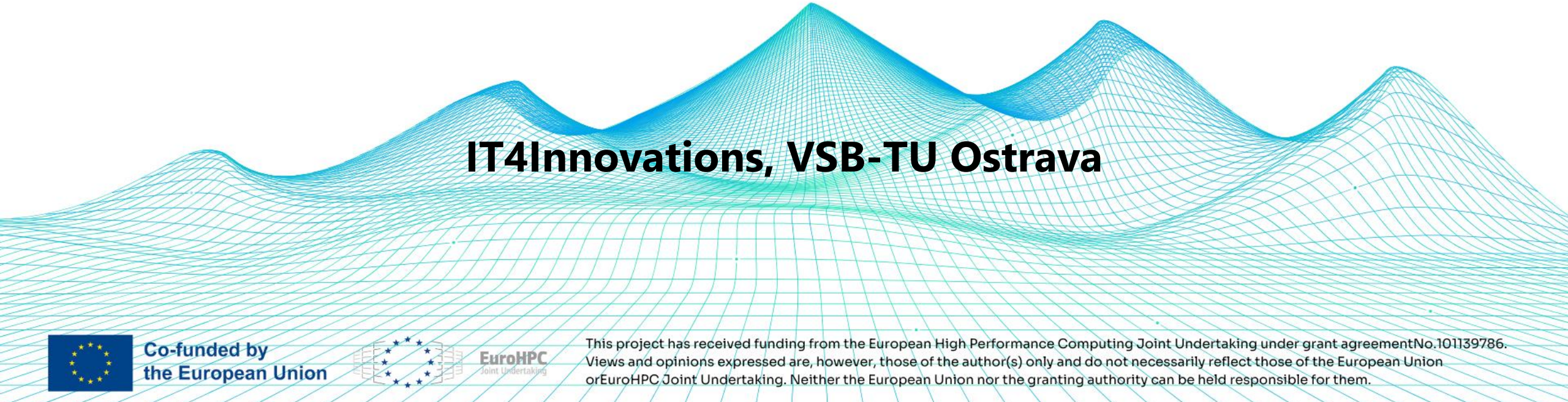
GPU Programming with CUDA

Lectures: Lubomír Říha

Hands-on: Jakub Homola, Milan Jaroš, Radim Vavřík, Filip Vaverka and Joao Barbosa

IT4Innovations, VSB-TU Ostrava



Co-funded by the European Union

EuroHPC Joint Undertaking

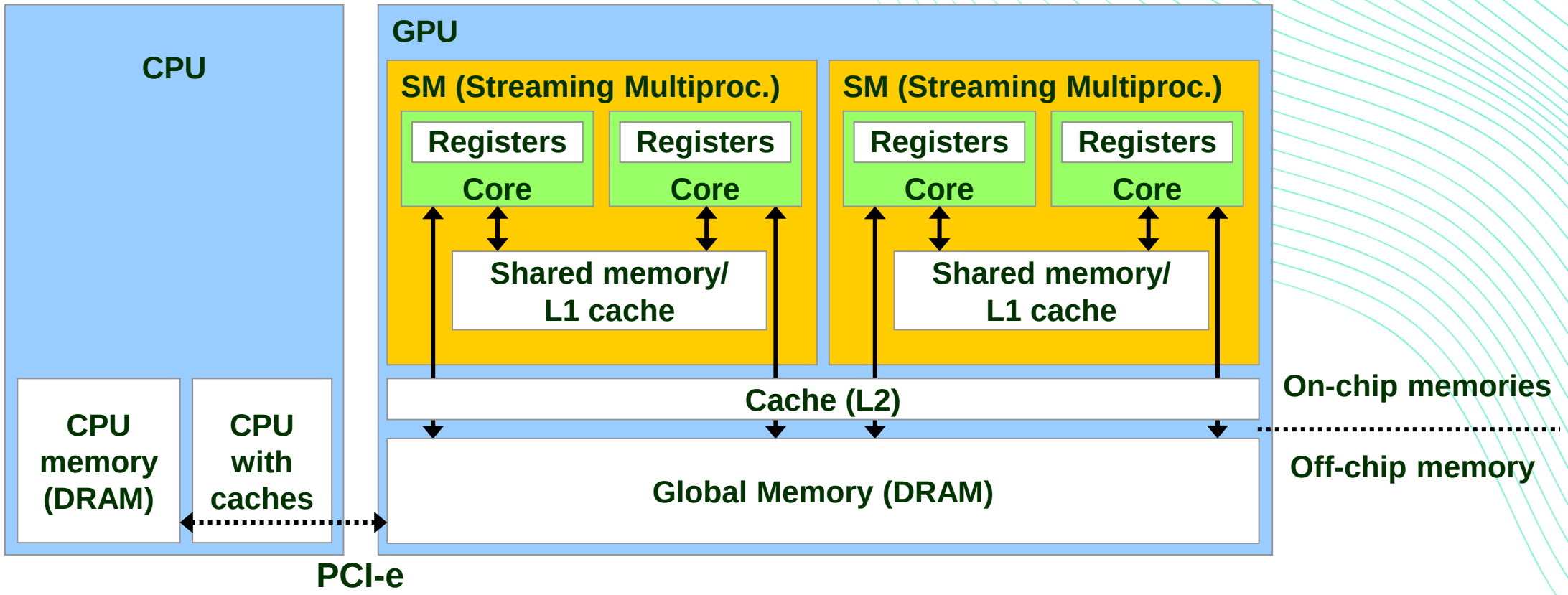
This project has received funding from the European High Performance Computing Joint Undertaking under grant agreementNo.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union orEuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

EPICURE

Unlocking European-level HPC Support

CUDA Memories

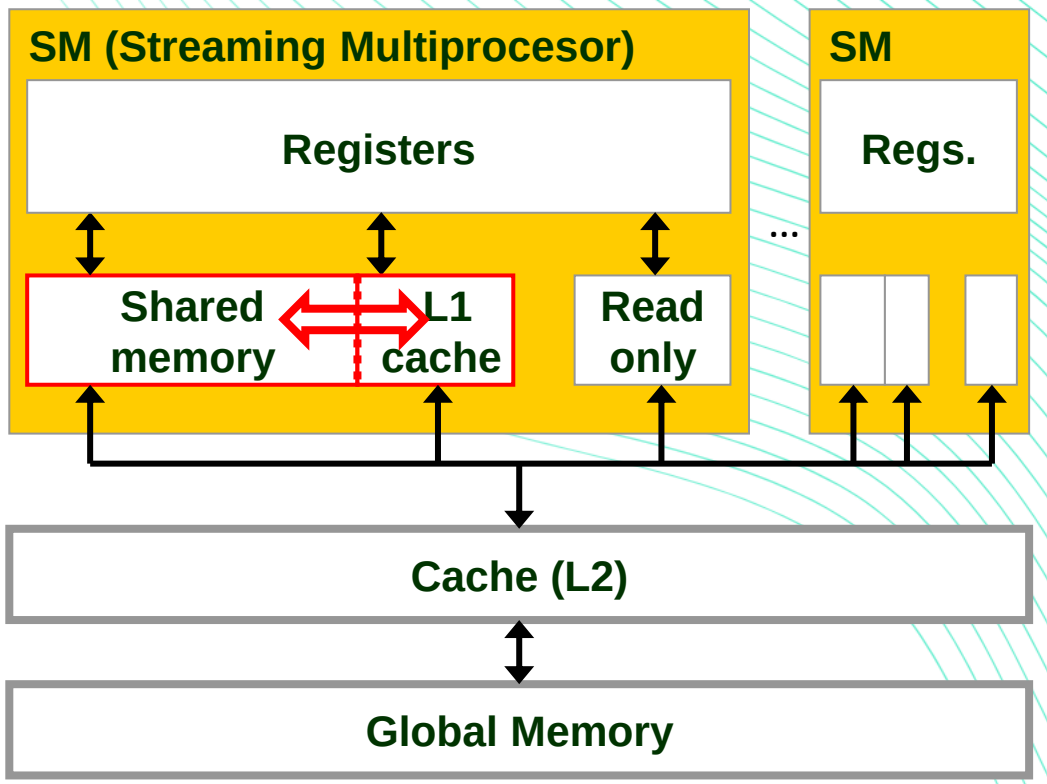
CUDA Memories
Hardware View



CUDA Memories Hardware View

Memory hierarchy in Ampere generation (GA100)

- **Registers**
  - 256 kB per SM
  - Storage local to each threads
- **Shared memory and L1 cache (192KB total)**
  - **configurable** up to 164KB for SM;
    - remainder for L1 Cache
  - low latency: ~22 cycles (SM), 34 cycles (L1d)
  - high bandwidth: ~18 TB/s
- **Read-only cache**
  - Up to 128 kB per SM
- **L2 - 40 MB**
  - latency: ~ 200 or 350 cycles
  - BW: ~ 7000 GB/s
- **Global memory – 40 or 80 GB HBM2**
  - BW ~ 1500 GB/s



Peter Van Sandt, Citadel, Zhe Jia, Citadel: Dissecting the Ampere GPU Architecture through Microbenchmarking
https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/

CUDA Memories Caches

Why do GPU have caches?

In general, not for cache blocking

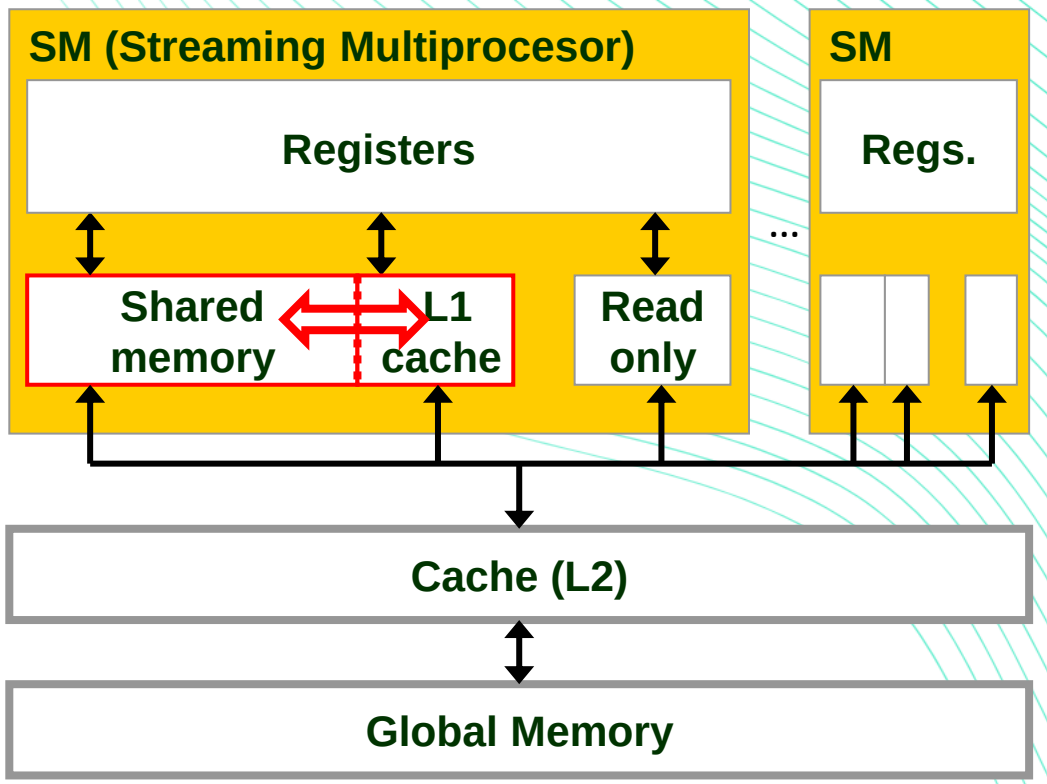
- 100s ~ 1000s of threads running per SM
- tens of thousands of threads sharing the L2 cache
- L1, L2 are small per thread.
- **Example**: at 2048 threads/SM, with 80 SMs:
  - 64 bytes L1 per thread,
  - 38 Bytes L2 per thread.

Shared Memory is usually better option to cache data explicitly:

- user managed -> no evictions out of user control.

Caches on GPUs are useful for:

- “Smoothing” irregular, unaligned access patterns
- Caching common data accessed by many threads
- Faster register spills, local memory
- Fast atomics
- Codes that don’t use shared memory (naïve code, OpenACC, …)

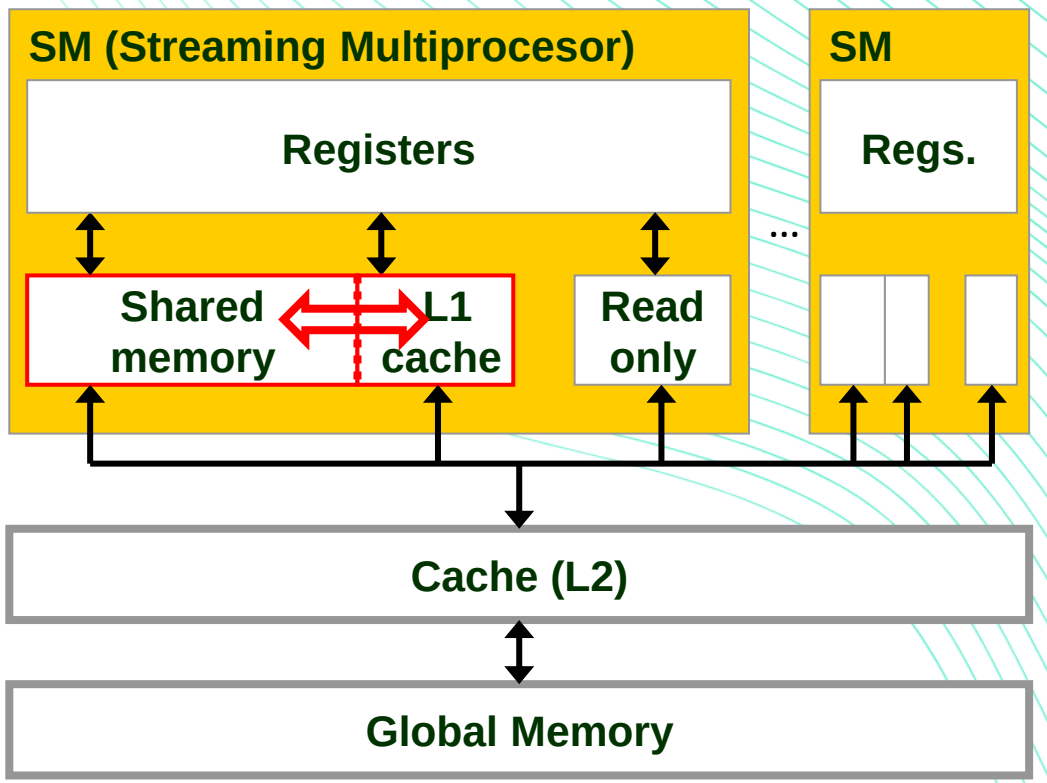


Source: NVIDIA https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf
Microbenchmarks: https://github.com/passlab/CUDAMicroBench , DOI Bookmark: 10.1109/IPDPSW52791.2021.00068

CUDA Memories
Constant memory

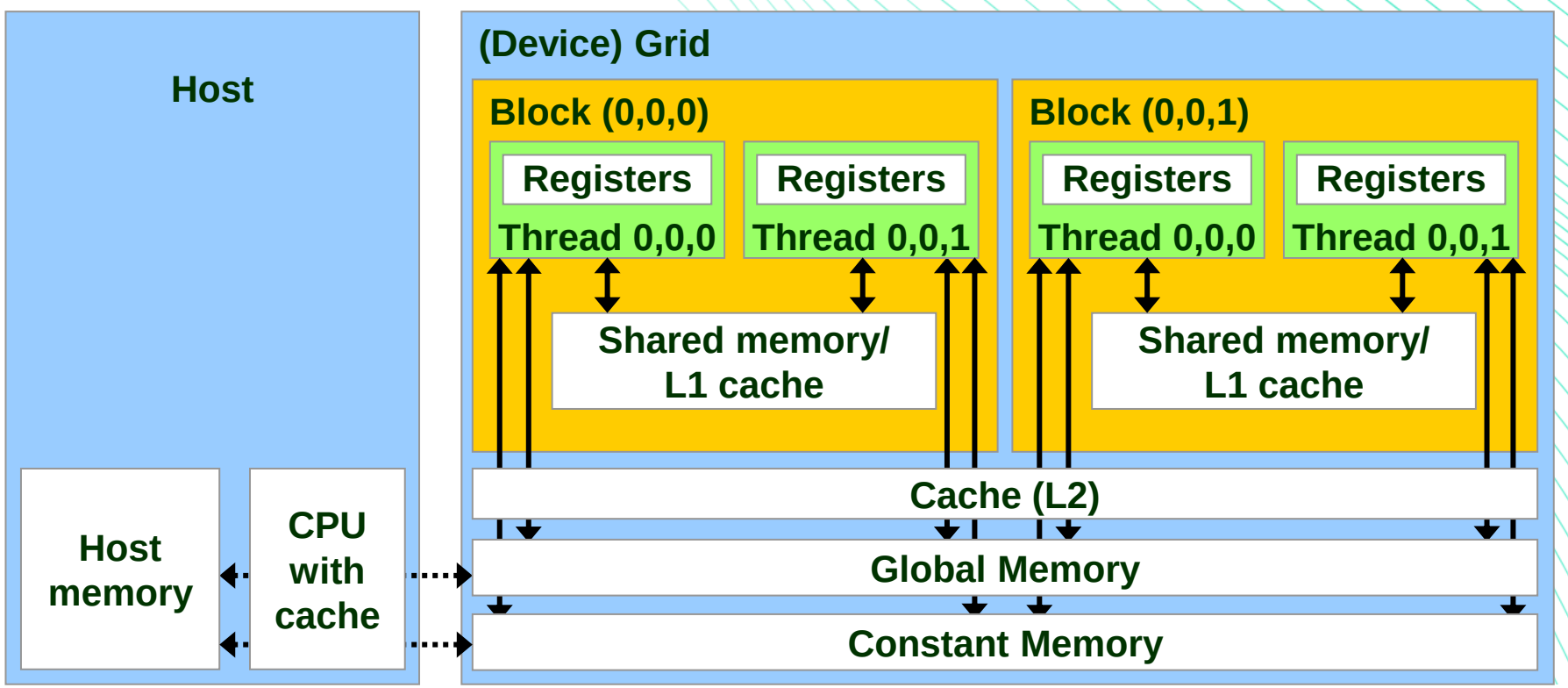
Constant memory

- Read-only variables or arrays of global scope
- Qualified with __**constant**__ keyword
- Capacity 64 KiB
- Cached in 8 KiB constant (read-only) cache
- **Very fast if all threads within a warp read the same address**
  - If the address is cached, throughput of constant cache
  - If not cached, throughput of device memory
- If different threads read different addresses, the accesses are serialized
- Example use: stencil coefficients



