The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.

// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {
        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            pixVal += in[curRow * w + curCol];
            pixels++;
        // Keep track of number of pixels in the accumulated total
        }
    }
}

// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
How about performance on a GPU

- All threads access global memory for their input matrix elements
  - One memory accesses (4 bytes) per floating-point addition
  - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
  - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
  - 4*1,500 = 6,000 GB/s required to achieve peak FLOPS rating
  - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS

- This limits the execution rate to 3.3% (50/1500) of the peak floating-point execution rate of the device!

- Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS
Programmer View of CUDA Memories

- Grid
- Block (0, 0)
  - Shared Memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)
- Block (1, 0)
  - Shared Memory
  - Registers
  - Thread (0, 0)
  - Thread (1, 0)
- Host
- Global Memory
- Constant Memory
## Declaring CUDA Variables

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __shared__, or __constant__
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory
Example:
Shared Memory Variable Declaration

```c
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{
    __shared__ float ds_in[TILE_WIDTH][TILE_WIDTH];

    ...
}
```
Where to Declare Variables?

- **Outside of any Function**
  - `global`
  - `constant`

- **In the kernel**
  - `register`
  - `shared`
Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture
Hardware View of CUDA Memories

Diagram showing the hardware components of a CUDA processor (SM):
- **Global Memory**
- **I/O**
- **Shared Memory**
- **Processing Unit**
  - **ALU**
  - **Register File**
- **Control Unit**
  - **PC**
  - **IR**
- **Processor (SM)**
Example – Matrix Multiplication
A Basic Matrix Multiplication

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;

    // Calculate the column index of P and 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 += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```
Example – Matrix Multiplication

__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {

    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;

    // Calculate the column index of P and 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 += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}

Calculation of $P_{0,0}$ and $P_{0,1}$
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory

Thread 1

Thread 2
Tiling/Blocking - Basic Idea

Global Memory

On-chip Memory

Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time
Tiling/Blocking - Basic Idea

Global Memory

On-chip Memory

Thread 1

Thread 2
Barrier Synchronization for Tiling

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

Thread N-3
Thread N-2
Thread N-1

Time
Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile
Matrix Multiplication

- Data access pattern
  - Each thread - a row of M and a column of N
  - Each thread block – a strip of M and a strip of N
Tiled Matrix Multiplication

- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension
Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in tiled code
Phase 0 Load for Block (0,0)

Shared Memory

Shared Memory
### Phase 0 Use for Block (0,0) (iteration 0)

<table>
<thead>
<tr>
<th>N_{0,0}</th>
<th>N_{0,1}</th>
<th>N_{0,2}</th>
<th>N_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{1,0}</td>
<td>N_{1,1}</td>
<td>N_{1,2}</td>
<td>N_{1,3}</td>
</tr>
<tr>
<td>N_{2,0}</td>
<td>N_{2,1}</td>
<td>N_{2,2}</td>
<td>N_{2,3}</td>
</tr>
<tr>
<td>N_{3,0}</td>
<td>N_{3,1}</td>
<td>N_{3,2}</td>
<td>N_{3,3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M_{0,0}</th>
<th>M_{0,1}</th>
<th>M_{0,2}</th>
<th>M_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{1,0}</td>
<td>M_{1,1}</td>
<td>M_{1,2}</td>
<td>M_{1,3}</td>
</tr>
<tr>
<td>M_{2,0}</td>
<td>M_{2,1}</td>
<td>M_{2,2}</td>
<td>M_{2,3}</td>
</tr>
<tr>
<td>M_{3,0}</td>
<td>M_{3,1}</td>
<td>M_{3,2}</td>
<td>M_{3,3}</td>
</tr>
</tbody>
</table>

- **Shared Memory**

<table>
<thead>
<tr>
<th>N_{0,0}</th>
<th>N_{0,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{1,0}</td>
<td>N_{1,1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M_{0,0}</th>
<th>M_{0,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{1,0}</td>
<td>M_{1,1}</td>
</tr>
</tbody>
</table>

- **Shared Memory**

<table>
<thead>
<tr>
<th>P_{0,0}</th>
<th>P_{0,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{1,0}</td>
<td>P_{1,1}</td>
</tr>
<tr>
<td>P_{2,0}</td>
<td>P_{2,1}</td>
</tr>
<tr>
<td>P_{3,0}</td>
<td>P_{3,1}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_{0,2}</th>
<th>P_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_{1,2}</td>
<td>P_{1,3}</td>
</tr>
<tr>
<td>P_{2,2}</td>
<td>P_{2,3}</td>
</tr>
<tr>
<td>P_{3,2}</td>
<td>P_{3,3}</td>
</tr>
</tbody>
</table>
Phase 0 Use for Block (0,0) (iteration 1)

