the system

```
int dev_count;
cudaGetDeviceCount( &dev_count);
```

- Capability of devices

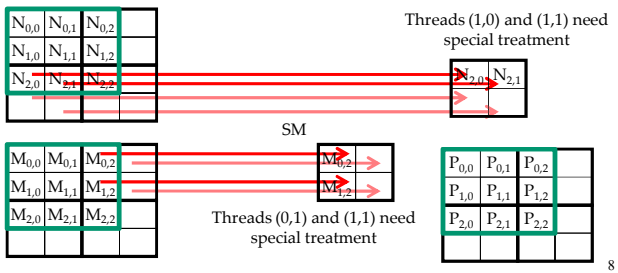
```
cudaDeviceProp          dev_prop;
for (i = 0; i < dev_count; i++) {
        cudaGetDeviceProperties( &dev_prop, i);

  // decide if device has sufficient resources and capabilities
}
```

- cudaDeviceProp is a built-in C structure type
  - dev_prop.dev_prop.maxThreadsPerBlock
  - Dev_prop.sharedMemoryPerBlock
  - …

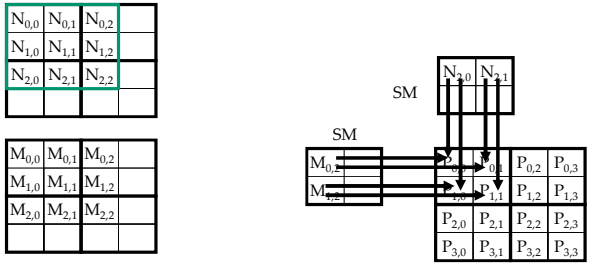
Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel in Lecture 5 can handle only the matrices whose dimensions are multiples of the tile dimensions
  - However, a real application needs to handle arbitrary sized matrices.
  - Could pad (add elements to) the rows and columns into multiples of block size, but will have significant space and data transfer time overhead.
- We will take a different approach.

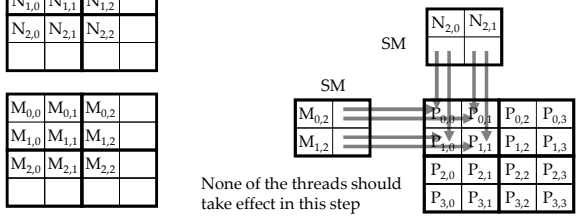
Review: Phase 1 Load for Block (0,0)



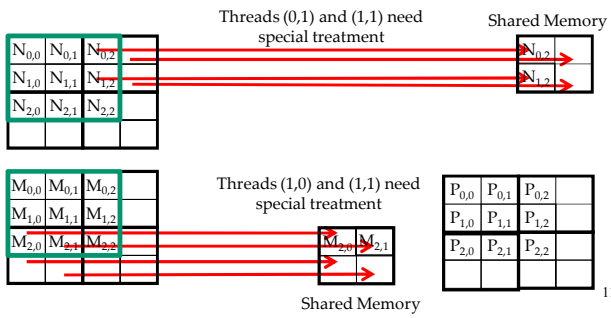
Review: Phase 1 Use for Block (0,0)



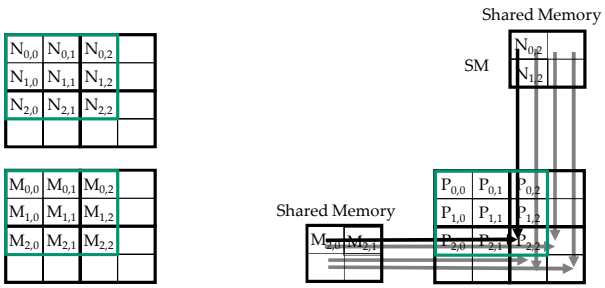
Review: Phase 1 Use for Block (0,0)