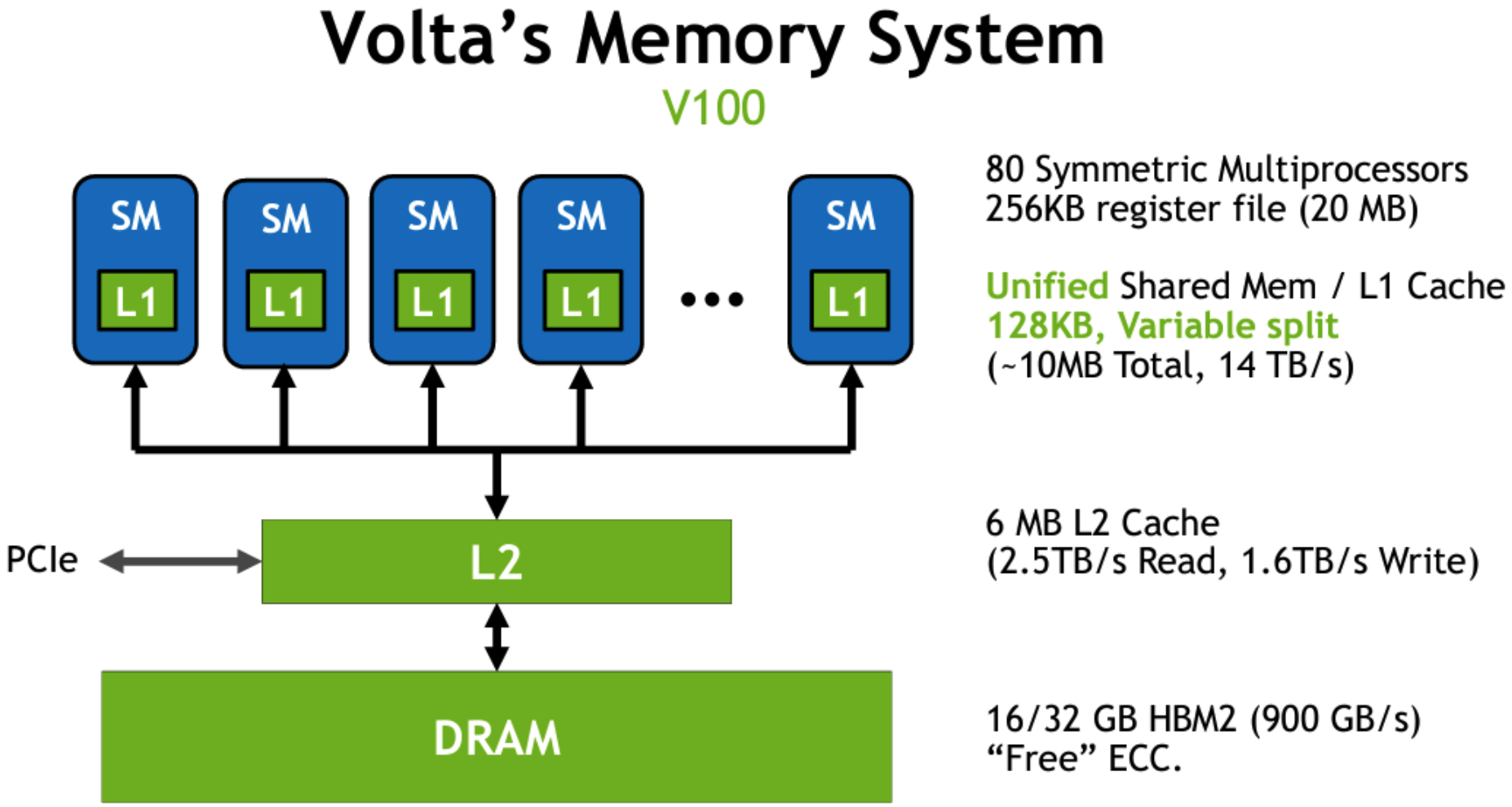
CUDA Memories Programmer View

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



Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

CUDA Memories
Hardware View



Source: NVIDIA https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

EPICURE

Unlocking European-level HPC Support

Global Memory

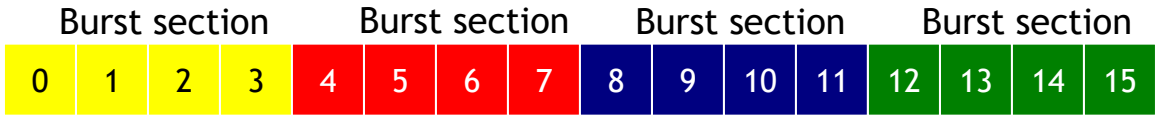
CUDA Memories
Global Memory Efficient Access

Memory Coalescing

- memory coalescing is important for effectively utilizing memory bandwidth in CUDA
  - its origin in the DRAM burst
- for good performance, CUDA memory access must be coalesced

DRAM Burst – A System View

- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the GPU (or CPU)
- **Basic example:**
  - a 16-byte address space, 4-byte burst sections



  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA Memories
Global Memory Efficient Access

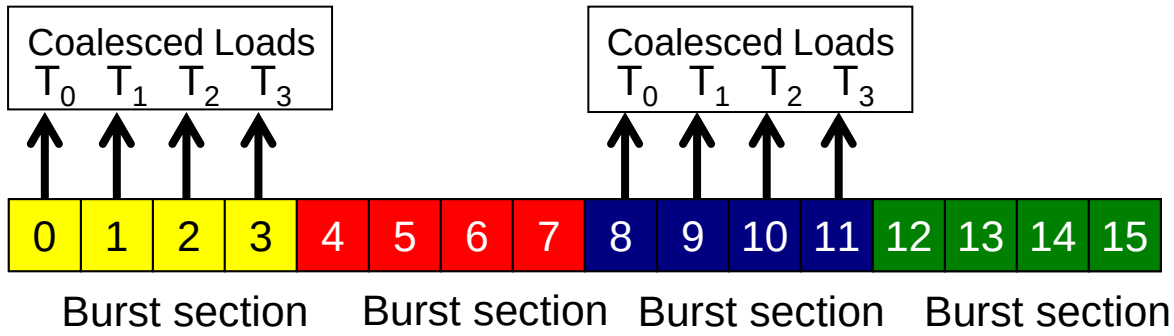
Memory Coalescing

- when all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

How to judge if an access is coalesced?

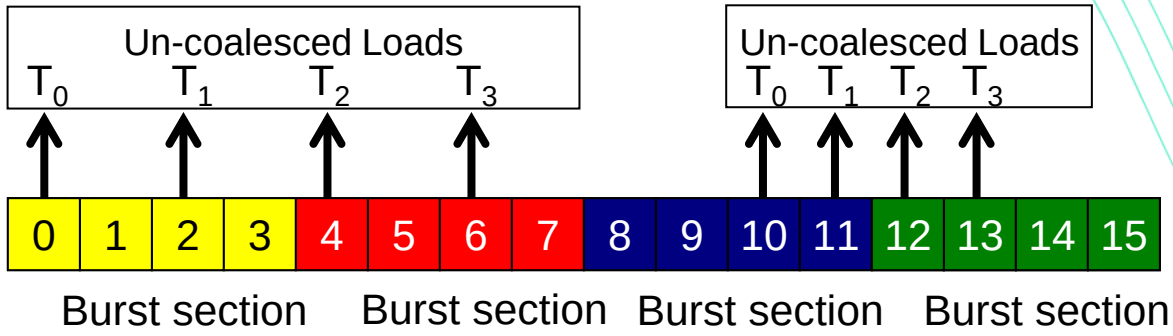
- Accesses in a warp are to consecutive locations if the index in an array access is in the form of:

A[(expression with terms independent of threadIdx.x) + threadIdx.x];



Un-coalesced Accesses

- When the accessed locations spread across burst section boundaries:
  - multiple DRAM requests are made
- Some of the bytes accessed and transferred are not used by the threads

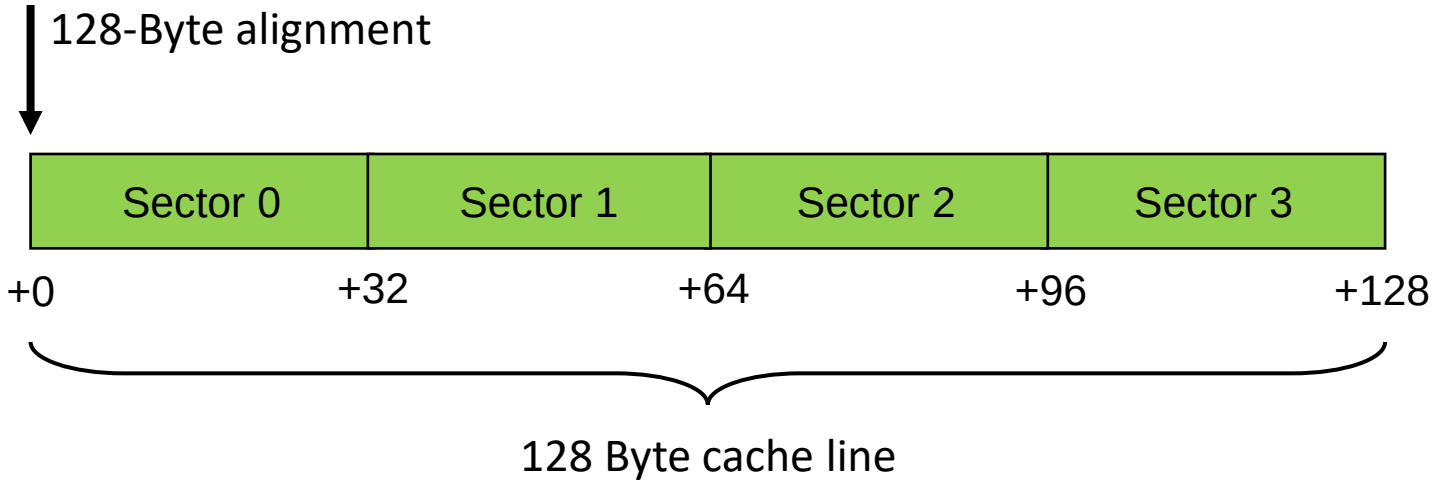


Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

CUDA Memories
Global Memory Efficient Access

Cache lines and Sectors

- Moving data between L1, L2 and DRAM



Memory access granularity

- **32 Bytes – 1 sector**
  - for Maxwell and Pascal
- **Volta architecture**
  - **64 Bytes**
  - 2 sectors is default – second sector is prefetched
- **Ampere architecture**
  - **granularity can be set to**
    - **32, 64 and 128 Bytes**

Cache line size

- **128 Bytes** – made of 4 sectors

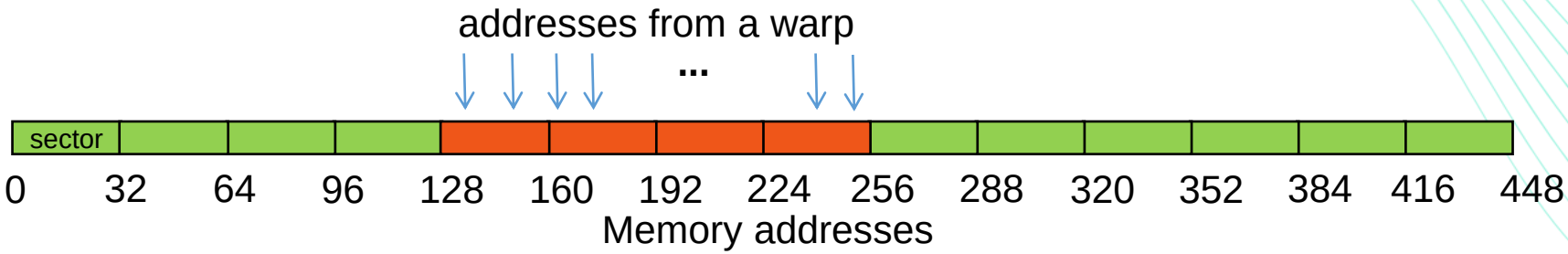
Cache management granularity

- 1 cache line

```
cudaDeviceSetLimit(cudaLimitMaxL2FetchGranularity, 32)
```

https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf

CUDA Memories
Global Memory Efficient Access

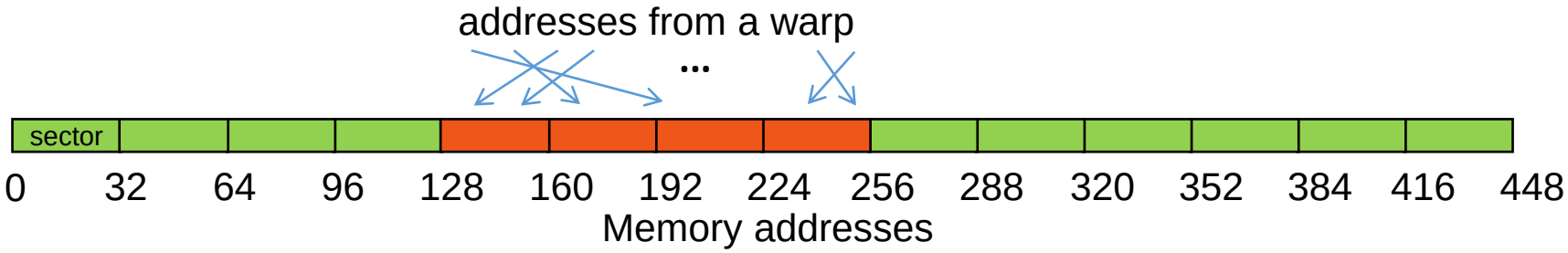


Scenario 1:

- Warp requests 32 aligned, **consecutive** 4-byte words

Addresses fall within 4 sectors

- Warp needs 128 bytes
- 128 bytes move across the bus
- **Bus utilization: 100%**



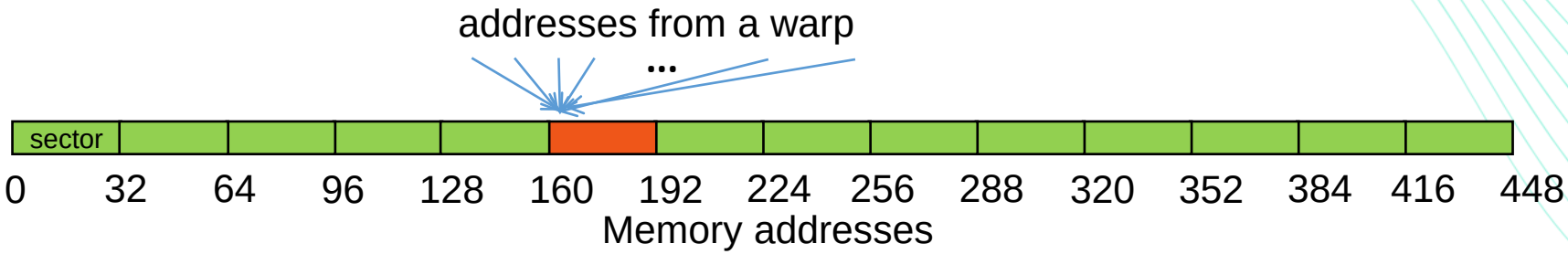
Scenario 2:

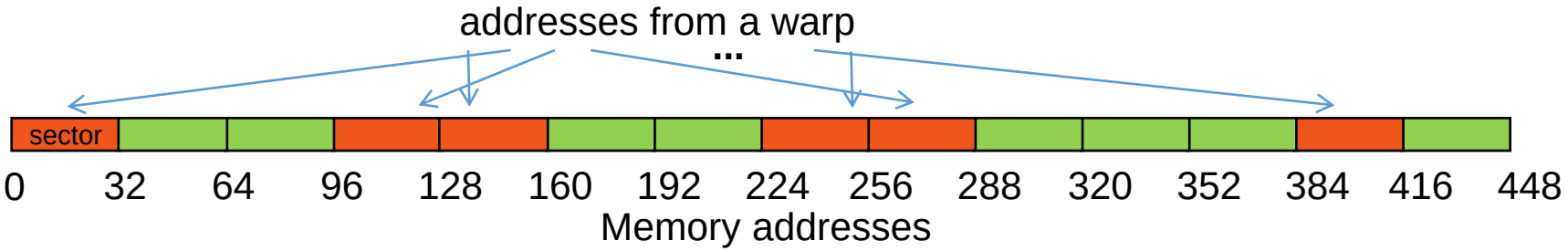
- Warp requests 32 aligned, **permuted** 4-byte words

Addresses fall within 4 sectors

- Warp needs 128 bytes
- 128 bytes move across the bus
- **Bus utilization: 100%**

CUDA Memories
Global Memory Efficient Access





Scenario 3:

- All threads in a warp request the same 4-byte word

Addresses fall within 1 sector

- Warp needs 4 bytes
- 32 bytes move across the bus
- **Bus utilization: 12.5%**

Scenario 4:

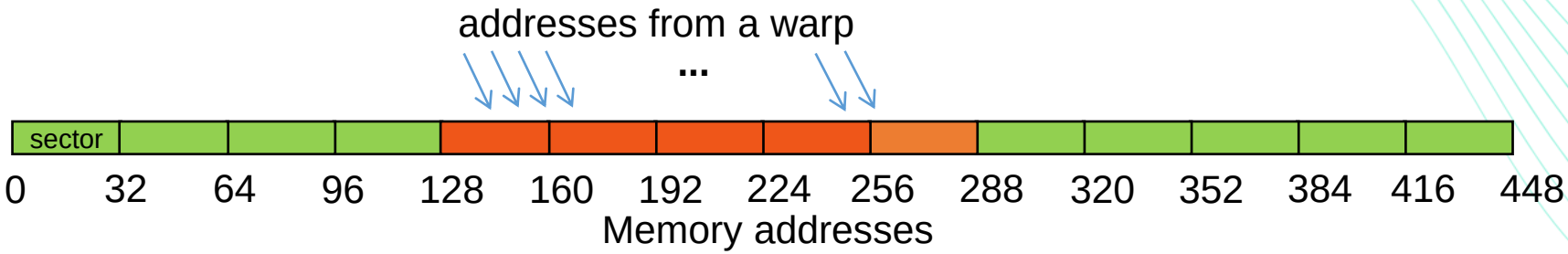
- Warp requests 32 scattered 4-byte words

Addresses fall within N sectors

- Warp needs 128 bytes
- *N**32 bytes move across the bus
- **Bus utilization: 128 / (N*32)**

https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

CUDA Memories
Global Memory Efficient Access

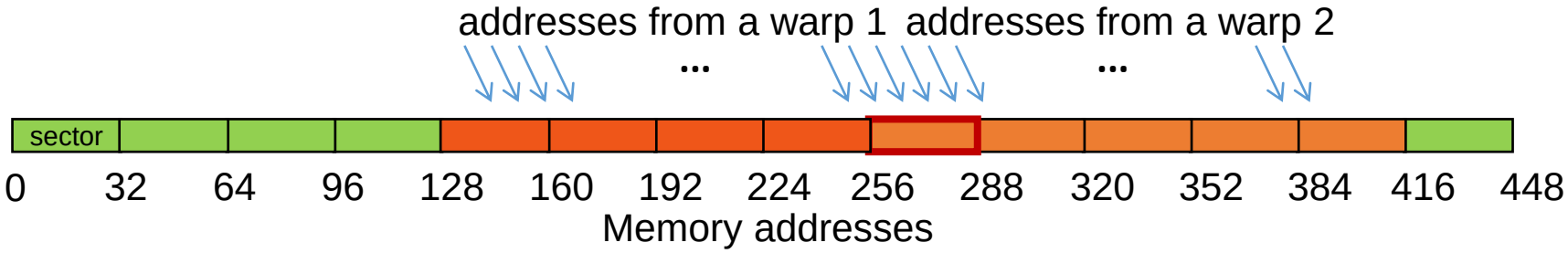


Scenario 5:

- Warp requests 32 **unaligned**, **consecutive** 4-byte words

Addresses fall within 5 sectors

- Warp needs 128 bytes
- 160 bytes move across the bus
- **Bus utilization: 80%**



Scenario 6:

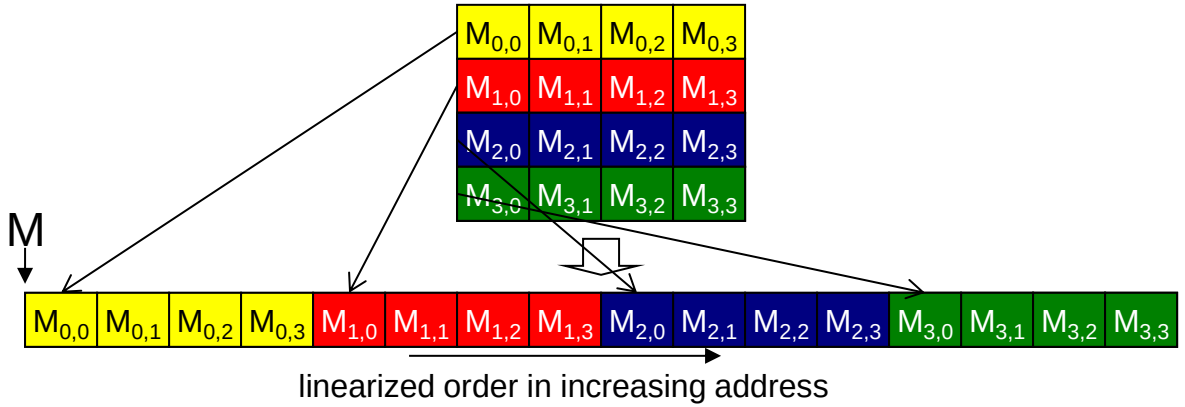
- 2 Warps request 32 **unaligned**, **consecutive** 4-byte words

Addresses fall within 9 sectors

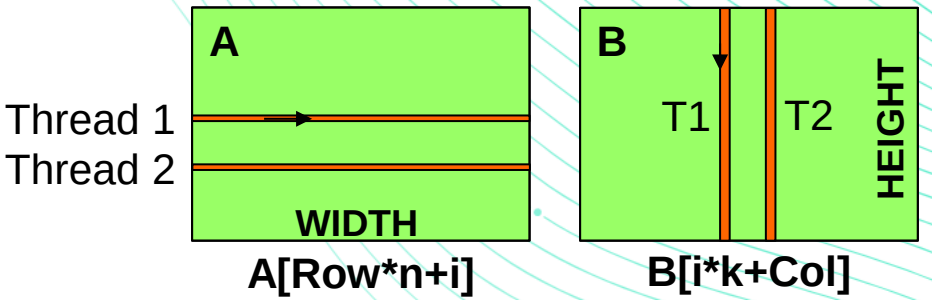
- 2 Warps need 256 bytes
- 288 or 320 bytes move across the bus (depends on presence of data in cache)
- **Bus utilization: 88% or 80%**