<table>
<thead>
<tr>
<th></th>
<th>N_{0,0}</th>
<th>N_{0,1}</th>
<th>N_{0,2}</th>
<th>N_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{1,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{1,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{1,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{2,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{2,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{2,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{3,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{3,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M_{0,0}</th>
<th>M_{0,1}</th>
<th>M_{0,2}</th>
<th>M_{0,3}</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{1,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{1,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{1,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{1,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{2,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{2,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{2,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{2,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{3,0}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{3,1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{3,2}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M_{3,3}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Shared Memory**

- **Shared Memory Access**
  - N_{0,0} to P_{0,0}
  - N_{1,0} to P_{0,1}
  - N_{2,0} to P_{0,2}
  - N_{3,0} to P_{0,3}
  - N_{0,1} to P_{1,0}
  - N_{1,1} to P_{1,1}
  - N_{2,1} to P_{1,2}
  - N_{3,1} to P_{1,3}
  - N_{0,2} to P_{2,0}
  - N_{1,2} to P_{2,1}
  - N_{2,2} to P_{2,2}
  - N_{3,2} to P_{2,3}
  - N_{0,3} to P_{3,0}
  - N_{1,3} to P_{3,1}
  - N_{2,3} to P_{3,2}
  - N_{3,3} to P_{3,3}
Phase 1 Load for Block (0,0)
Phase 1 Use for Block (0,0) (iteration 0)
Phase 1 Use for Block (0,0) (iteration 1)

<table>
<thead>
<tr>
<th></th>
<th>N₀₀</th>
<th>N₀₁</th>
<th>N₀₂</th>
<th>N₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₁₀</td>
<td>N₁₁</td>
<td>N₁₂</td>
<td>N₁₃</td>
<td></td>
</tr>
<tr>
<td>N₂₀</td>
<td>N₂₁</td>
<td>N₂₂</td>
<td>N₂₃</td>
<td></td>
</tr>
<tr>
<td>N₃₀</td>
<td>N₃₁</td>
<td>N₃₂</td>
<td>N₃₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>M₀₀</th>
<th>M₀₁</th>
<th>M₀₂</th>
<th>M₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁₀</td>
<td>M₁₁</td>
<td>M₁₂</td>
<td>M₁₃</td>
<td></td>
</tr>
<tr>
<td>M₂₀</td>
<td>M₂₁</td>
<td>M₂₂</td>
<td>M₂₃</td>
<td></td>
</tr>
<tr>
<td>M₃₀</td>
<td>M₃₁</td>
<td>M₃₂</td>
<td>M₃₃</td>
<td></td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>N₂₀</th>
<th>N₂₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₃₀</td>
<td>N₃₁</td>
<td></td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>P₀₀</th>
<th>P₀₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀₂</td>
<td>P₀₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₁₀</th>
<th>P₁₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁₂</td>
<td>P₁₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₂₀</th>
<th>P₂₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₂₂</td>
<td>P₂₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₃₀</th>
<th>P₃₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₃₂</td>
<td>P₃₃</td>
<td></td>
</tr>
</tbody>
</table>
Execution Phases of Toy Example

