Memory and Data Locality

CUDA Memories
Tiled Parallel Algorithms
Tiled Matrix Multiplication – kernel implementation
Handling Arbitrary Matrix Sizes in Tiled Algorithms
Objective

To learn to effectively use the CUDA memory types in a parallel program

- Importance of memory access efficiency
- Registers, shared memory, global memory
- Scope and lifetime

```c
// 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
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;
    }
}
```
__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;
    }
}
A Toy Example: Thread to P Data Mapping

\[
\begin{array}{cccc}
P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\
P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\
P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\
P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \\
\end{array}
\]

\text{BLOCK_WIDTH} = 2
Calculation of $P_{0,0}$ and $P_{0,1}$

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

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

<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>
</tbody>
</table>
Memory and Registers in the Von-Neumann Model
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
Objective

- To understand the motivation and ideas for tiled parallel algorithms
  - Reducing the limiting effect of memory bandwidth on parallel kernel performance
  - Tiled algorithms and barrier synchronization
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory

Diagram showing the access pattern between Global Memory and Threads 1 and 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
Basic Concept of Tiling

- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
  - Carpooling for commuters
  - Tiling for global memory accesses
    - drivers = threads accessing their memory data operands
    - cars = memory access requests
Some Computations are More Challenging to Tile

- Some carpools may be easier than others
  - Car pool participants need to have similar work schedule
  - Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling
Carpools need synchronization.

– Good: when people have similar schedule
Carpools need synchronization.

– Bad: when people have very different schedule
Same with Tiling

- **Good:** when threads have similar access timing

- **Bad:** when threads have very different timing
Barrier Synchronization for Tiling
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
Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
  - Loading a tile
  - Phased execution
  - Barrier Synchronization
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)
Phase 0 Use for Block (0,0) (iteration 0)

<table>
<thead>
<tr>
<th>N0,0</th>
<th>N0,1</th>
<th>N0,2</th>
<th>N0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1,0</td>
<td>N1,1</td>
<td>N1,2</td>
<td>N1,3</td>
</tr>
<tr>
<td>N2,0</td>
<td>N2,1</td>
<td>N2,2</td>
<td>N2,3</td>
</tr>
<tr>
<td>N3,0</td>
<td>N3,1</td>
<td>N3,2</td>
<td>N3,3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M0,0</th>
<th>M0,1</th>
<th>M0,2</th>
<th>M0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1,0</td>
<td>M1,1</td>
<td>M1,2</td>
<td>M1,3</td>
</tr>
<tr>
<td>M2,0</td>
<td>M2,1</td>
<td>M2,2</td>
<td>M2,3</td>
</tr>
<tr>
<td>M3,0</td>
<td>M3,1</td>
<td>M3,2</td>
<td>M3,3</td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th>P0,0</th>
<th>P0,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1,0</td>
<td>P1,1</td>
</tr>
<tr>
<td>P2,0</td>
<td>P2,1</td>
</tr>
<tr>
<td>P3,0</td>
<td>P3,1</td>
</tr>
<tr>
<td>P0,2</td>
<td>P0,3</td>
</tr>
<tr>
<td>P1,2</td>
<td>P1,3</td>
</tr>
<tr>
<td>P2,2</td>
<td>P2,3</td>
</tr>
<tr>
<td>P3,2</td>
<td>P3,3</td>
</tr>
</tbody>
</table>

Shared Memory
Phase 0 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>N₀₀</th>
<th>N₀₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₁₀</td>
<td>N₁₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₀₀</th>
<th>P₀₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁₀</td>
<td>P₁₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₀₂</th>
<th>P₀₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁₂</td>
<td>P₁₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₀₂</th>
<th>P₀₃</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>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>

<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>
Phase 1 Load for Block (0,0)
Phase 1 Use for Block (0,0) (iteration 0)