CUDA Memories
Global Memory Access for Matrix Multiplication

2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication

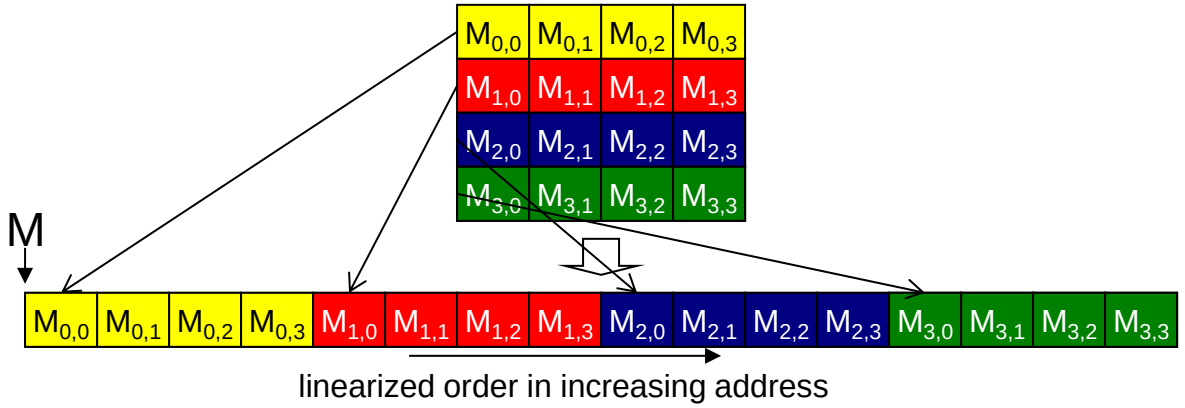


i is the loop counter in the inner product loop of the kernel code

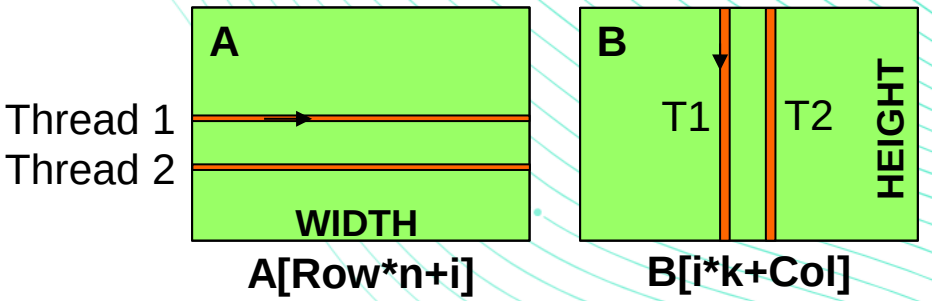
Col = blockIdx.x*blockDim.x + threadIdx.x

CUDA Memories
Global Memory Access for Matrix Multiplication

2D C Array in Linear Memory Space



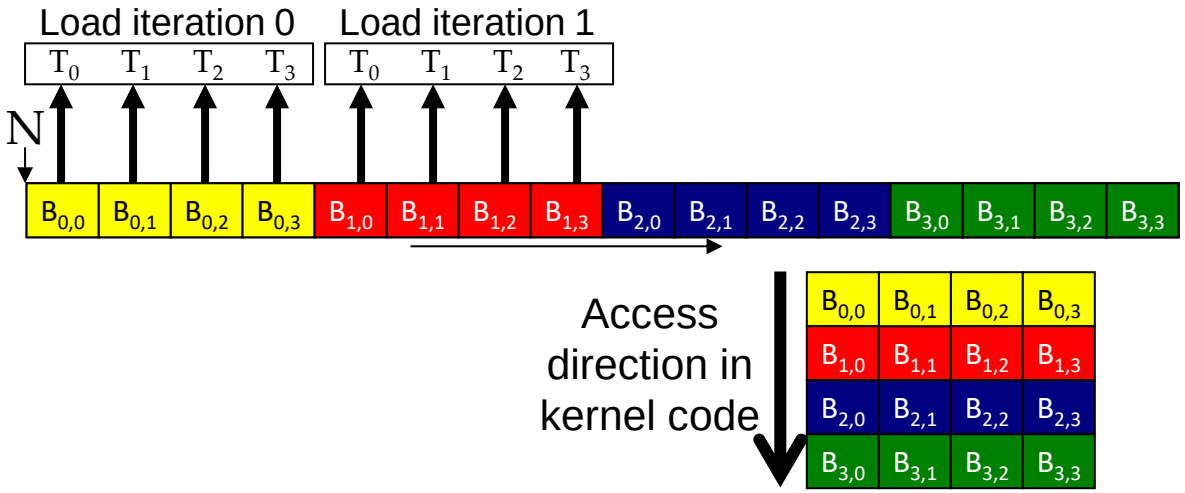
Two Access Patterns of Basic Matrix Multiplication



i is the loop counter in the inner product loop of the kernel code

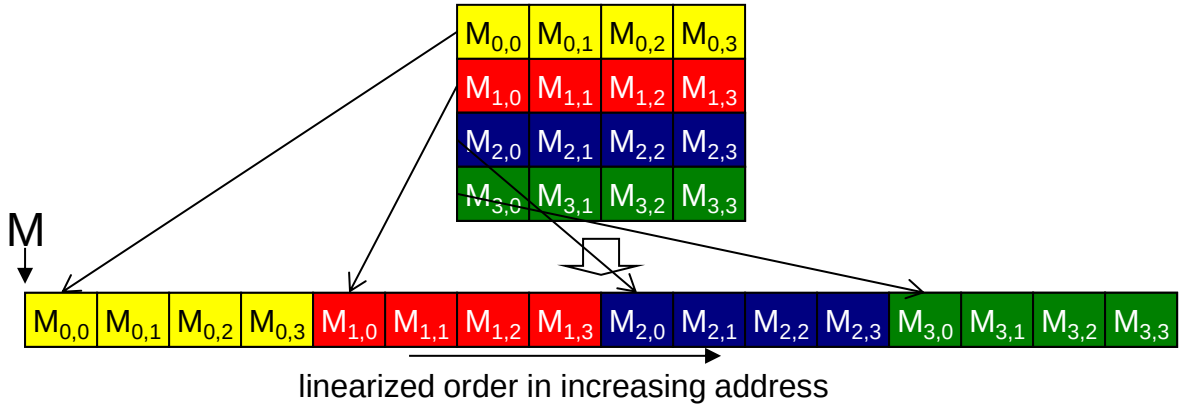
B[i*k+Col] and Col = blockIdx.x*blockDim.x + threadIdx.x

Matrix B accesses are coalesced

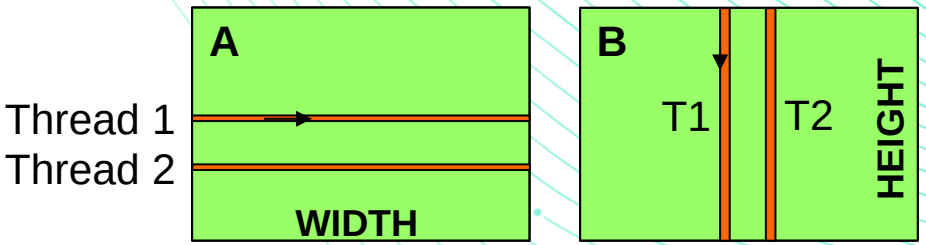


CUDA Memories
Global Memory Access for Matrix Multiplication

2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication



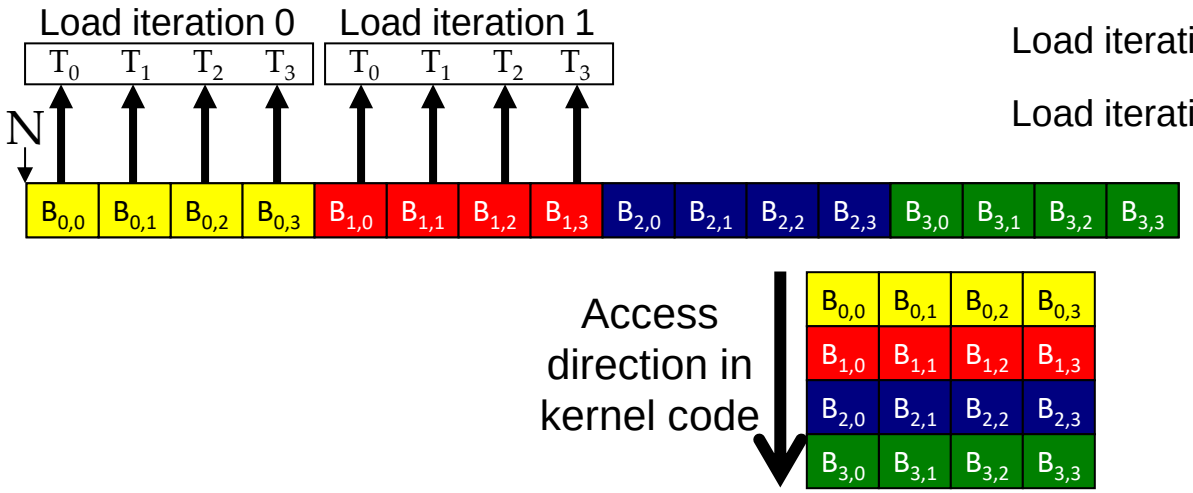
A is m × n,
B is n × k

A[Row*n+i] **B[i*k+Col]**

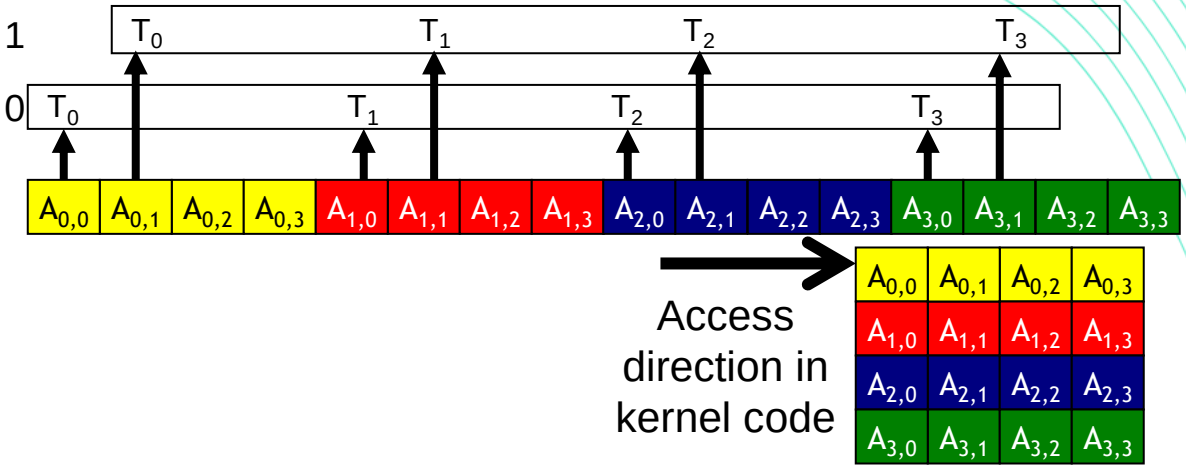
i is the loop counter in the inner product loop of the kernel code

B[i*k+Col] and Col = blockIdx.x*blockDim.x + threadIdx.x
A[Row*n+i] and Row = blockIdx.y*blockDim.y + threadIdx.y

Matrix B accesses are coalesced



Matrix A Accesses are Not Coalesced



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

Unlocking European-level HPC Support

Hands-on Matrix Sum

EPICURE

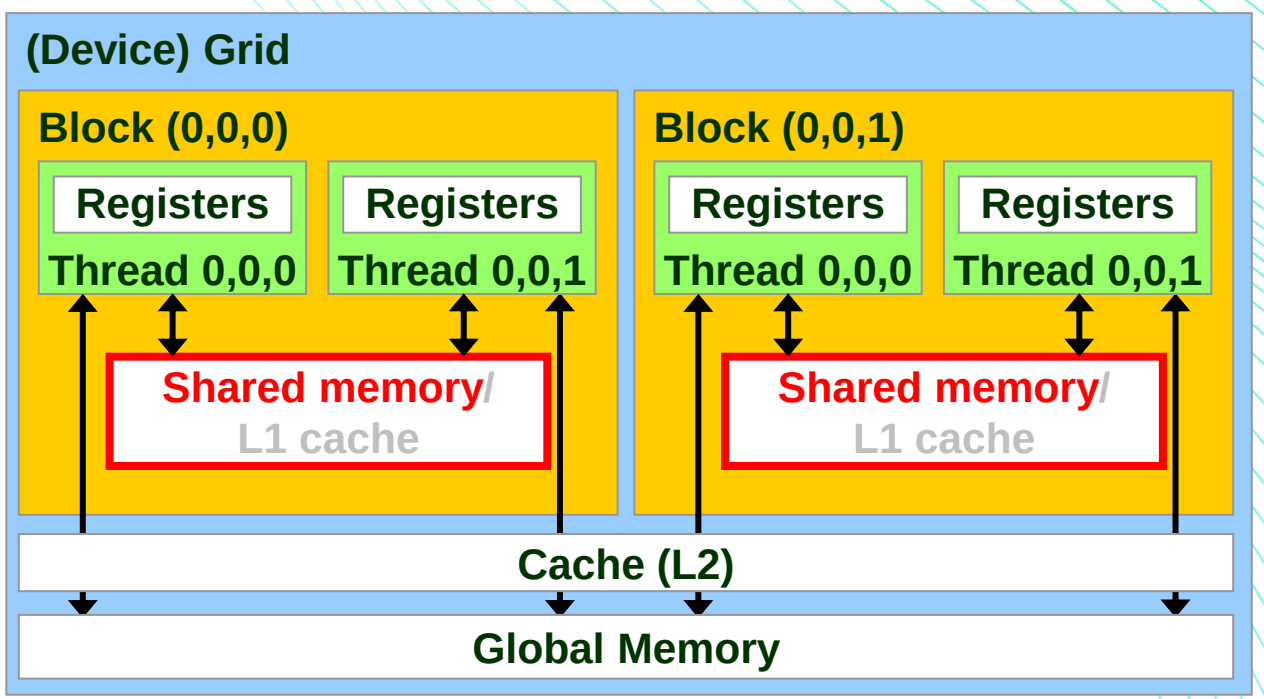
Unlocking European-level HPC Support

Shared Memory

CUDA Memories
Shared Memory in CUDA

Special type of memory whose contents are explicitly defined and used only in the kernel source code

- **one independent chunk in each SM**
- **accessed at much higher speed** (in both latency and throughput) than global memory
- **scope of access and sharing – all threads in a block**
- lifetime – thread block, contents will disappear after the corresponding block (all threads) finishes and terminates execution
- accessed by memory load/store instructions
- a form of scratchpad memory in computer architecture



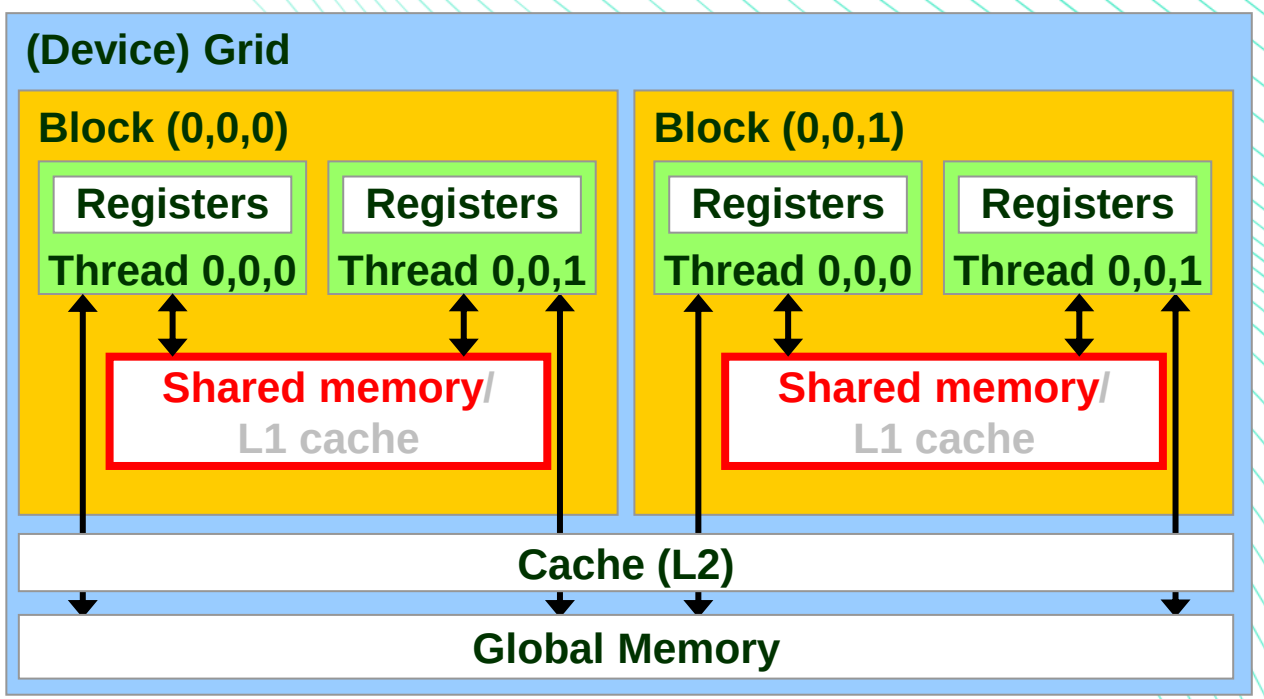
CUDA Memories
Shared Memory in CUDA

Static declaration:

```
void CUDA_Kernel_static_reserve(unsigned char
* in, unsigned char * out, int w, int h)
{

 __shared__ float ds_1D[SIZE];

 __shared__ float ds_2D[SIZE_X][SIZE_Y];

 …
}
```

Dynamic declaration:

```
void CUDA_Kernel_dynamic_reserve(unsigned char
* in, unsigned char * out, int w, int h)
{

 extern __shared__ float ds_1D[];    <-- empty brackets and use of the extern specifier
 …
}
```



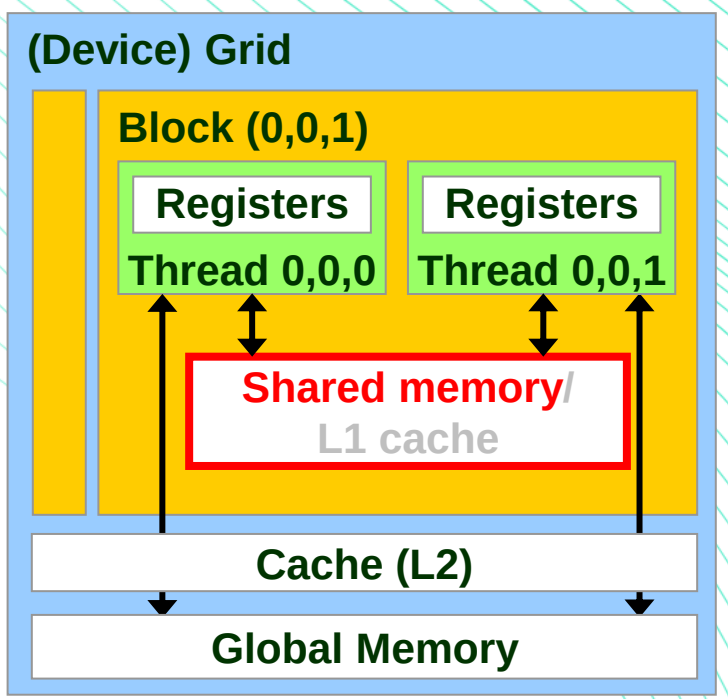

```
CUDA_Kernel_dynamic_reserve <<<1, n, SIZE*sizeof(float), stream>>>(d_in, d_out, … );
```

CUDA Memories
Shared Memory in CUDA

Dynamic declaration: multiple dynamically sized arrays in a single kernel

- you must declare a single extern unsized array as before, and
- use pointers to divide it into multiple arrays:

```
void CUDA_Kernel_dynamic_reserve(...)
{

 extern __shared__ int s[];

 int   *integerData = s;                          // nI ints

 float *floatData   = (float*)&integerData[nI];   // nF floats

 char  *charData    = (char*)&floatData[nF];      // nC chars

}
```



In the kernel launch, specify the total shared memory needed, as in the following.

```
CUDA_Kernel_dynamic_reserve <<<gridSize, blockSize,
nI*sizeof(int)+nF*sizeof(float)+nC*sizeof(char), stream>>>(...);
```

CUDA Memories
Shared Memory in CUDA

Performance benefits compared to DRAM:

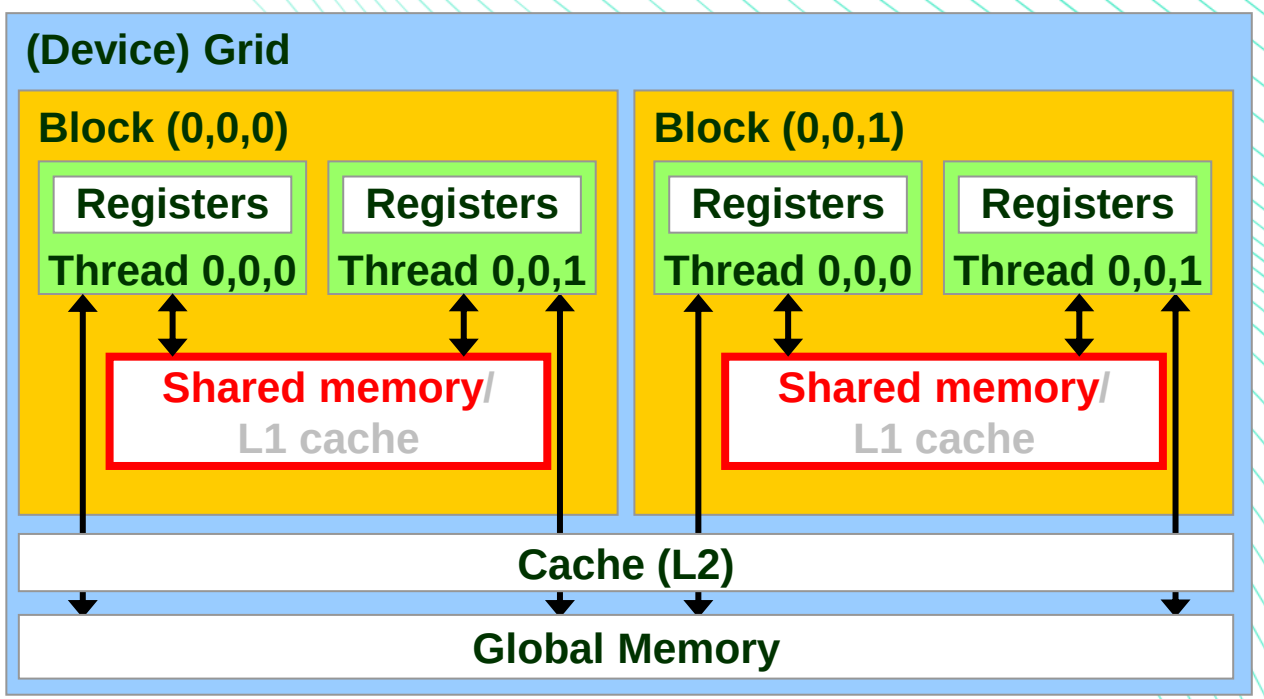
- **20-40x lower latency**
- **~15x higher bandwidth**
- accessed at 4-byte granularity
- Global Memory granularity is 32 Bytes

Ampere generation shared memory + L1 cache

- GA102 – 128 KB (used by A40 for CG/single precision)
  - Configurable up to 100 KB
- GA100 – 192 KB (used by A100 for HPC)
  - Configurable up to 164 KB

Organization

- organized in 32 banks, each 4 Bytes wide
  - bandwidth: 4 Bytes per bank per clock per SM
    - 128 Bytes per clk per SM
- successive 4-byte words go to successive banks



Bank index computation examples:

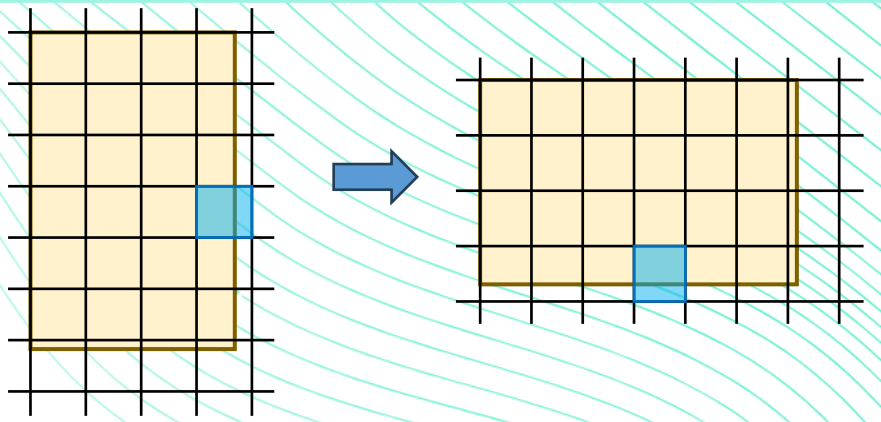
- (4B word index) % 32
- ((1B word index) / 4) % 32
- 8B word spans two successive banks

EPICURE

Unlocking European-level HPC Support

Hands-on
Matrix transpose

Hands-on: matrix transpose



- `tasks/matrix_transpose`
- Data preparation and timing is already implemented
- Implement and launch 3 kernels
  - Naïve transposition
  - Transposition with shared memory
  - Transposition with shared memory, where you avoid the bank conflicts
- Compare the timings

EPICURE
Unlocking European-level HPC Support

Coffee break

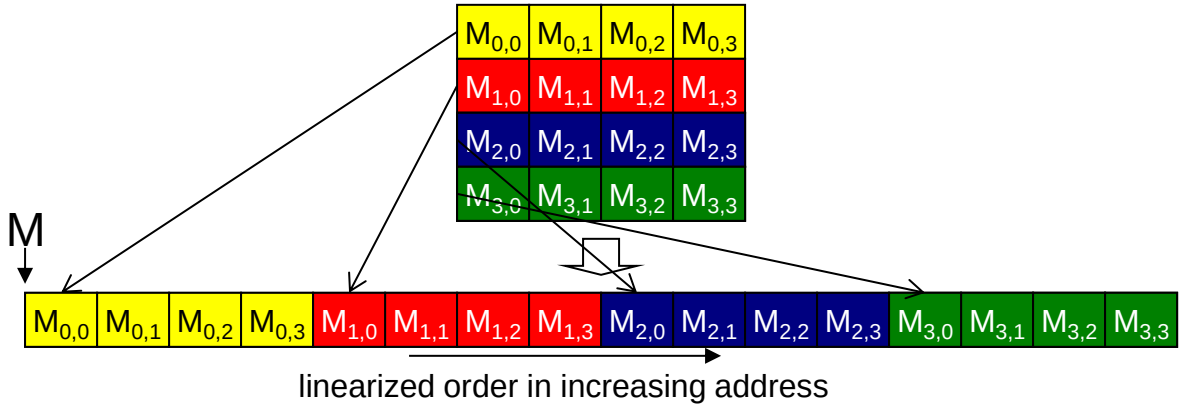
EPICURE

Unlocking European-level HPC Support

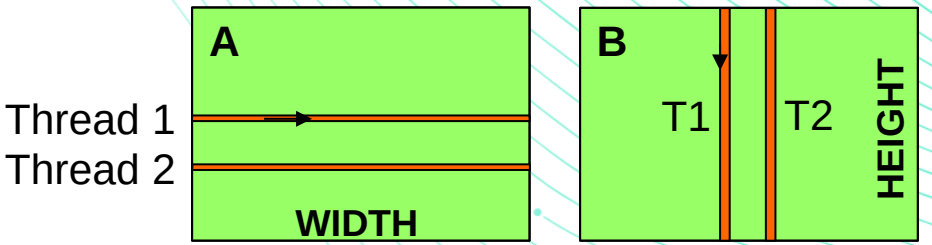
Memory and Data Locality: Tiling Technique

Motivation
Matrix Multiplication – Memory access problem

2D C Array in Linear Memory Space



Two Access Patterns of Basic Matrix Multiplication



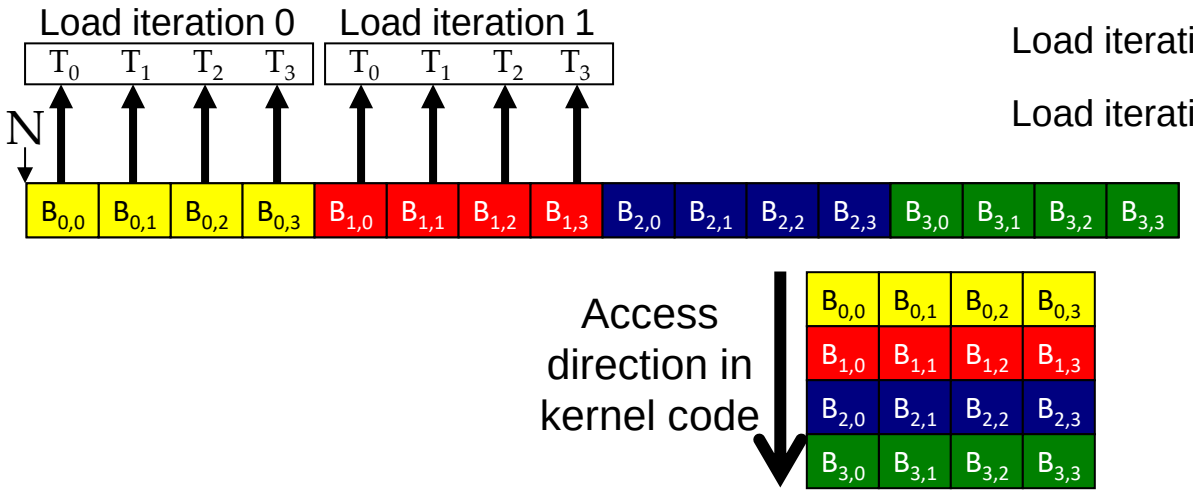
A is m × n,
B is n × k

A[Row*n+i] **B[i*k+Col]**

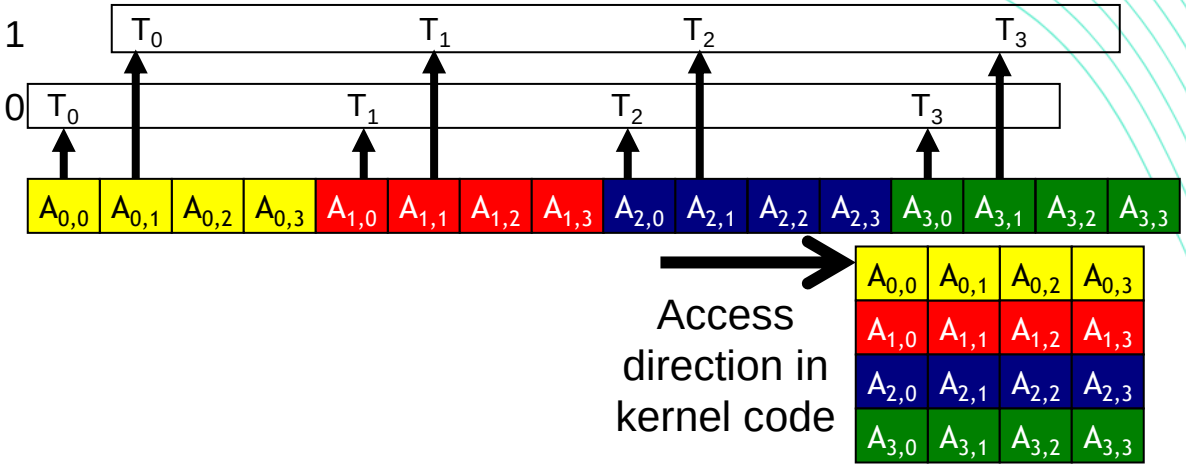
i is the loop counter in the inner product loop of the kernel code

B[i*k+Col] and Col = blockIdx.x*blockDim.x + threadIdx.x
A[Row*n+i] and Row = blockIdx.y*blockDim.y + threadIdx.y

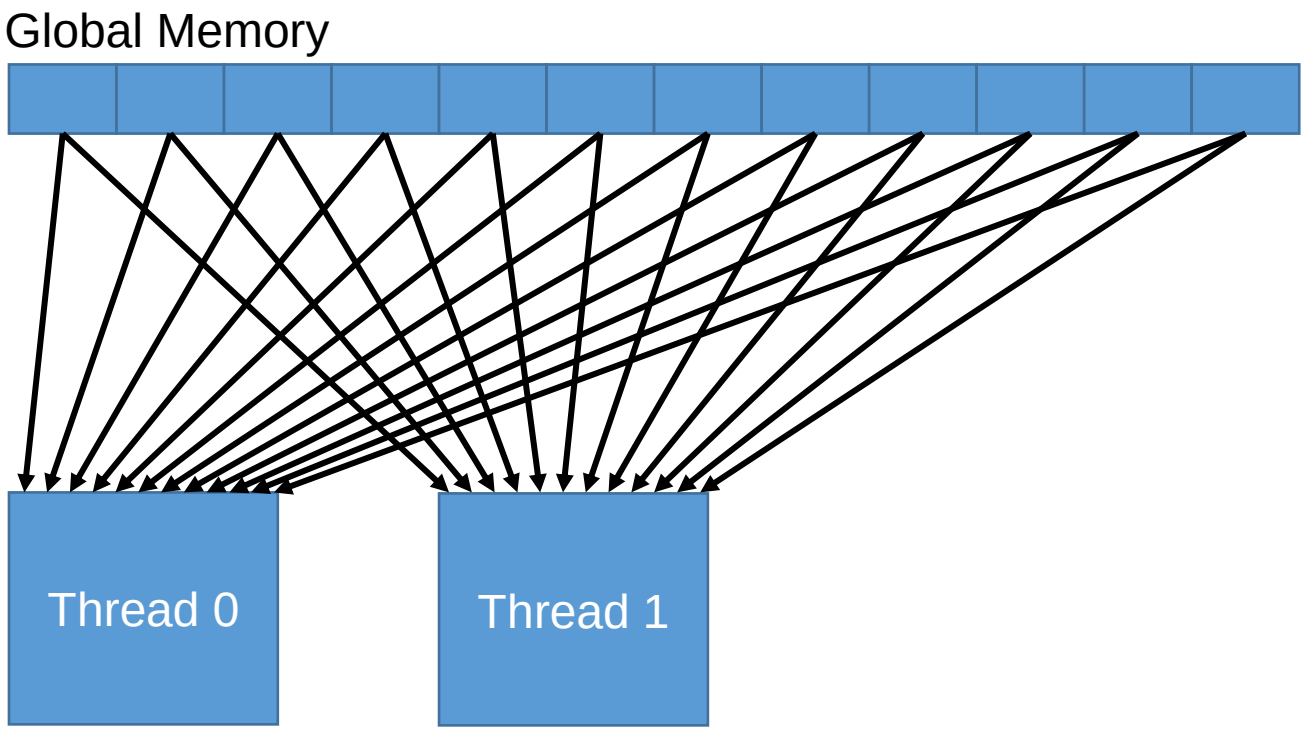
Matrix B accesses are coalesced



Matrix A Accesses are Not Coalesced

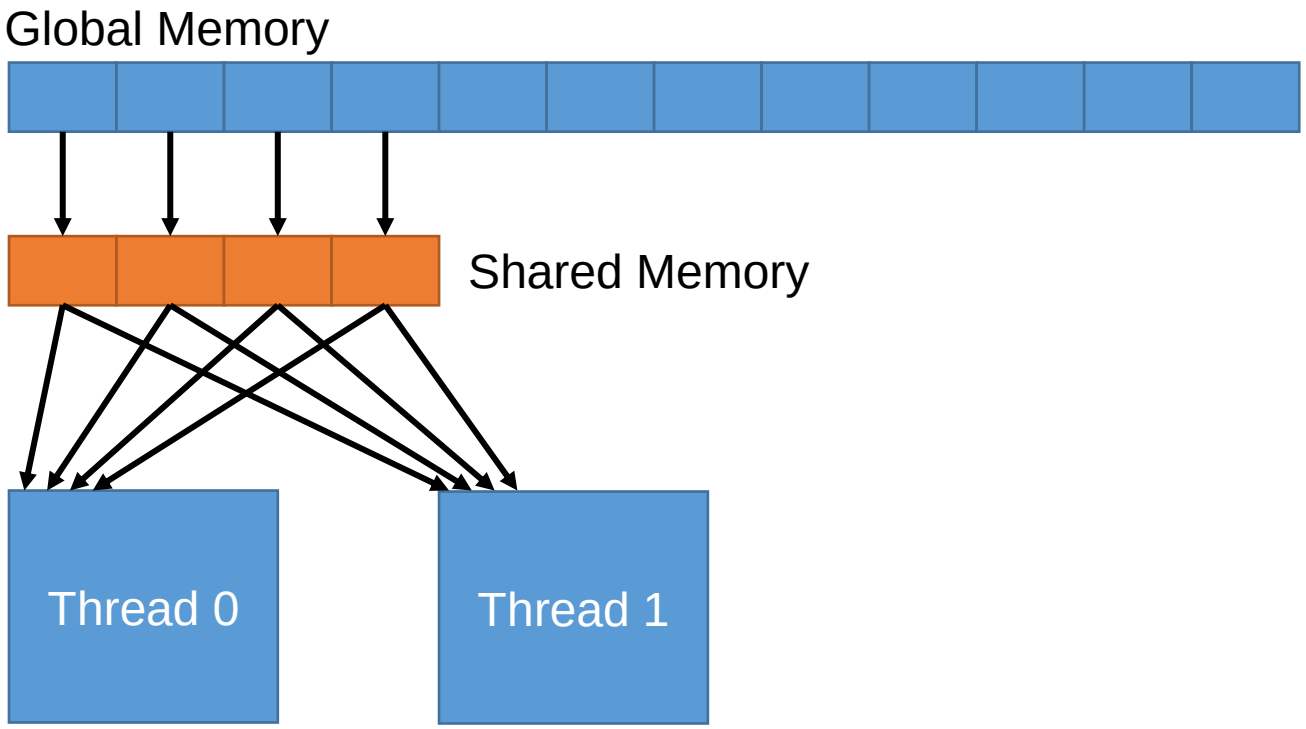


CUDA Memories
Tiling Technique



24 reads from Global Memory

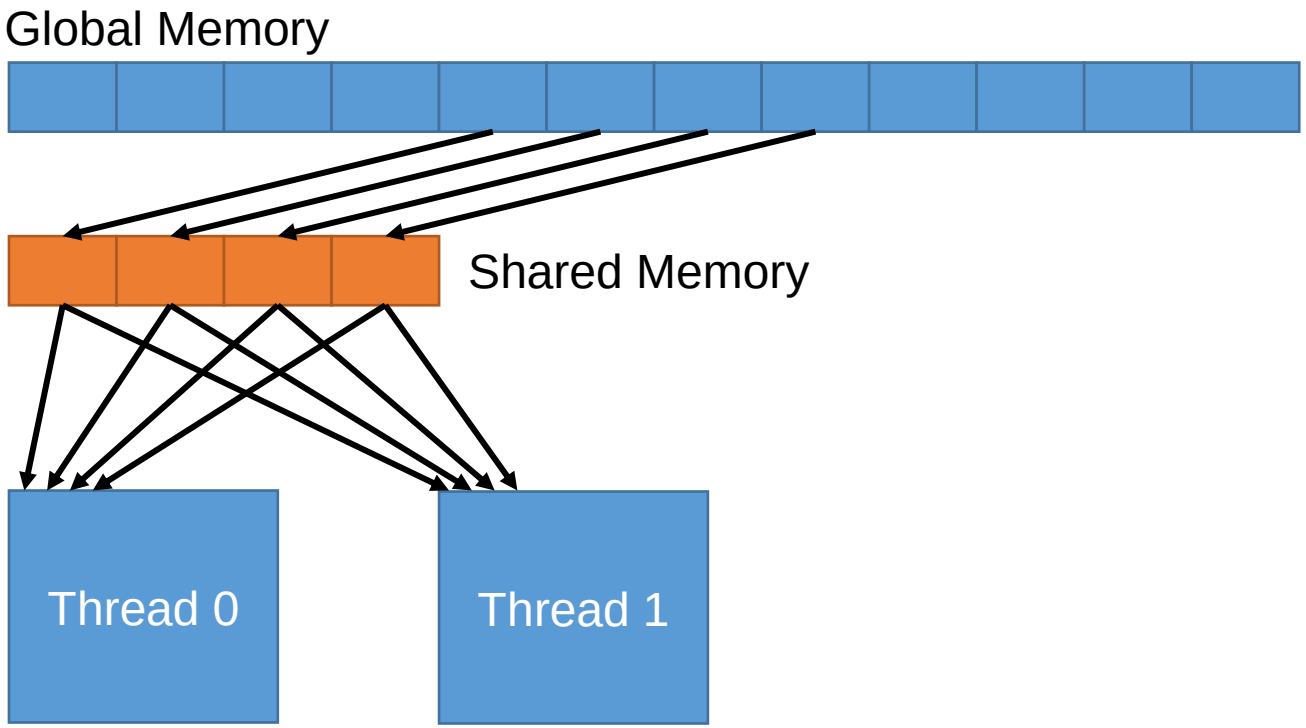
CUDA Memories
Tiling Technique



4 reads from Global Memory

8 reads from Shared Memory

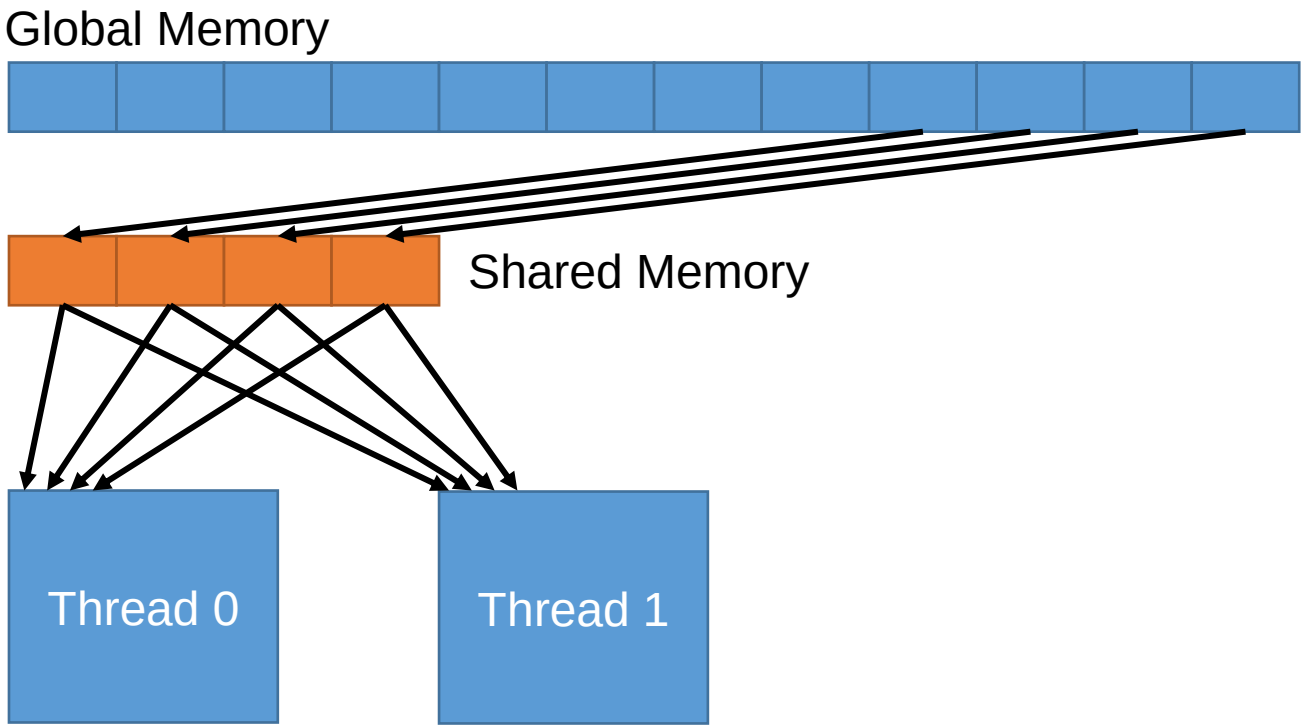
CUDA Memories
Tiling Technique



+4 reads from Global Memory (8 total)