<table>
<thead>
<tr>
<th>thread&lt;sub&gt;0,0&lt;/sub&gt;</th>
<th>Phase 0</th>
<th>Phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>M&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>N&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>PValue&lt;sub&gt;0,0&lt;/sub&gt; += M&lt;sub&gt;0,0&lt;/sub&gt;*N&lt;sub&gt;0,0&lt;/sub&gt; + M&lt;sub&gt;0,1&lt;/sub&gt;*N&lt;sub&gt;1,0&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;ds&lt;sub&gt;0,0&lt;/sub&gt;&lt;/sub&gt;</td>
<td>N&lt;sub&gt;ds&lt;sub&gt;0,0&lt;/sub&gt;&lt;/sub&gt;</td>
<td>M&lt;sub&gt;ds&lt;sub&gt;0,0&lt;/sub&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>thread&lt;sub&gt;0,1&lt;/sub&gt;</td>
<td>M&lt;sub&gt;0,1&lt;/sub&gt;</td>
<td>N&lt;sub&gt;0,1&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;ds&lt;sub&gt;0,1&lt;/sub&gt;&lt;/sub&gt;</td>
<td>N&lt;sub&gt;ds&lt;sub&gt;0,1&lt;/sub&gt;&lt;/sub&gt;</td>
<td>M&lt;sub&gt;ds&lt;sub&gt;0,1&lt;/sub&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>thread&lt;sub&gt;1,0&lt;/sub&gt;</td>
<td>M&lt;sub&gt;1,0&lt;/sub&gt;</td>
<td>N&lt;sub&gt;1,0&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;ds&lt;sub&gt;1,0&lt;/sub&gt;&lt;/sub&gt;</td>
<td>N&lt;sub&gt;ds&lt;sub&gt;1,0&lt;/sub&gt;&lt;/sub&gt;</td>
<td>M&lt;sub&gt;ds&lt;sub&gt;1,0&lt;/sub&gt;&lt;/sub&gt;</td>
</tr>
<tr>
<td>thread&lt;sub&gt;1,1&lt;/sub&gt;</td>
<td>M&lt;sub&gt;1,1&lt;/sub&gt;</td>
<td>N&lt;sub&gt;1,1&lt;/sub&gt;</td>
</tr>
<tr>
<td>M&lt;sub&gt;ds&lt;sub&gt;1,1&lt;/sub&gt;&lt;/sub&gt;</td>
<td>N&lt;sub&gt;ds&lt;sub&gt;1,1&lt;/sub&gt;&lt;/sub&gt;</td>
<td>M&lt;sub&gt;ds&lt;sub&gt;1,1&lt;/sub&gt;&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

PValue<sub>0,0</sub> += M<sub>ds<sub>0,0</sub></sub>*N<sub>ds<sub>0,0</sub></sub> + M<sub>ds<sub>0,1</sub></sub>*N<sub>ds<sub>1,0</sub></sub>
PValue<sub>0,1</sub> += M<sub>ds<sub>0,0</sub></sub>*N<sub>ds<sub>0,1</sub></sub> + M<sub>ds<sub>0,1</sub></sub>*N<sub>ds<sub>1,1</sub></sub>
PValue<sub>1,0</sub> += M<sub>ds<sub>1,0</sub></sub>*N<sub>ds<sub>0,0</sub></sub> + M<sub>ds<sub>1,1</sub></sub>*N<sub>ds<sub>0,0</sub></sub>
PValue<sub>1,1</sub> += M<sub>ds<sub>1,0</sub></sub>*N<sub>ds<sub>0,1</sub></sub> + M<sub>ds<sub>1,1</sub></sub>*N<sub>ds<sub>1,1</sub></sub>
Shared memory allows each value to be accessed by multiple threads.
Barrier Synchronization

- Synchronize all threads in a block
  - __syncthreads()

- All threads in the same block must reach the __syncthreads() before any of the them can move on

- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase
Tiled Matrix Multiplication Kernel

__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{

__shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
__shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;
float Pvalue = 0;

// Loop over the M and N tiles required to compute the P element
for (int p = 0; p < Width/TILE_WIDTH; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
}
P[Row*Width+Col] = Pvalue;
}
Tiled Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < Width/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```
Tiled Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < Width/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```
Tile (Thread Block) Size Considerations

- Each **thread block** should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads

- For 16, in each phase, each block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations. (16 floating-point operations for each memory load)

- For 32, in each phase, each block performs 2*1024 = 2048 float loads from global memory for 1024 * (2*32) = 65,536 mul/add operations. (32 floating-point operation for each memory load)
Shared Memory and Threading

- For an SM with 16KB shared memory
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses $2*256*4B = 2KB$ of shared memory.
  - For 16KB shared memory, one can potentially have up to 8 thread blocks executing
    - 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 $2*32*32*4$ Byte = 8K Byte shared memory usage per thread block, allowing 2 thread blocks active at the same time
    - However, the thread count limitation of 1536 threads per SM in current generation GPUs will reduce the number of blocks per SM to one!
- Each __syncthread() can reduce the number of active threads for a block
  - More thread blocks can be advantageous
Performance Analysis

Matrices are 2048 x 2048

Tiles are 16 x 16

Running on a GTX 980

Max theoretical throughput is 4,616 GFLOPS

This kernel achieves 500 - 600 GFLOPS
Module 4.5 - Memory and Data Locality
Handling Arbitrary Matrix Sizes in Tiled Algorithms
Objective

- To learn to handle arbitrary matrix sizes in tiled matrix multiplication
  - Boundary condition checking
  - Regularizing tile contents
  - Rectangular matrices
Handling Matrix of Arbitrary Size