<table>
<thead>
<tr>
<th></th>
<th>N₀,₀</th>
<th>N₀,₁</th>
<th>N₀,₂</th>
<th>N₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₁,₀</td>
<td>N₁,₁</td>
<td>N₁,₂</td>
<td>N₁,₃</td>
<td></td>
</tr>
<tr>
<td>N₂,₀</td>
<td>N₂,₁</td>
<td>N₂,₂</td>
<td>N₂,₃</td>
<td></td>
</tr>
<tr>
<td>N₃,₀</td>
<td>N₃,₁</td>
<td>N₃,₂</td>
<td>N₃,₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M₀,₀</th>
<th>M₀,₁</th>
<th>M₀,₂</th>
<th>M₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁,₀</td>
<td>M₁,₁</td>
<td>M₁,₂</td>
<td>M₁,₃</td>
</tr>
<tr>
<td>M₂,₀</td>
<td>M₂,₁</td>
<td>M₂,₂</td>
<td>M₂,₃</td>
</tr>
<tr>
<td>M₃,₀</td>
<td>M₃,₁</td>
<td>M₃,₂</td>
<td>M₃,₃</td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>N₂,₀</th>
<th>N₂,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₃,₀</td>
<td>N₃,₁</td>
<td></td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>P₀,₀</th>
<th>P₀,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁,₀</td>
<td>P₁,₁</td>
<td></td>
</tr>
<tr>
<td>P₂,₀</td>
<td>P₂,₁</td>
<td></td>
</tr>
<tr>
<td>P₃,₀</td>
<td>P₃,₁</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P₀,₂</th>
<th>P₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁,₂</td>
<td>P₁,₃</td>
<td></td>
</tr>
<tr>
<td>P₂,₂</td>
<td>P₂,₃</td>
<td></td>
</tr>
<tr>
<td>P₃,₂</td>
<td>P₃,₃</td>
<td></td>
</tr>
</tbody>
</table>
Phase 1 Use for Block (0,0) (iteration 1)
## Execution Phases of Toy Example

<table>
<thead>
<tr>
<th>Thread</th>
<th>Phase 0</th>
<th>Phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{thread}_{0,0}$</td>
<td>$\begin{array}{c} M_{0,0} \ \downarrow \ Mds_{0,0} \end{array}$</td>
<td>$\begin{array}{c} P\text{Value}<em>{0,0} += \ Mds</em>{0,0} \times Nds_{0,0} + \ Mds_{0,1} \times Nds_{1,0} \ M_{0,2} \ \downarrow \ Mds_{0,0} \end{array}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{array}{c} N_{0,0} \ \downarrow \ Nds_{0,0} \end{array}$</td>
<td>$\begin{array}{c} N_{2,0} \ \downarrow \ Nds_{0,0} \end{array}$</td>
</tr>
<tr>
<td>$\text{thread}_{0,1}$</td>
<td>$\begin{array}{c} M_{0,1} \ \downarrow \ Mds_{0,1} \end{array}$</td>
<td>$\begin{array}{c} P\text{Value}<em>{0,1} += \ Mds</em>{0,0} \times Nds_{0,1} + \ Mds_{0,1} \times Nds_{1,1} \ M_{0,3} \ \downarrow \ Mds_{0,1} \end{array}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{array}{c} N_{0,1} \ \downarrow \ Nds_{1,0} \end{array}$</td>
<td>$\begin{array}{c} N_{2,1} \ \downarrow \ Nds_{0,1} \end{array}$</td>
</tr>
<tr>
<td>$\text{thread}_{1,0}$</td>
<td>$\begin{array}{c} M_{1,0} \ \downarrow \ Mds_{1,0} \end{array}$</td>
<td>$\begin{array}{c} P\text{Value}<em>{1,0} += \ Mds</em>{1,0} \times Nds_{0,0} + \ Mds_{1,1} \times Nds_{1,0} \ M_{1,2} \ \downarrow \ Mds_{1,0} \end{array}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{array}{c} N_{1,0} \ \downarrow \ Nds_{1,0} \end{array}$</td>
<td>$\begin{array}{c} N_{3,0} \ \downarrow \ Nds_{1,0} \end{array}$</td>
</tr>
<tr>
<td>$\text{thread}_{1,1}$</td>
<td>$\begin{array}{c} M_{1,1} \ \downarrow \ Mds_{1,1} \end{array}$</td>
<td>$\begin{array}{c} P\text{Value}<em>{1,1} += \ Mds</em>{1,0} \times Nds_{0,1} + \ Mds_{1,1} \times Nds_{1,1} \ M_{1,3} \ \downarrow \ Mds_{1,1} \end{array}$</td>
</tr>
<tr>
<td></td>
<td>$\begin{array}{c} N_{1,1} \ \downarrow \ Nds_{1,1} \end{array}$</td>
<td>$\begin{array}{c} N_{3,1} \ \downarrow \ Nds_{1,1} \end{array}$</td>
</tr>
</tbody>
</table>

**Note:** The values $P\text{Value}_i$ represent some form of computation or state change, and the $M_{ij}$ and $N_{ij}$ symbols represent operations or indexing, likely related to some form of data or memory management in a parallel computing context. The $Mds_{ij}$ and $Nds_{ij}$ likely represent data or state variables accessed by different threads at different times.
Execution Phases of Toy Example (cont.)