+8 reads from Shared Memory (16 total)

CUDA Memories
Tiling Technique



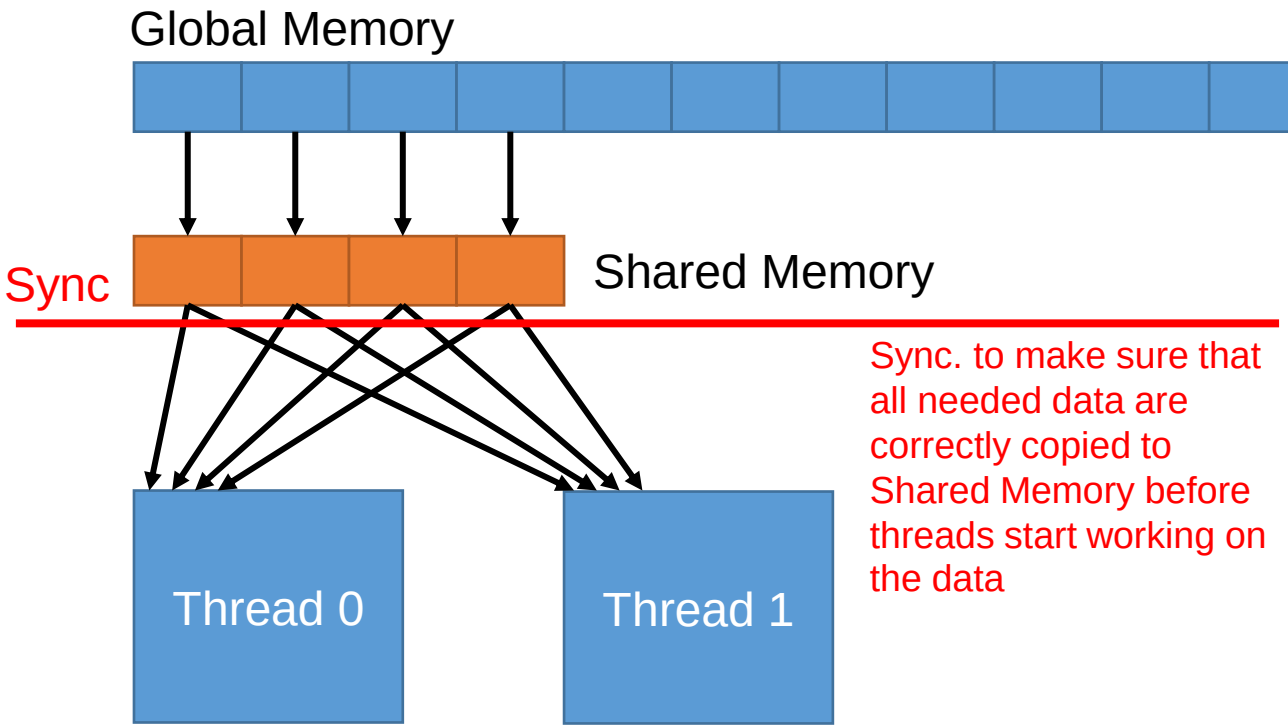
+4 reads from Global Memory (12 total)

+8 reads from Shared Memory (24 total)

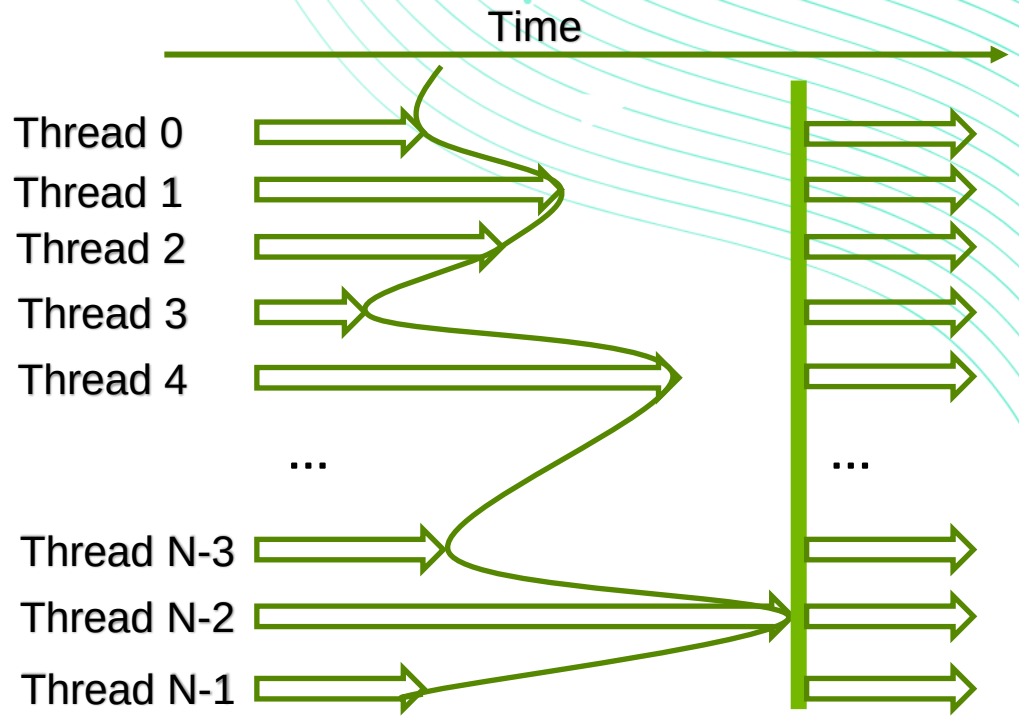
Compare to: 24 reads from Global Memory without shared memory.

CUDA Memories
Tiling Technique

Tiling needs synchronization

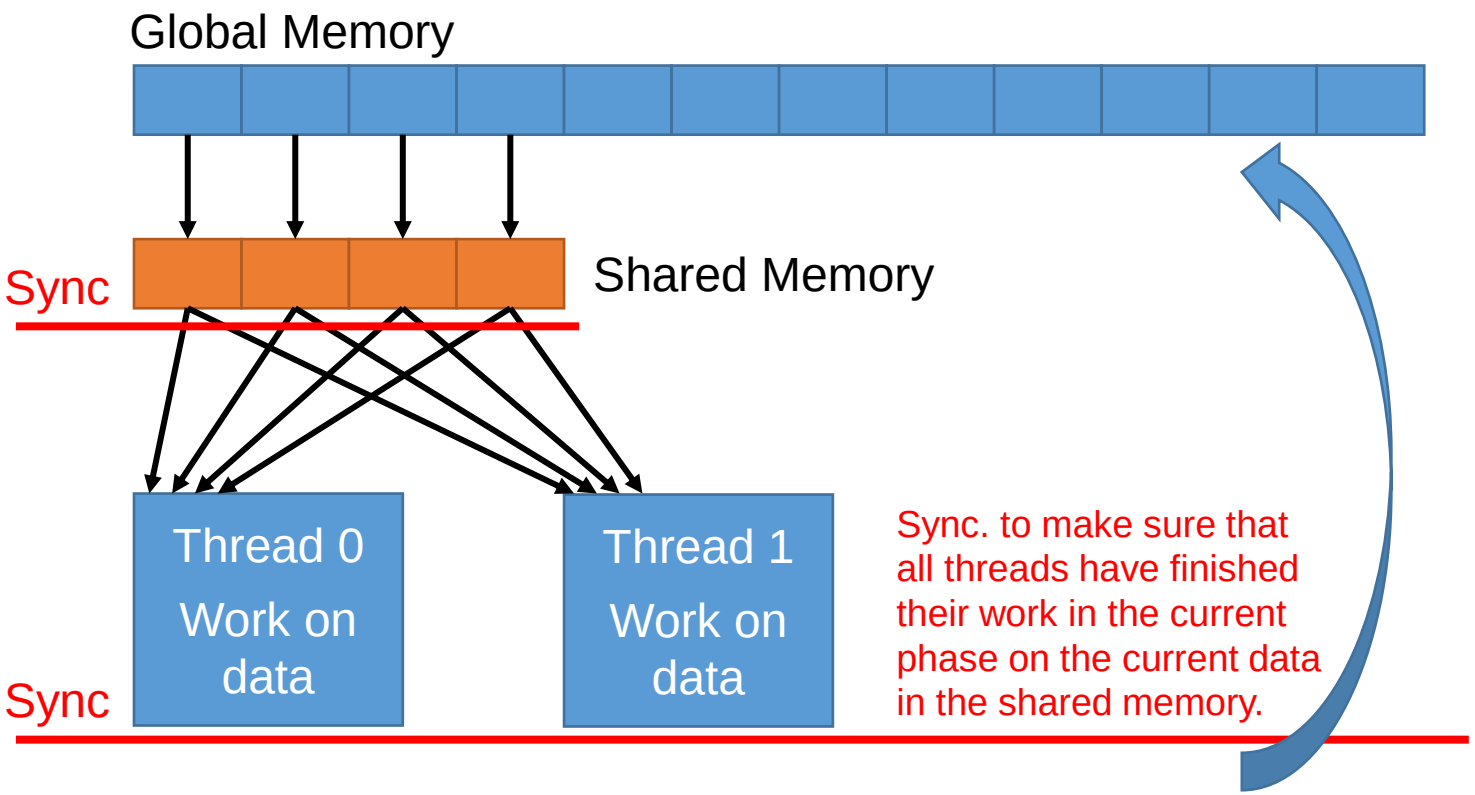


Sync. to make sure that all needed data are correctly copied to Shared Memory before threads start working on the data



CUDA Memories
Tiling Technique

Tiling needs synchronization

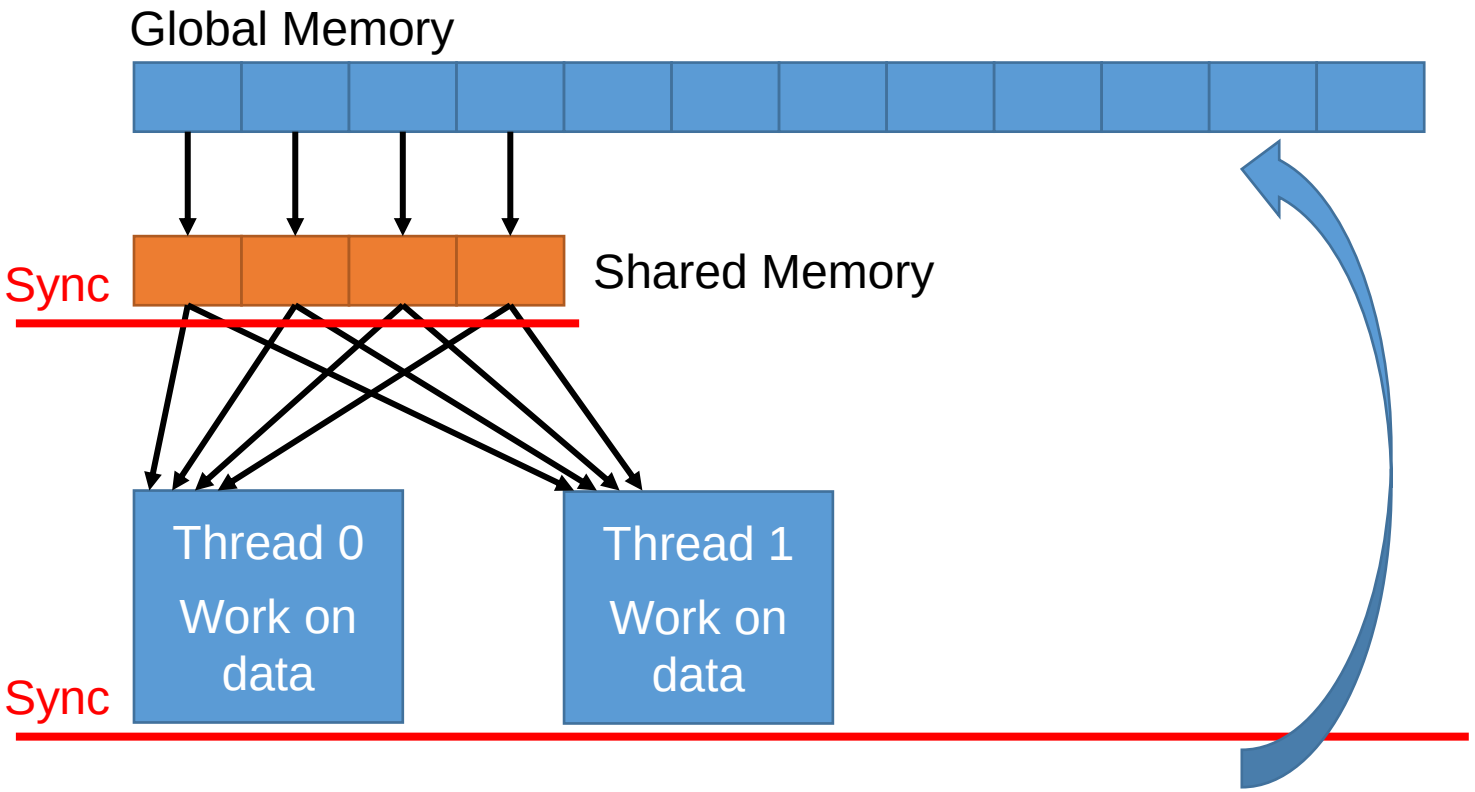


Tiling Techniques step by step

- 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

CUDA Memories
Tiling Technique

Tiling needs synchronization



Barrier Synchronization

- CUDA call to 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 of a 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

EPICURE

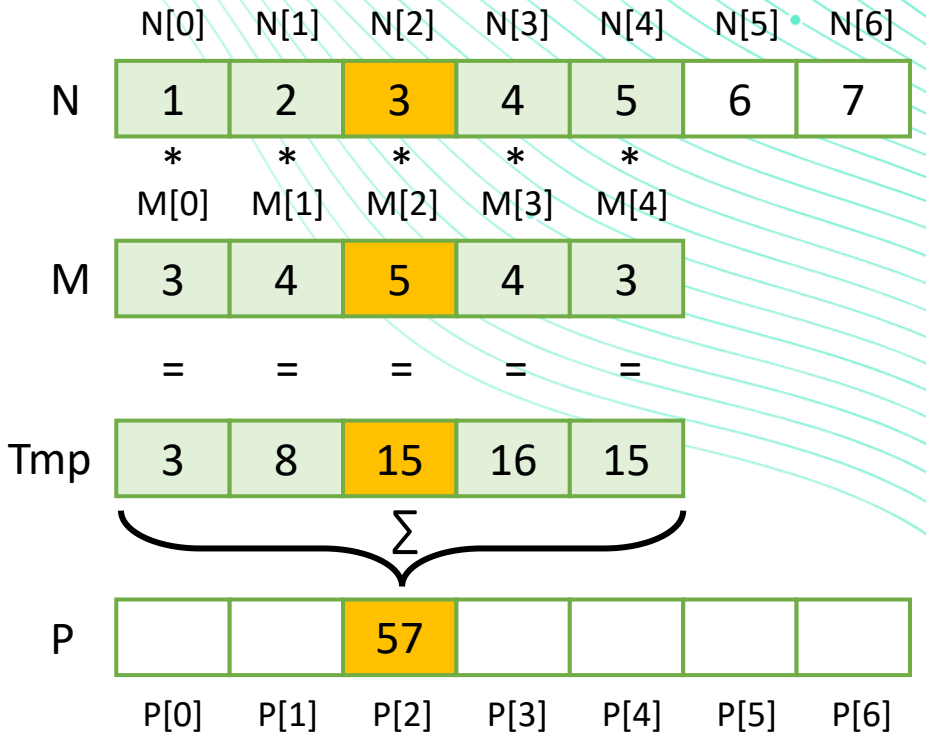
Unlocking European-level HPC Support

Parallel Computation Patterns: Stencil

Parallel Computation Patterns Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
  - we will refer to these mask arrays as convolution masks to avoid confusion.
  - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.

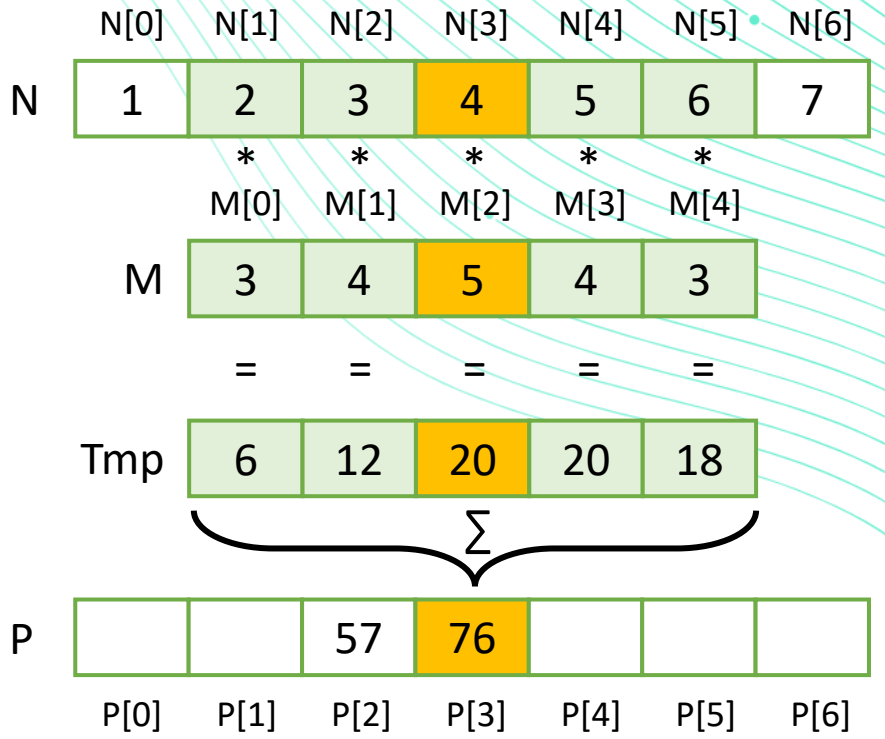


P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]

Parallel Computation Patterns Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
  - we will refer to these mask arrays as convolution masks to avoid confusion.
  - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.