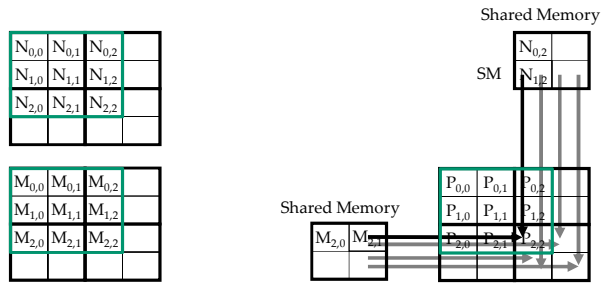
Phase 0 Load for Block (1,1)



Phase 0 Use for Block (1,1)



Phase 0 Use for Block (1,1)



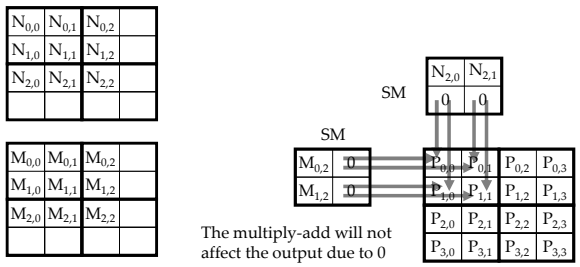
Major Cases in Toy Example

- Threads that calculate valid P elements but can step outside valid input
  - Phase 1 of Block(0,0), 2nd step, all threads
- Threads that do not calculate valid P elements
  - Block(1,1), Thread(1,0), non-existent row
  - Block(1,1), Thread(0,1), non-existing column
  - Block(1,1), Thread(1,1), non-existing row/column

A "Simple" Solution

- Test if a thread is to load any input element outside the valid range
  - If valid, proceed to load
  - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element

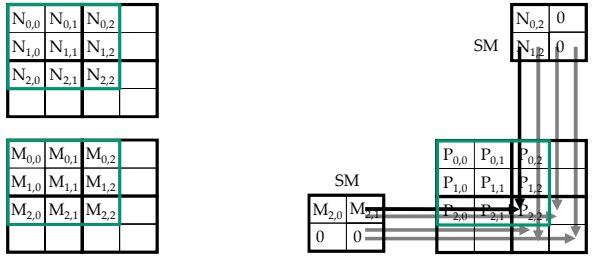
Phase 1 Use for Block (0,0)



A “Simple” Solution (Cont.)

- If a thread does not calculate a valid P element
  - Can still perform multiply-add into its register
  - As long as it is not allowed to write to the global memory at the end of the kernel

Phase 1 Use for Block (1,1)



Loading Elements – Before and After

```
8   for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m)
{
9     Mds[ty][tx] = M[Row*Width + m*TILE_WIDTH + tx];

10    Nds[ty][tx] = N[(m*TILE_WIDTH + ty)*Width + Col];

11    __syncthreads();
```

```
8    for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m) {
++       if(Row < Width && m*TILE_WIDTH+tx < Width) {
9        Mds[ty][tx] = M[Row*Width + m*TILE_WIDTH + tx];
++       } else {
++         Mds[ty][tx] = 0;
++       }
++       if(m*TILE_WIDTH+ty < Width && Col < Width ) {
10         Nds[ty][tx] = N[(m*TILE_WIDTH + ty)*Width + Col];
++       } else {
++         Nds[ty][tx] = 0;
++       }
11     __syncthreads();
```


Inner Product – Before and After

```
12    for (int k = 0; k < TILE_WIDTH; ++k) {
13          Pvalue += Mds[ty][k] * Nds[k][tx];
      }

14  __syncthreads();
15  }

16    P[Row*Width + Col] = Pvalue;
```

```
++    if(Row < Width && Col < Width) {
12      for (int k = 0; k < TILE_WIDTH; ++k) {
13          Pvalue += Mds[ty][k] * Nds[k][tx];
        }
++    }
14  __syncthreads();
15  }
++    if (Row < Width && Col < Width)
16      P[Row*Width + Col] = Pvalue;
```

Some Important Points

- For each thread the conditions are different for
  - Loading M element
  - Loading N element
  - Calculation/storing output elements
- The effect of control divergence should be small for large matrices
- How about rectangular matrices?

ANY MORE QUESTIONS?
READ CHAPTER 4!