<table>
<thead>
<tr>
<th>Phase 0</th>
<th>Phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread&lt;sub&gt;0,0&lt;/sub&gt;</td>
<td>thread&lt;sub&gt;0,1&lt;/sub&gt;</td>
</tr>
<tr>
<td>( M_{0,0} )</td>
<td>( M_{0,1} )</td>
</tr>
<tr>
<td>( N_{0,0} )</td>
<td>( N_{0,1} )</td>
</tr>
<tr>
<td>( Mds_{0,0} )</td>
<td>( Mds_{0,1} )</td>
</tr>
<tr>
<td>( Nds_{0,0} )</td>
<td>( Nds_{1,0} )</td>
</tr>
</tbody>
</table>

\[
\text{PValue}_{0,0} += Mds_{0,0} \times Nds_{0,0} + Mds_{0,1} \times Nds_{1,0}
\]

\[
\text{PValue}_{0,1} += Mds_{0,0} \times Nds_{0,1} + Mds_{0,1} \times Nds_{1,1}
\]

\[
\text{PValue}_{1,0} += Mds_{1,0} \times Nds_{0,0} + Mds_{1,1} \times Nds_{1,0}
\]

\[
\text{PValue}_{1,1} += Mds_{1,0} \times Nds_{0,1} + Mds_{1,1} \times Nds_{1,1}
\]

\[
\text{PValue}_{0,2} += Mds_{0,0} \times Nds_{0,0} + Mds_{0,1} \times Nds_{1,0}
\]

\[
\text{PValue}_{0,3} += Mds_{0,0} \times Nds_{0,1} + Mds_{0,1} \times Nds_{1,1}
\]

\[
\text{PValue}_{1,2} += Mds_{1,0} \times Nds_{0,0} + Mds_{1,1} \times Nds_{1,0}
\]

\[
\text{PValue}_{1,3} += Mds_{1,0} \times Nds_{0,1} + Mds_{1,1} \times Nds_{1,1}
\]

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
Objective

- To learn to write a tiled matrix-multiplication kernel
  - Loading and using tiles for matrix multiplication
  - Barrier synchronization, shared memory
  - Resource Considerations
  - Assume that Width is a multiple of tile size for simplicity
Loading Input Tile 0 of M (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

```c
int Row = by * blockDim.y + ty;
int Col = bx * blockDim.x + tx;
```

2D indexing for accessing Tile 0:

- M[Row][tx]
- N[ty][Col]
Loading Input Tile 0 of N (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

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

2D indexing for accessing Tile 0:

\[
\begin{align*}
M[Row][tx] \\
N[ty][Col]
\end{align*}
\]
Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

\[
\begin{align*}
&M[\text{Row}][1*\text{TILE}_-\text{WIDTH} + \text{tx}] \\
&N[1*\text{TILE}_-\text{WIDTH} + \text{ty}][\text{Col}]
\end{align*}
\]
Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

\[ M[\text{Row}][1*\text{TILE\_WIDTH} + \text{tx}] \]
\[ N[1*\text{TILE\_WIDTH} + \text{ty}][\text{Col}] \]
M and N are dynamically allocated - use 1D indexing

\[
M[\text{Row}][p\times\text{TILE\_WIDTH}+tx] \\
M[\text{Row}\times\text{Width} + p\times\text{TILE\_WIDTH} + tx]
\]

\[
N[p\times\text{TILE\_WIDTH}+ty][\text{Col}] \\
N[(p\times\text{TILE\_WIDTH}+ty)\times\text{Width} + \text{Col}]
\]

where \( p \) is the sequence number of the current phase
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 < n/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[(t*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 < n/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[(t*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 < n/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[(t*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 `__syncthreads()` can reduce the number of active threads for a block
  - More thread blocks can be advantageous
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)
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)
Boundary Condition for Input M Tile

Each thread loads
- \( M[\text{Row}][p*\text{TILE}_\text{WIDTH}+tx] \)
- \( M[\text{Row}*\text{Width} + p*\text{TILE}_\text{WIDTH}+tx] \)

Need to test
- \((\text{Row} < \text{Width}) \&\& (p*\text{TILE}_\text{WIDTH}+tx < \text{Width})\)
- 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

8    for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {

++    if (Row < Width && t * TILE_WIDTH + tx < Width) {
9        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) {
10        ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
++    } else {
++        ds_N[ty][tx] = 0.0;
++    }
11    __syncthreads();

Inner Product – Before and After

```
++ if(Row < Width && Col < Width) {
  for (int i = 0; i < TILE_WIDTH; ++i) {
    Pvalue += ds_M[ty][i] * ds_N[i][tx];
  }
__syncthreads();
} /* end of outer for loop */
++ if (Row < Width && Col < Width)
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 $l$
The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).