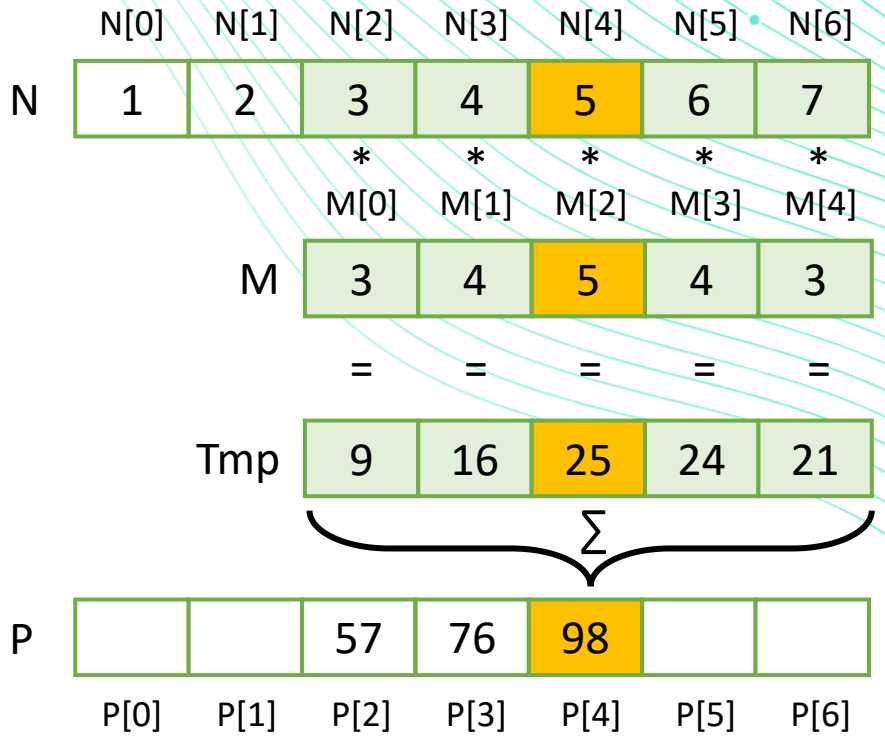


P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]

Parallel Computation Patterns Stencil

Convolution

- basic example for stencil computation pattern
- an array operation where each output data element is a weighted sum of a collection of neighboring input elements
- the weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution kernel
  - we will refer to these mask arrays as convolution masks to avoid confusion.
  - the value pattern of the mask array elements defines the type of filtering done
- Image Blur example is a special case where all mask elements are of the same value and hard coded into the source code.

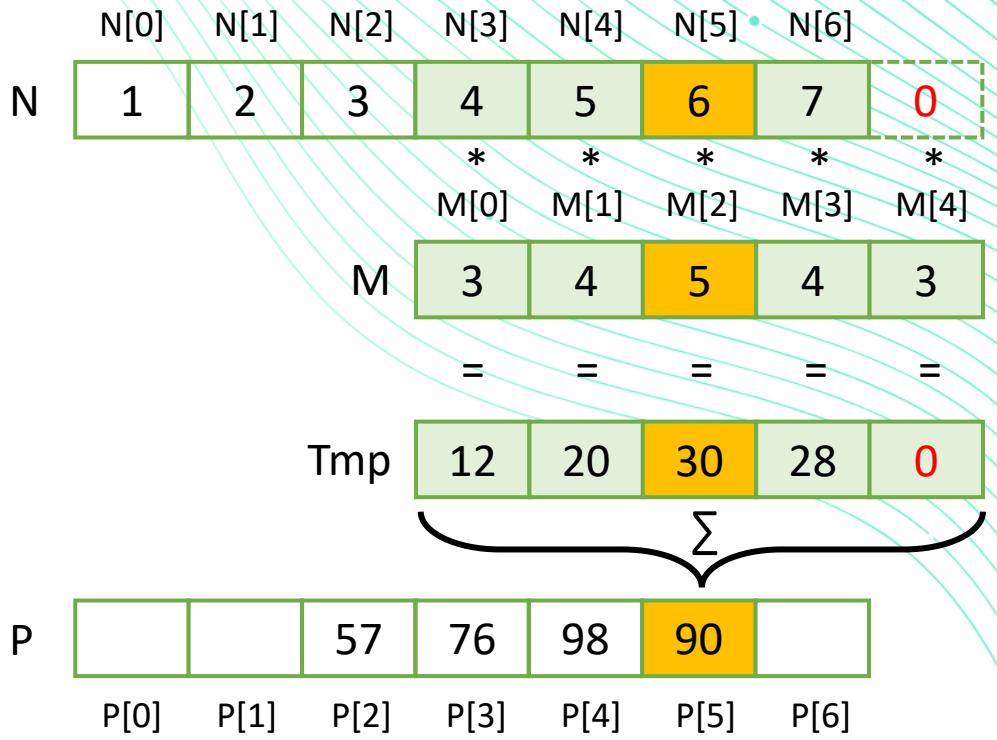


P[4] = N[2]*M[0] + N[3]*M[1] + N[4]*M[2] + N[5]*M[3] + N[6]*M[4]

Parallel Computation Patterns Stencil

Boundary condition

- calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
  - different policies (0, replicates of boundary values, etc.)

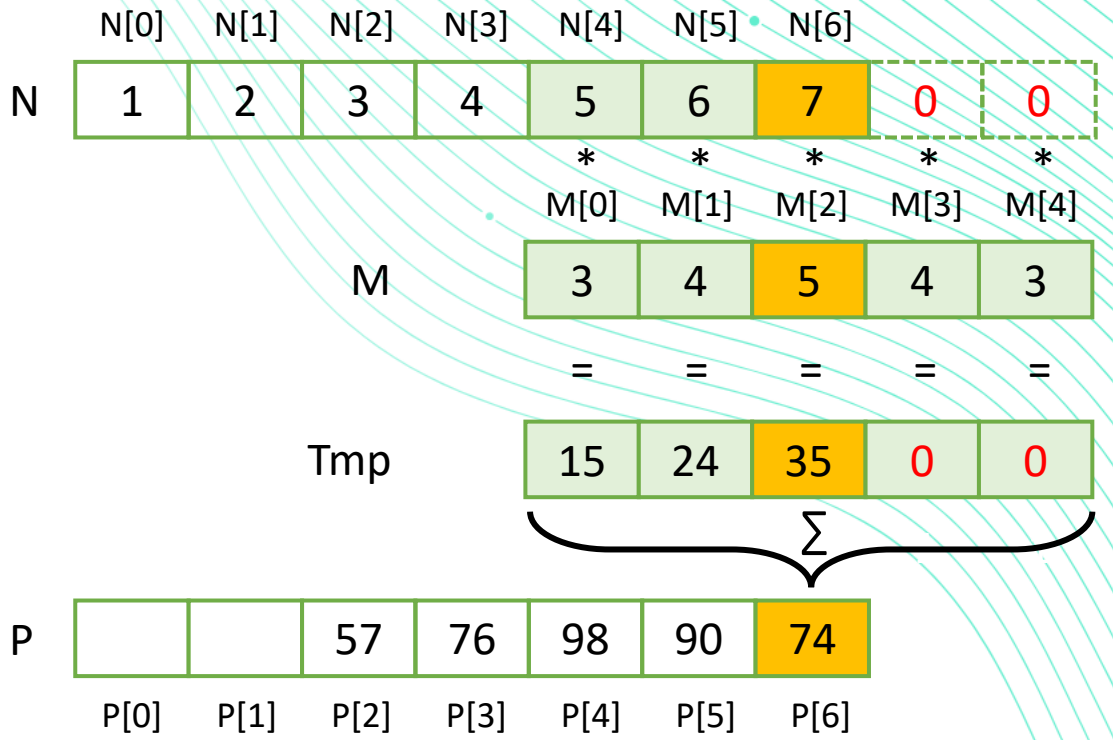


P[5] = N[3]*M[0] + N[4]*M[1] + N[5]*M[2] + N[6]*M[3] + **0***M[4]

Parallel Computation Patterns Stencil

Boundary condition

- calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements
  - different policies (0, replicates of boundary values, etc.)



P[3] = N[4]*M[0] + N[5]*M[1] + N[6]*M[2] + **0***M[3] + **0***M[4]

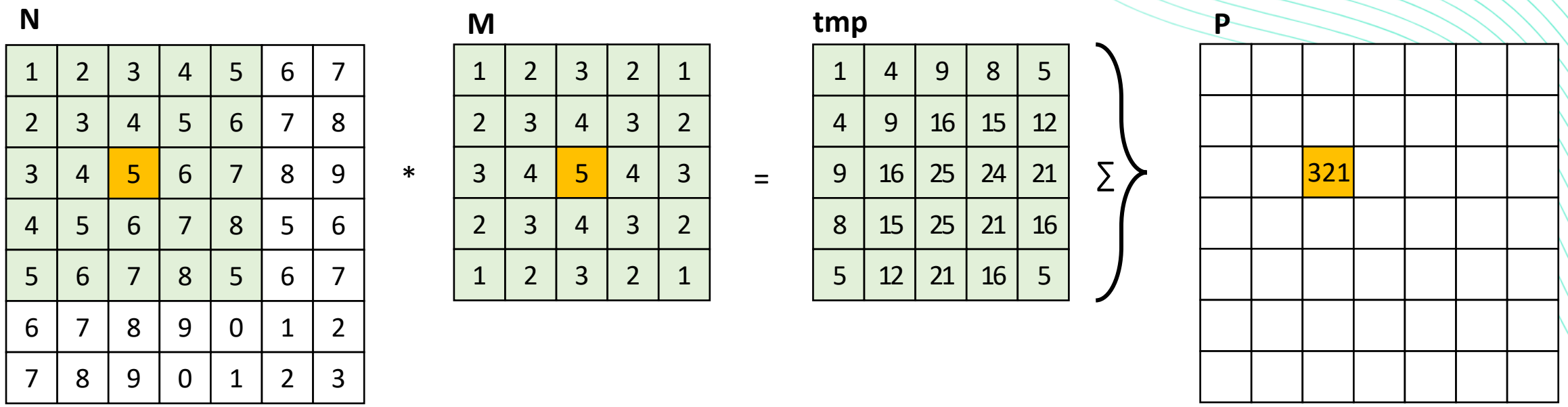
Parallel Computation Patterns
Basic Stencil kernel

```
__global__ void convolution_1D_basic_kernel(
    float *N, float *M, float *P,
    int Mask_Width, int Width)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int   N_start_point = i – (Mask_Width/2);

  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width)
    {
      Pvalue += N[N_start_point + j] * M[j];
    }
  }

  P[i] = Pvalue;
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

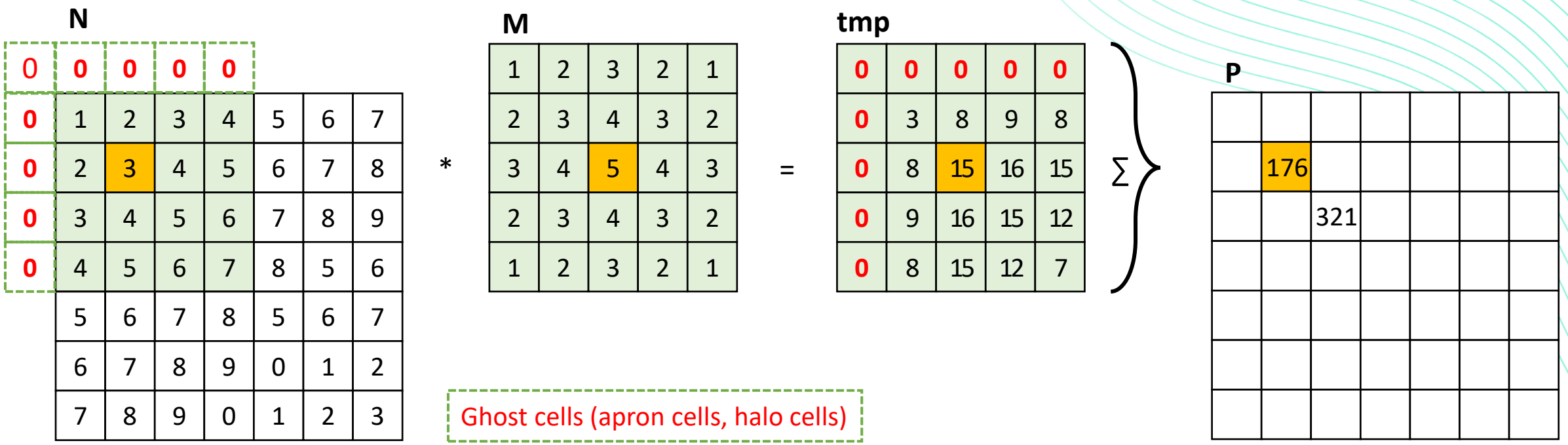
Parallel Computation Patterns
Stencil

2D Convolution



Parallel Computation Patterns
Stencil

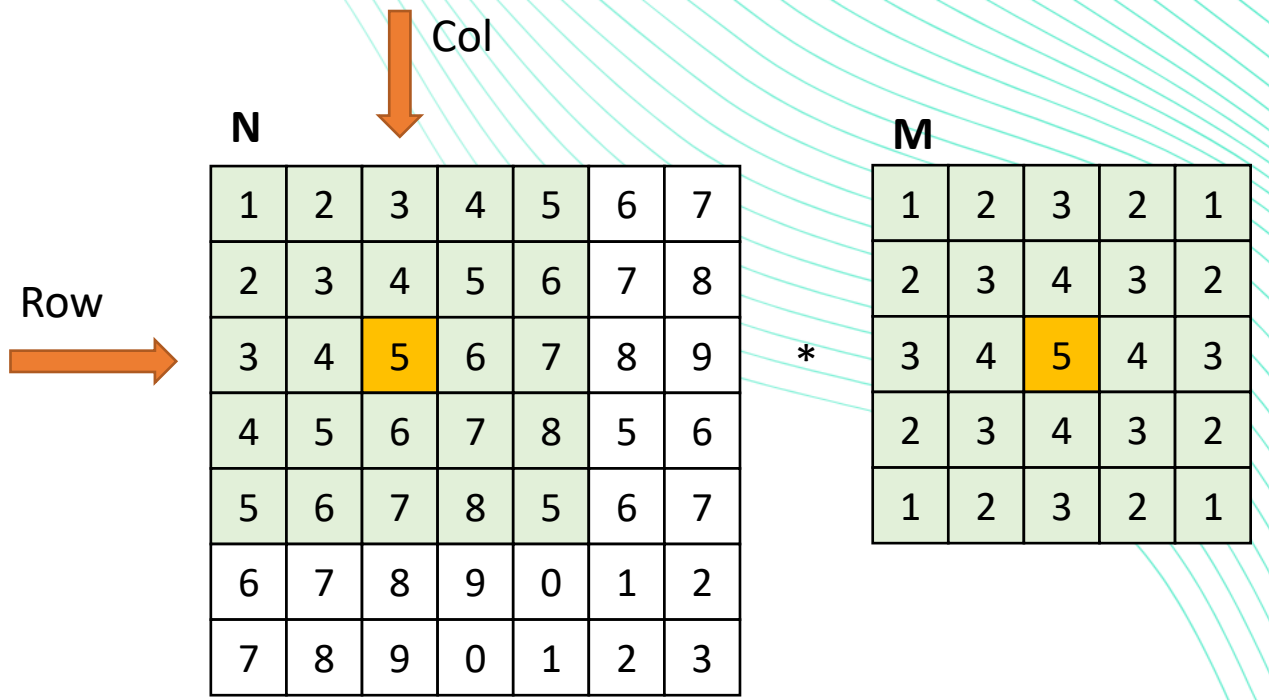
2D Convolution – boundaries with ghost cells



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

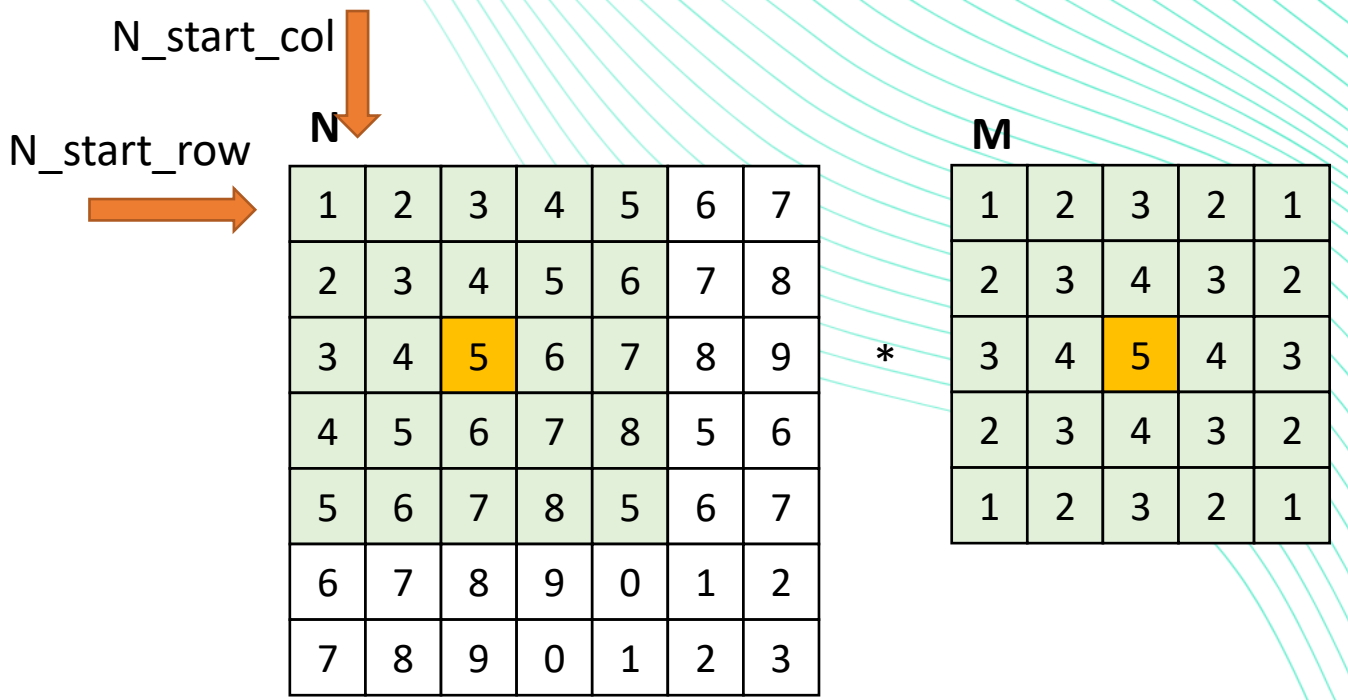
```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col = Col – (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskWidth; ++j) {
      for(int k = 0; k < maskWidth; ++k) {

        int curRow = N_start_row + j;
        int curCol = N_start_col + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];
        }
      }
    }

    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

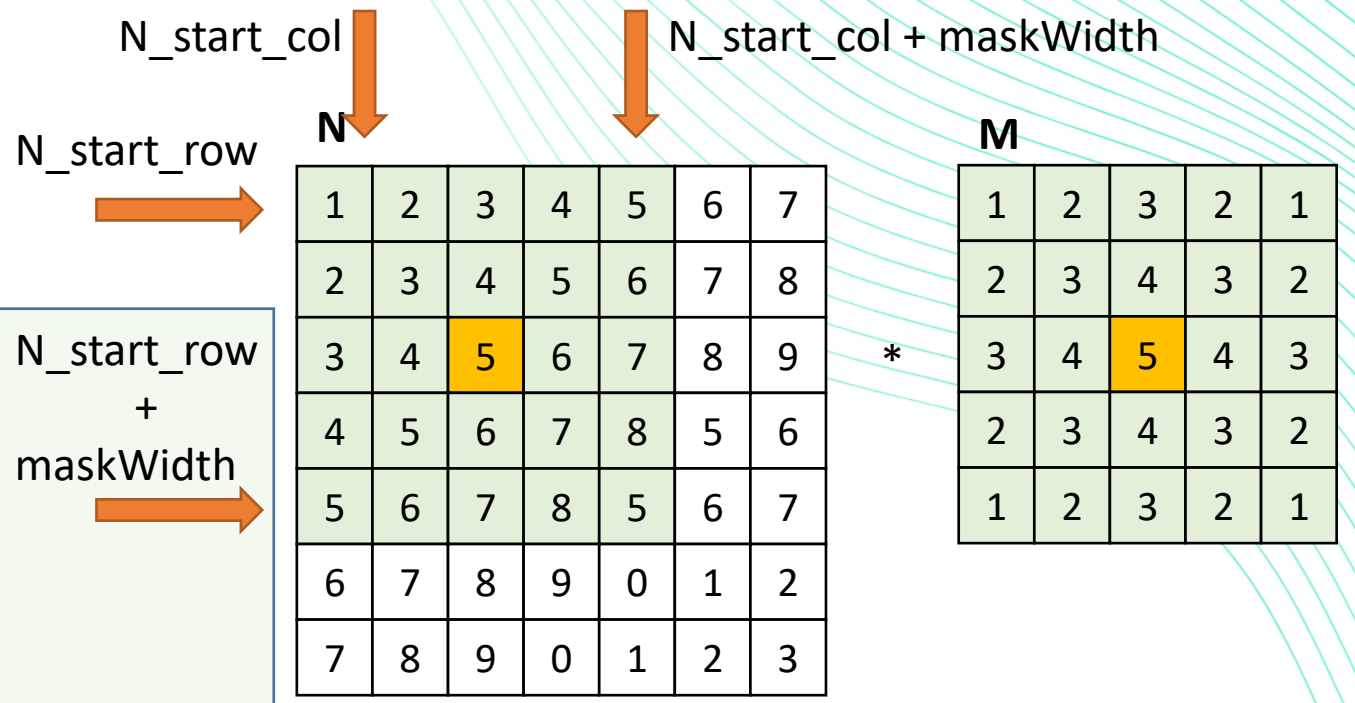
```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col = Col – (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskWidth; ++j) {
      for(int k = 0; k < maskWidth; ++k) {

        int curRow = N_start_row + j;
        int curCol = N_start_col + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];
        }
      }
    }

    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