• The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
  • However, real applications need to handle arbitrary sized matrices.
  • One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
• We will take a different approach.
Phase 1 Loads for Block (0,0) for a 3x3 Example

Threads (1,0) and (1,1) need special treatment in loading N tile

Threads (0,1) and (1,1) need special treatment in loading M tile
Phase 1 Use for Block (0,0) (iteration 0)

Shared Memory

N_0,0  N_0,1  N_0,2
N_1,0  N_1,1  N_1,2
N_2,0  N_2,1  N_2,2

M_0,0  M_0,1  M_0,2
M_1,0  M_1,1  M_1,2
M_2,0  M_2,1  M_2,2

P_0,0  P_0,1  P_0,2
P_1,0  P_1,1  P_1,2
P_2,0  P_2,1  P_2,2
Phase 1 Use for Block (0,0) (iteration 1)

All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.
Phase 0 Loads for Block (1,1) for a 3x3 Example

Threads (0,1) and (1,1) need special treatment in loading N tile

Threads (1,0) and (1,1) need special treatment in loading M tile
Major Cases in Toy Example

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
  - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]

- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
  - Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]
A “Simple” Solution

- When a thread is to load any input element, test if it is in the valid index 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

- The condition tested for loading input elements is different from the test for calculating output P element
  - A thread that does not calculate valid P element can still participate in loading input tile elements
Phase 1 Use for Block (0,0) (iteration 1)

Shared Memory

N₀,₀  N₀,₁  N₀,₂
N₁,₀  N₁,₁  N₁,₂
N₂,₀  N₂,₁  N₂,₂

M₀,₀  M₀,₁  M₀,₂
M₁,₀  M₁,₁  M₁,₂
M₂,₀  M₂,₁  M₂,₂

N₂,₀  N₂,₁
N₂,₀  N₂,₁
N₂,₀  N₂,₁

M₀,₂  0
M₁,₂  0
M₂,₀  0

P₀,₀  P₀,₁  P₀,₂
P₁,₀  P₁,₁  P₁,₂
P₂,₀  P₂,₁  P₂,₂
Boundary Condition for Input M Tile

- Each thread loads
  - $M[\text{Row}][p*\text{TILE_WIDTH}+tx]$
  - $M[\text{Row}*\text{Width} + p*\text{TILE_WIDTH}+tx]$

- Need to test
  - $(\text{Row} < \text{Width}) \&\& (p*\text{TILE_WIDTH}+tx < \text{Width})$
  - If true, load M element
  - Else, load 0
Boundary Condition for Input N Tile

- Each thread loads
  - \(N[p*TILE\_WIDTH+ty][Col]\)
  - \(N[(p*TILE\_WIDTH+ty)*Width+ Col]\)

- Need to test
  - \((p*TILE\_WIDTH+ty < Width) && (Col< Width)\)
  - If true, load N element
  - Else, load 0
Loading Elements – with boundary check

for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {
    if(Row < Width && t * TILE_WIDTH+tx < Width) {
        ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    } else {
        ds_M[ty][tx] = 0.0;
    }
    if (p*TILE_WIDTH+ty < Width && Col < Width) {
        ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
    } else {
        ds_N[ty][tx] = 0.0;
    }
    __syncthreads();
}
Inner Product – Before and After

```c
++ if(Row < Width && Col < Width) {
12   for (int i = 0; i < TILE_WIDTH; ++i) {
13       Pvalue += ds_M[ty][i] * ds_N[i][tx];
14   }
15   __syncthreads();
16 } /* end of outer for loop */
++ if (Row < Width && Col < Width)
16   P[Row*Width + Col] = Pvalue;
} /* end of kernel */
```
Some Important Points

– For each thread the conditions are different for
  – Loading M element
  – Loading N element
  – Calculating and storing output elements
– The effect of control divergence should be small for large matrices
Handling General Rectangular Matrices

- In general, the matrix multiplication is defined in terms of rectangular matrices
  - A $j \times k$ M matrix multiplied with a $k \times l$ N matrix results in a $j \times l$ P matrix

- We have presented square matrix multiplication, a special case

- The kernel function needs to be generalized to handle general rectangular matrices
  - The Width argument is replaced by three arguments: $j$, $k$, $l$
  - When Width is used to refer to the height of M or height of P, replace it with $j$
  - When Width is used to refer to the width of M or height of N, replace it with $k$
  - When Width is used to refer to the width of N or width of P, replace it with $l$