CUDA MEMORIES
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
Example – Matrix Multiplication

Row

Col
__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

 BLOCK_WIDTH = 2

Thread(0,0)  Thread(1,0)  Thread(1,1)

Block(0,0)  Block(0,1)  Block(1,0)  Block(1,1)
Calculation of $P_{0,0}$ and $P_{0,1}$

<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>$P_{0,0}$</td>
<td>$P_{0,1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>

| $P_{1,0}$ | $P_{1,1}$ |
Memory and Registers in the Von-Neumann Model
Programmer View of CUDA Memories
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?

Can host access it?

- global
- constant
- Outside of any Function

- register
- shared
- In the kernel
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

- Global Memory
- I/O
- Shared Memory
- Processing Unit
  - ALU
  - Register File
- Control Unit
  - PC
  - IR
- Processor (SM)
TILED PARALLEL ALGORITHMS
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory
Tiling/Blocking - Basic Idea

Global 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
Tiling needs synchronization

– Good: when threads have similar access timing

– Bad: when threads have very different timing
Barrier Synchronization for Tiling

Thread 0
Thread 1
Thread 2
Thread 3
Thread 4

Time

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

...
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
TILED MATRIX MULTIPLICATION
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)
Phase 0 Use for Block (0,0) (iteration 1)
Phase 1 Load for Block (0,0)

<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>

Shared Memory

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

Shared Memory

<table>
<thead>
<tr>
<th></th>
<th>P0,0</th>
<th>P0,1</th>
<th>P0,2</th>
<th>P0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1,0</td>
<td>P1,1</td>
<td>P1,2</td>
<td>P1,3</td>
<td></td>
</tr>
<tr>
<td>P2,0</td>
<td>P2,1</td>
<td>P2,2</td>
<td>P2,3</td>
<td></td>
</tr>
<tr>
<td>P3,0</td>
<td>P3,1</td>
<td>P3,2</td>
<td>P3,3</td>
<td></td>
</tr>
</tbody>
</table>
Phase 1 Use for Block (0,0) (iteration 0)
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>thread_0,0</td>
<td>(M_{0,0}) (\downarrow) (Mds_{0,0}) (\downarrow) (N_{0,0}) (\downarrow) (Nds_{0,0})</td>
<td>(PValue_{0,0} += Mds_{0,0} * Nds_{0,0} + Mds_{0,1} * Nds_{1,0})</td>
</tr>
<tr>
<td>thread_0,1</td>
<td>(M_{0,1}) (\downarrow) (Mds_{0,1}) (\downarrow) (N_{0,1}) (\downarrow) (Nds_{1,0})</td>
<td>(PValue_{0,1} += Mds_{0,0} * Nds_{0,1} + Mds_{0,1} * Nds_{1,1})</td>
</tr>
<tr>
<td>thread_1,0</td>
<td>(M_{1,0}) (\downarrow) (Mds_{1,0}) (\downarrow) (N_{1,0}) (\downarrow) (Nds_{1,0})</td>
<td>(PValue_{1,0} += Mds_{1,0} * Nds_{0,0} + Mds_{1,1} * Nds_{1,0})</td>
</tr>
<tr>
<td>thread_1,1</td>
<td>(M_{1,1}) (\downarrow) (Mds_{1,1}) (\downarrow) (N_{1,1}) (\downarrow) (Nds_{1,1})</td>
<td>(PValue_{1,1} += Mds_{1,0} * Nds_{0,1} + Mds_{1,1} * Nds_{1,1})</td>
</tr>
</tbody>
</table>
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
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.

```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 1 of M (Phase 1)

2D indexing for accessing Tile 1:
- \( M[\text{Row}][1*\text{TILE_WIDTH} + \text{tx}] \)
- \( N[1*\text{TILE_WIDTH} + \text{ty}][\text{Col}] \)
2D indexing for accessing Tile 1:

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

\[
\begin{align*}
M[\text{Row}][p\times\text{TILE\_WIDTH} + \text{tx}] \\
M[\text{Row}\times\text{Width} + p\times\text{TILE\_WIDTH} + \text{tx}] \\
N[p\times\text{TILE\_WIDTH} + \text{ty}][\text{Col}] \\
N[(p\times\text{TILE\_WIDTH} + \text{ty})\times\text{Width} + \text{Col}]
\end{align*}
\]

where \( p \) is the sequence number of the current phase
__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, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!
- Each __syncthread() can reduce the number of active threads for a block
  - More thread blocks can be advantageous
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)
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 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
  - $M[\text{Row}][p\times\text{TILE\_WIDTH}+tx]$
  - $M[\text{Row}\times\text{Width} + p\times\text{TILE\_WIDTH}+tx]$

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

- Each thread loads
  - $N[p \cdot \text{TILE WIDTH} + ty][\text{Col}]$
  - $N[(p \cdot \text{TILE WIDTH} + ty) \cdot \text{Width} + \text{Col}]$

- Need to test
  - $(p \cdot \text{TILE WIDTH} + ty < \text{Width}) \&\& (\text{Col} < \text{Width})$
  - 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) {
8

++     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;
++     }
++     __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;
17} /* 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 \( l \)
TILED MATRIX MULTIPLY
CONTROL DIVERGENCE
Performance Impact of Control Divergence

- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation
  - For example, see the tile loading code below:

```c
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;
}
```
Two types of blocks in loading M Tiles

- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way
Analysis of Control Divergence Impact

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of 100/16)

- There are 49 thread blocks (7 in each dimension)
Control Divergence in Loading M Tiles

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence
Control Divergence in Loading M Tiles (Type 2)

- Type 2: the 7 block assigned to load the bottom tiles, with a total of 56 (8*7) warps
- They all have 7 phases, so there are 392 (56*7) warp-phases
- The first 2 warps in each Type 2 block will stay within the valid range until the last phase
- The 6 remaining warps stay outside the valid range
- So, only 14 (2*7) warp-phases have control divergence
Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12% \( \frac{350}{2,944} \) or \( \frac{336+14}{2352+392} \)
Additional Comments

- The calculation of impact of control divergence in loading N tiles is somewhat different and is left as an exercise.

- The estimated performance impact is data dependent.
  - For larger matrices, the impact will be significantly smaller.

- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant.
  - One should not hesitate to use boundary checks to ensure full functionality.

- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence.

- We will cover some algorithm patterns that naturally incur control divergence (such as parallel reduction) in the Parallel Algorithm Patterns modules.