```
__global__
void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask, unsigned char * out, int maskWidth, int w, int h) {
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  int Row = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
    int pixVal = 0;

    N_start_col = Col – (maskwidth/2);
    N_start_row = Row – (maskwidth/2);

    // Get the of the surrounding box
    for(int j = 0; j < maskWidth; ++j) {
      for(int k = 0; k < maskWidth; ++k) {

        int curRow = N_start_row + j;
        int curCol = N_start_col + k;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
          pixVal += in[curRow * w + curCol] * mask[j*maskWidth+k];
        }
      }
    }

    // Write our new pixel value out
    out[Row * w + Col] = (unsigned char)(pixVal);
  }
}
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Stencil

Using constant memory and caching for Mask

- mask is used by all threads but not modified in the convolution kernel
  - all threads in a warp access the same locations at each point in time

- CUDA devices provide constant memory whose contents are aggressively cached
  - cached values are broadcast to all threads in a warp
  - effectively magnifies memory bandwidth without consuming shared memory

- use of **`const __restrict__`** qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

Mask

1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

Mask

3	4	5	4	3

```
__global__ void convolution_2D_kernel(
        float *P,
        float *N,
        int height, int width,
        const float __restrict__ *M)
{ ... }
```

More info: https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

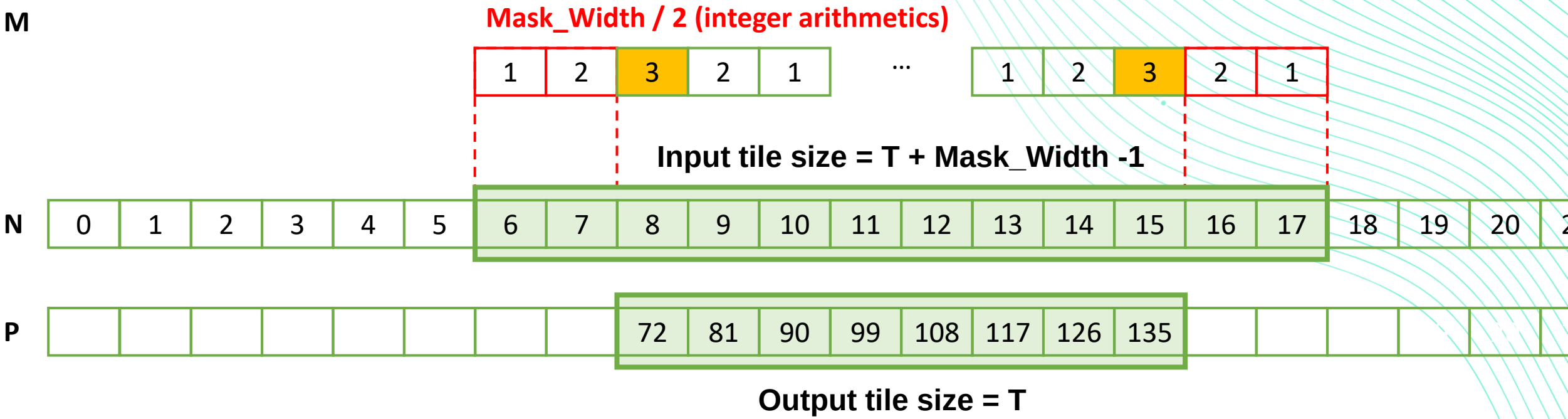
Parallel Computation Patterns Stencil

Tiling Opportunity Convolution

- calculation of adjacent output elements involve shared input elements
  - e.g., N[2] is used in calculation of P[0], P[1], P[2], P[3] and P[4] assuming a 1D convolution Mask_Width of width 5
- we can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses

Parallel Computation Patterns
Stencil

Tile considerations

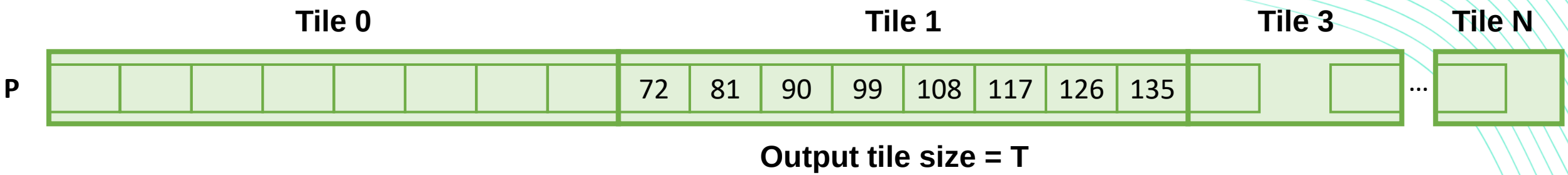


Assume that we want to have each block to calculate T output elements

- T + Mask_Width -1 input elements are needed to calculate T output elements
- T + Mask_Width -1 is usually not a multiple of T, except for small T values
- T is usually significantly larger than Mask_Width

Parallel Computation Patterns Stencil

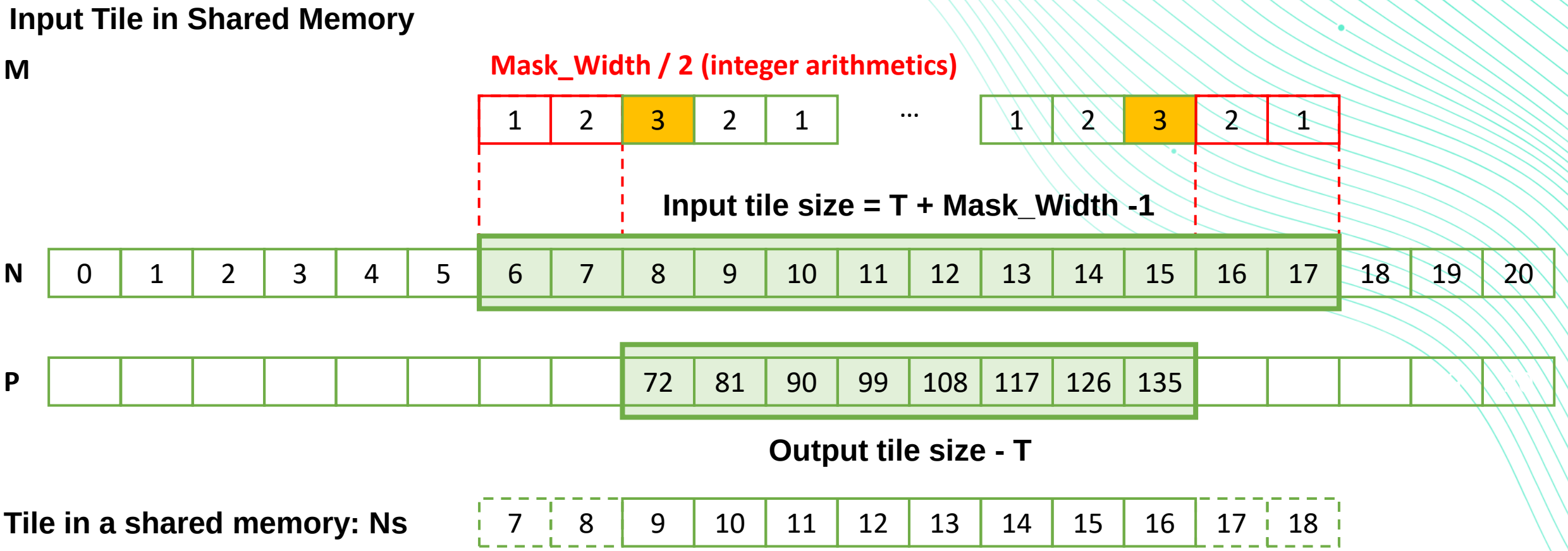
Output tile definition



- each thread block calculates one output tile
- each output tile width is **T**
  - **T** is 8 in this example

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Stencil



- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns
Stencil

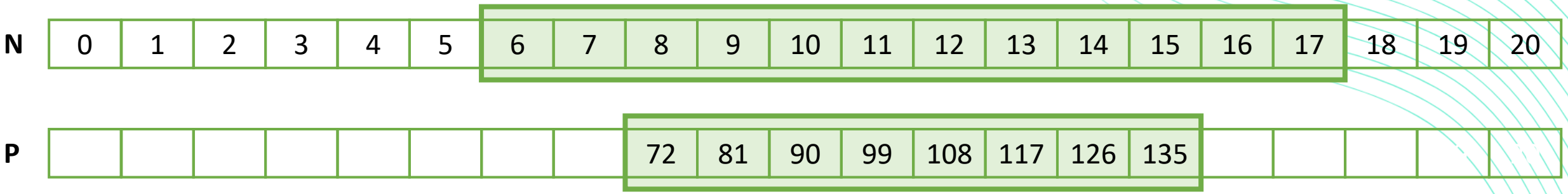
Design 1: The size of each thread block matches the size of an output tile

- All threads participate in calculating output elements
- blockDim.x would be 8 in our example
- Some threads need to load more than one input element into the shared memory

Design 2: The size of each thread block matches the size of an input tile

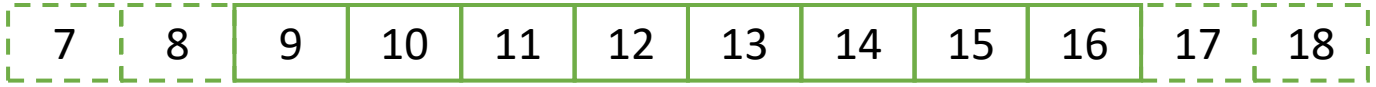
- Some threads will not participate in calculating output elements
- blockDim.x would be 12 in our example
- Each thread loads one input element into the shared memory

Input tile size = T + Mask_Width -1



Output tile size - T

Tile in a shared memory: Ns



- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns
Stencil

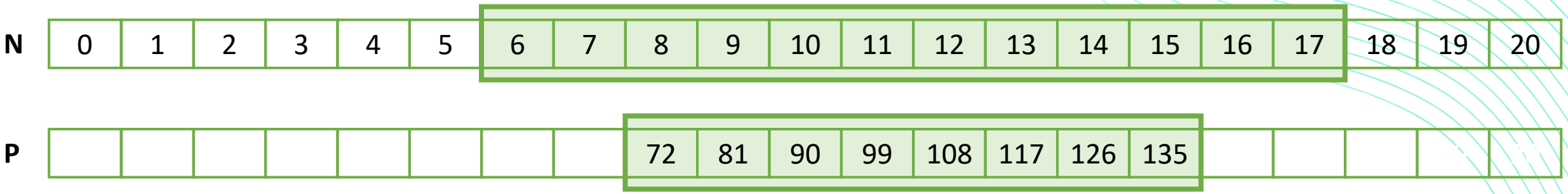
Design 1: The size of each thread block matches the size of an output tile

- All threads participate in calculating output elements
- blockDim.x would be 8 in our example
- Some threads need to load more than one input element into the shared memory

Design 2: The size of each thread block matches the size of an input tile

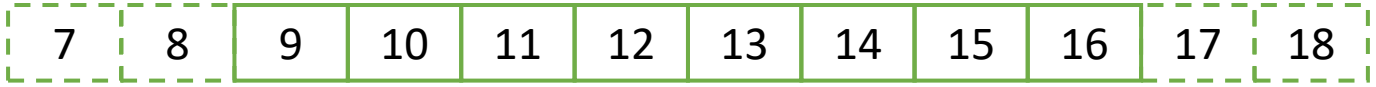
- Some threads will not participate in calculating output elements
- blockDim.x would be 12 in our example
- Each thread loads one input element into the shared memory

Input tile size = T + Mask_Width -1



Output tile size - T

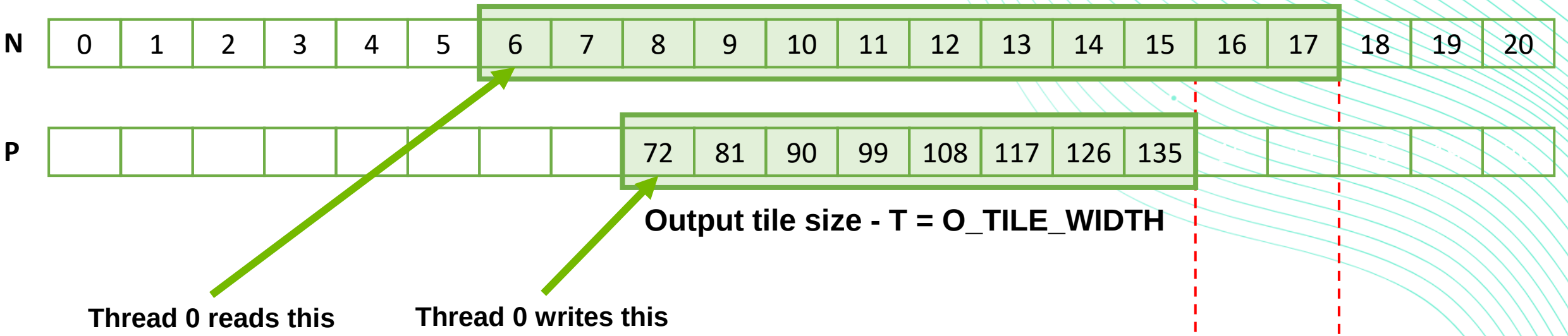
Tile in a shared memory: Ns



- each input tile has all values needed to calculate the corresponding output tile.

Parallel Computation Patterns
Stencil

Thread to Input and Output Data Mapping



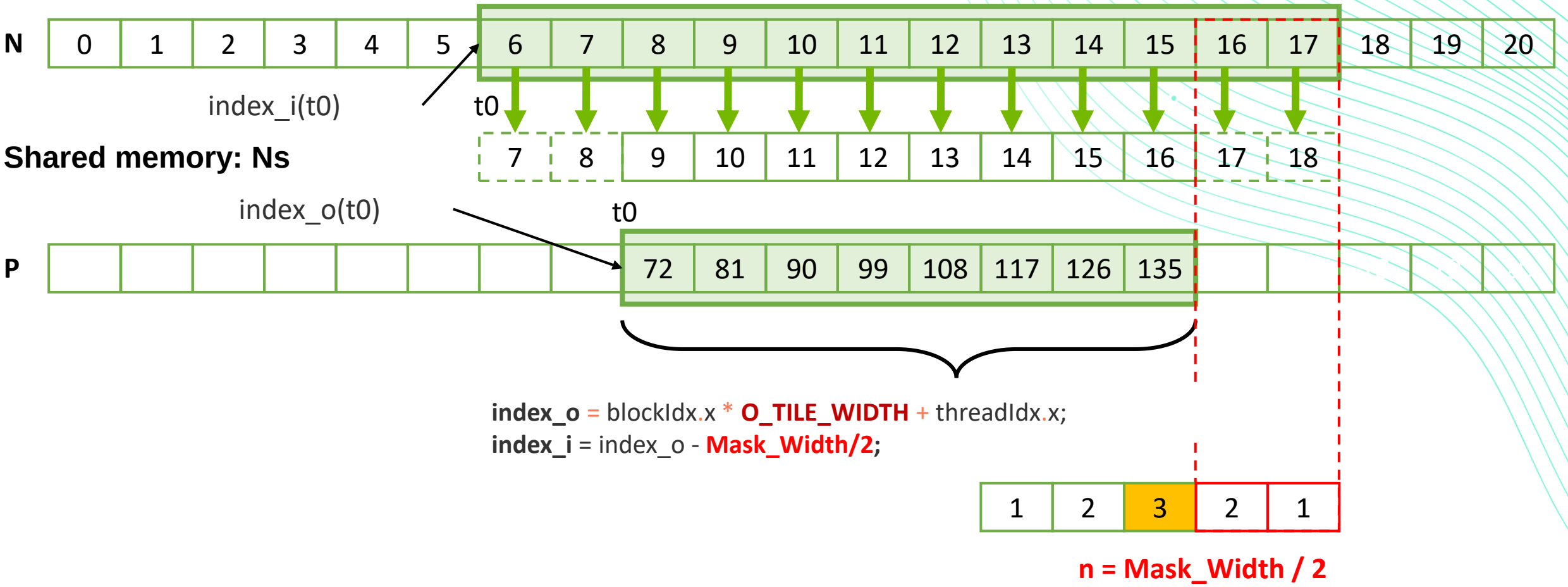
For each thread:

- **index_i = index_o – n**
- **where:**
  - **n is Mask_Width/2**
  - **n is 2 in this example**

n = Mask_Width / 2

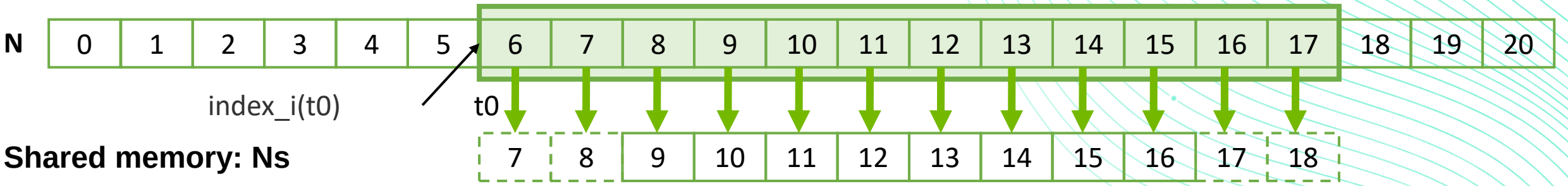
Parallel Computation Patterns
Stencil

Thread to Input and Output Data Mapping



Parallel Computation Patterns Stencil

Thread to Input and Output Data Mapping



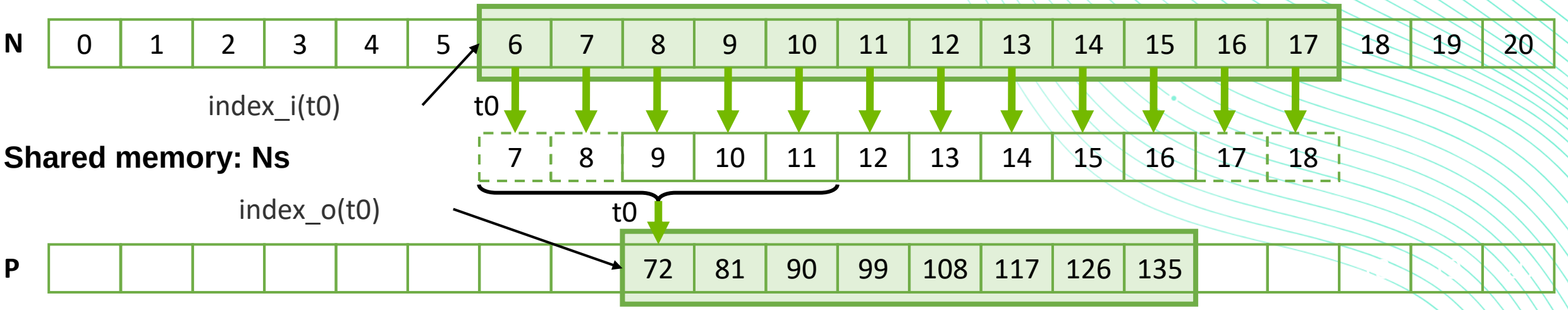
- all threads participate in loading input tiles

```
if((index_i >= 0) && (index_i < Width)) {
  Ns[threadIdx.x] = N[index_i];
}
else{
  Ns[threadIdx.x] = 0.0f;
}
__syncthreads()
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

Thread to Input and Output Data Mapping



- some threads do not participate in calculating output

```
float output = 0.0f;

if (threadIdx.x < O_TILE_WIDTH){
  output = 0.0f;
  for(j = 0; j < Mask_Width; j++) {
    output += M[j] * Ns[j+threadIdx.x];
  }
  P[index_o] = output;
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

Setting Block Size

```
#define O_TILE_WIDTH 1020
#define BLOCK_WIDTH (O_TILE_WIDTH +
                (Mask_Width-1))

dim3 dimBlock(BLOCK_WIDTH,1, 1);

dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
```

Kernel code (partial)

```
…
index_o = blockIdx.x * O_TILE_WIDTH +
        threadIdx.x;
index_i = index_o – n - Mask_Width/2;

if((index_i >= 0) && (index_i < Width)) {
  Ns[threadIdx.x] = N[index_i];
}
else{
  Ns[threadIdx.x] = 0.0f;
}
__syncthreads()
if (threadIdx.x < O_TILE_WIDTH){
  float output = 0.0f;
  for(j = 0; j < Mask_Width; j++) {
    output += M[j] * Ns[j+threadIdx.x];
  }
  P[index_o] = output;
} …
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Stencil

The Efficiency of Tiling

- Significant reduction of Global Memory bandwidth

1D Convolution

- **The reduction ratio – how many times tiling reduces accesses to Global Memory**
- MASK_WIDTH * (O_TILE_WIDTH)/(O_TILE_WIDTH+MASK_WIDTH-1)

O_TILE_WIDTH	16	32	64	128	256
MASK_WIDTH= 5	4.0	4.4	4.7	4.9	4.9
MASK_WIDTH = 9	6.0	7.2	8.0	8.5	8.7

2D Convolution

- The reduction ratio is:
  - $O\_TILE\_WIDTH^2 * MASK\_WIDTH^2 / (O\_TILE\_WIDTH+MASK\_WIDTH-1)^2$

O_TILE_WIDTH	8	16	32	64
MASK_WIDTH = 5	11.1	16	19.7	22.1
MASK_WIDTH = 9	20.3	36	51.8	64

Tile size has significant effect on of the memory bandwidth reduction ratio.

This often argues for larger shared memory size.

EPICURE

Unlocking European-level HPC Support

Hands-On: 1D Convolution

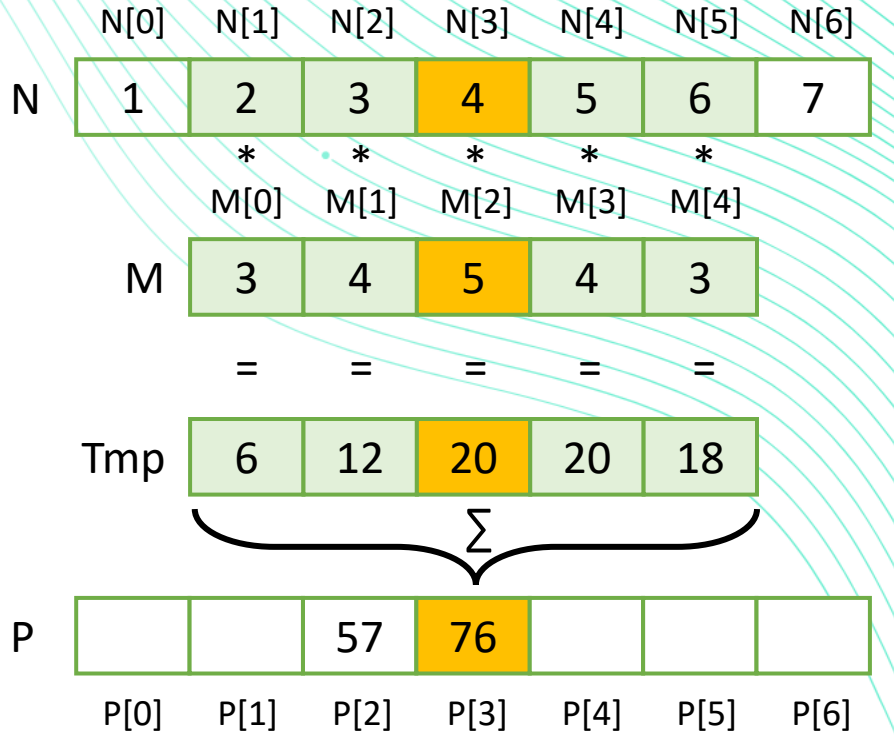
Hands-On
1D Convolution

- `tasks/convolution_1d`
- Finish the TODO tasks
  - Finish the naïve 1D convolution kernel
  - Finish the 1D convolution kernel that uses shared memory and tiling
- Compare the execution times
  - Why do you think the difference is so small?

- Recommend **`ssize_t`** type for indexing

Expected output:

```
Naive implementation
  Everything seems OK
  Kernel time: 87.584770 ms
Shared memory implementation
  Everything seems OK
  Kernel time: 84.019203 ms
```



EPICURE

Unlocking European-level HPC Support

Parallel Computation Patterns: Reduction

Parallel Computation Patterns Reduction

Parallel Reduction

- a commonly used strategy for processing large input data sets
- there is no required order of processing elements in a data set (associative and commutative)

Approach:

- partition the data set into smaller chunks
- have each thread to process a chunk
- use a reduction tree to summarize the results from each chunk into the final answer
- **we will focus on the reduction tree step now**

Reduction also enables other techniques

- reduction is also needed to clean up after some commonly used parallelizing transformations
- Example: privatization
  - multiple threads write into an output location
  - replicate the output location so that each thread has a private output location (privatization)
  - use a reduction tree to combine the values of private locations into the original output location

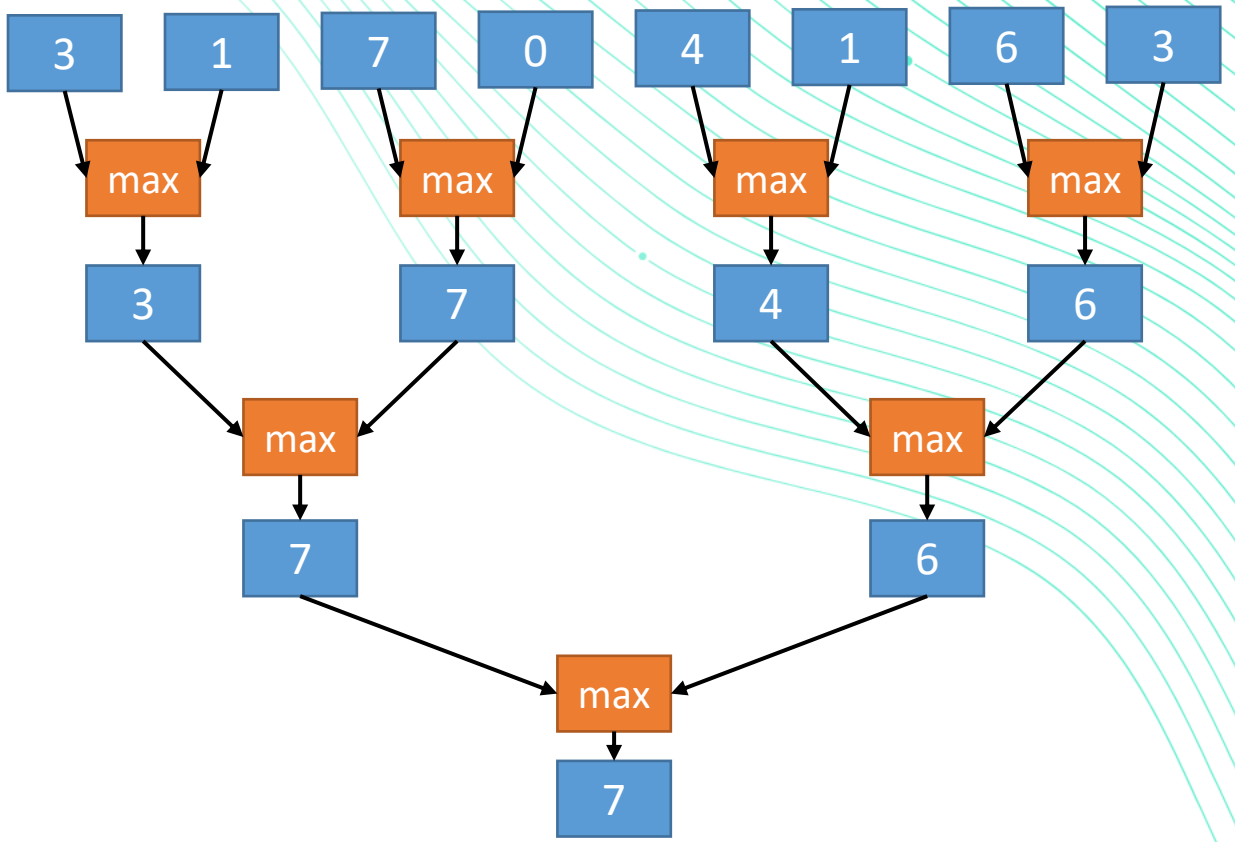
Parallel Computation Patterns
Reduction

Parallel Reduction

- summarize a set of input values into one value using a "reduction operation"
  - Max, Min, Sum, Product, …
- can be used with a user defined reduction operation function if the operation:
  - is associative and commutative
  - has a well-defined identity value (e.g., 0 for sum)

An Efficient Sequential Reduction O(N)

- initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- iterate through the input and perform the reduction operation between the result value and the current input value
- **N-1 reduction operations performed for N input values**
- each input value is only visited once – an O(N) algorithm



A parallel reduction tree algorithm performs N-1 operations in log(N) steps

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Reduction

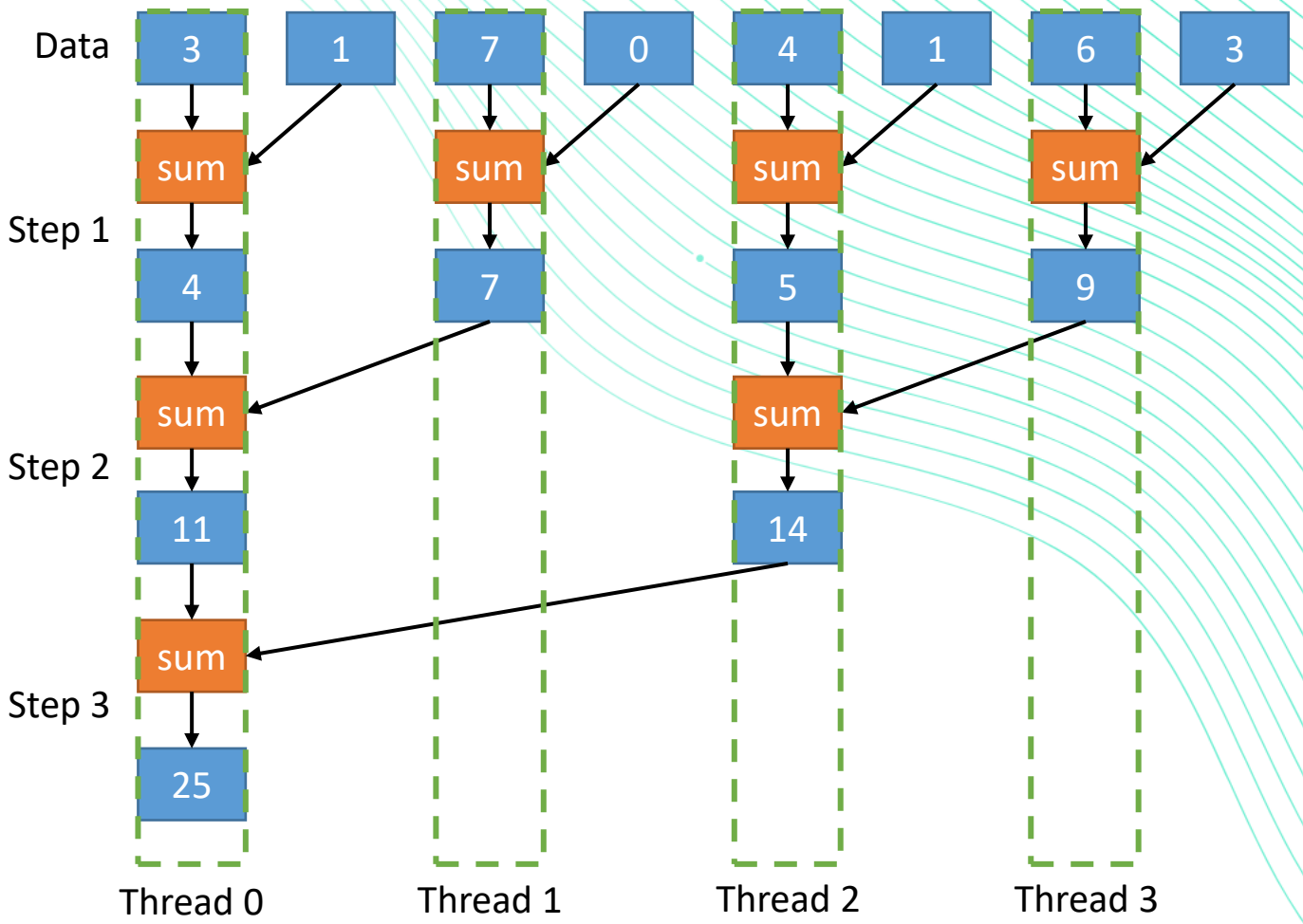
Parallel Sum Reduction on GPU

Parallel implementation

- each thread adds two values in each step,
- recursively halve # of threads,
- takes **log(n)** steps for n elements,
- requires **n/2** threads

Assume an in-place reduction using shared memory

- the original vector is in device global memory
- the shared memory is used to hold a partial sum vector
  - initially, the partial sum vector is simply the original vector
- each step brings the partial sum vector closer to the sum
- the final sum will be in element 0 of the partial sum vector
- reduces global memory traffic due to reading and writing partial sum values
- thread block size limits n to be less than or equal to 2,048



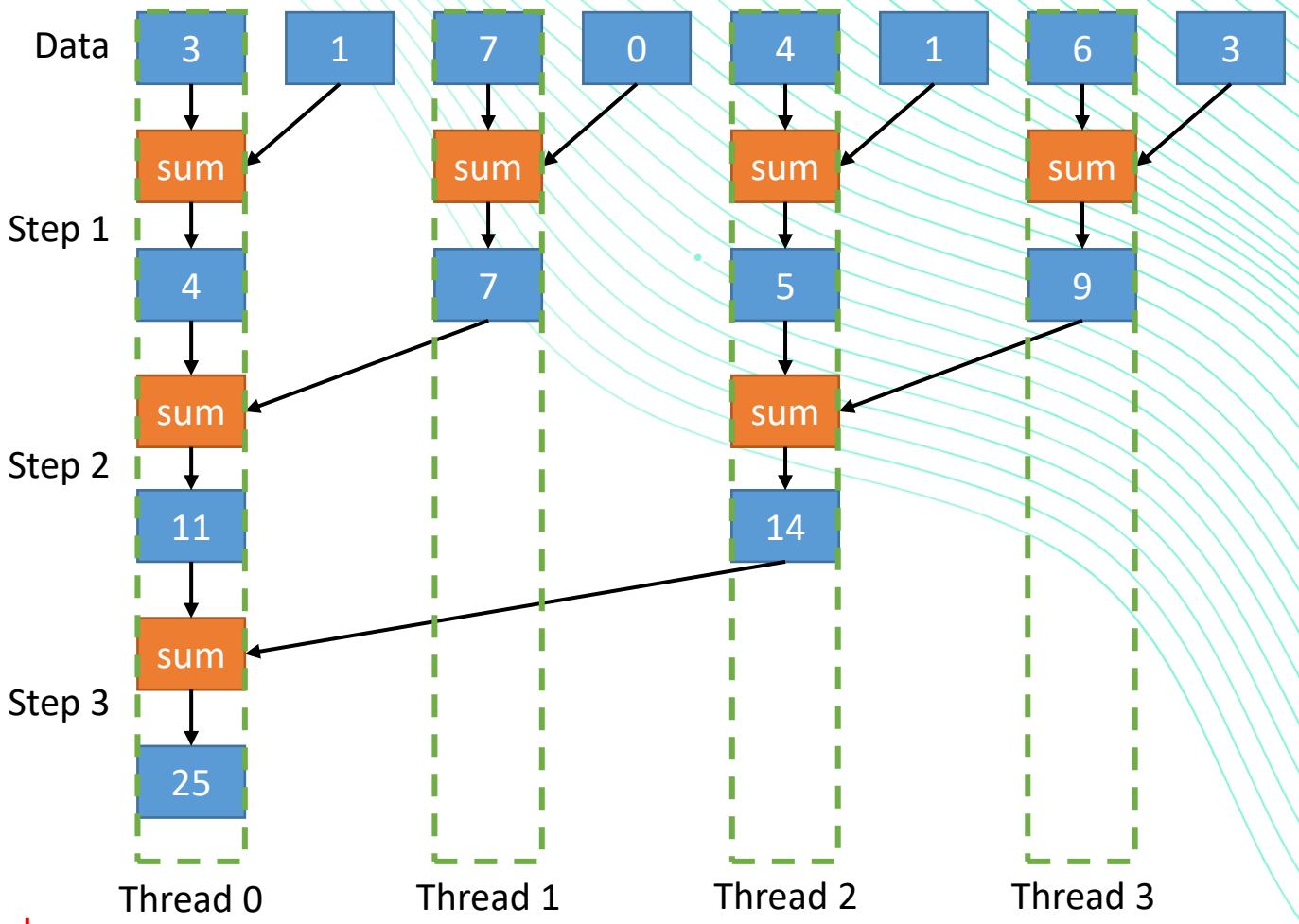
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Reduction

A Simple Thread Block Design

- each thread block takes 2*BlockDim.x input elements
- each thread loads 2 elements into shared memory

```
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t     = threadIdx.x;
unsigned int start = 2 * blockIdx.x * blockDim.x;

partialSum[t]         = input[start + t];
partialSum[blockDim+t] = input[start +
                        blockDim.x + t];

// The reduction step
for (unsigned int stride = 1;
             stride <= blockDim.x;
             stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```



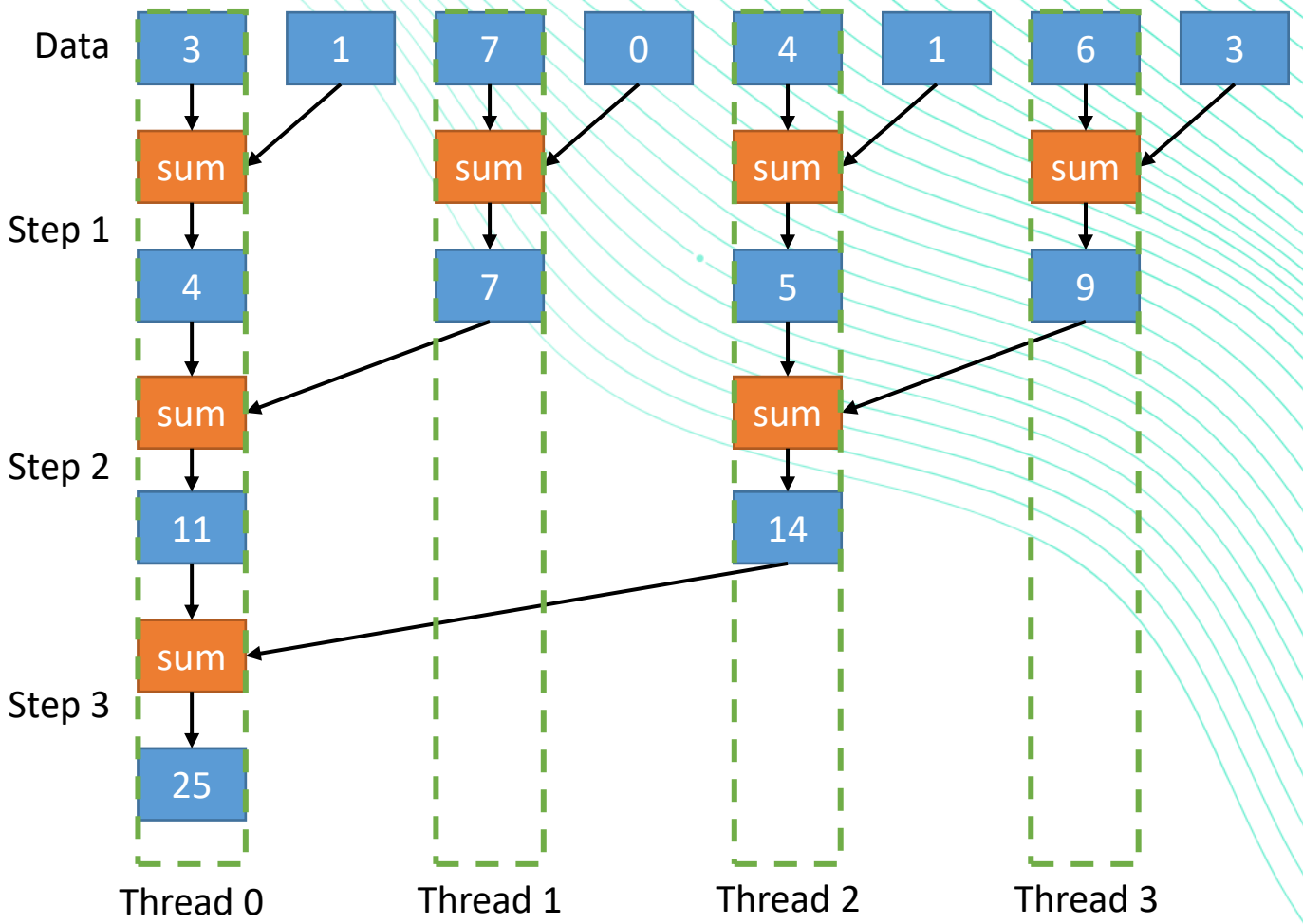
__syncthreads() is needed to ensure that all elements of each step of partial sums have been generated before the next step

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Reduction

Global Picture

- at the end of the kernel, Thread 0 in each block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x
- there can be a large number of such sums if the original vector is very large
- the host code may iterate and launch another kernel
- if there are only a small number of sums, the host can simply transfer the data back and add them together
- alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

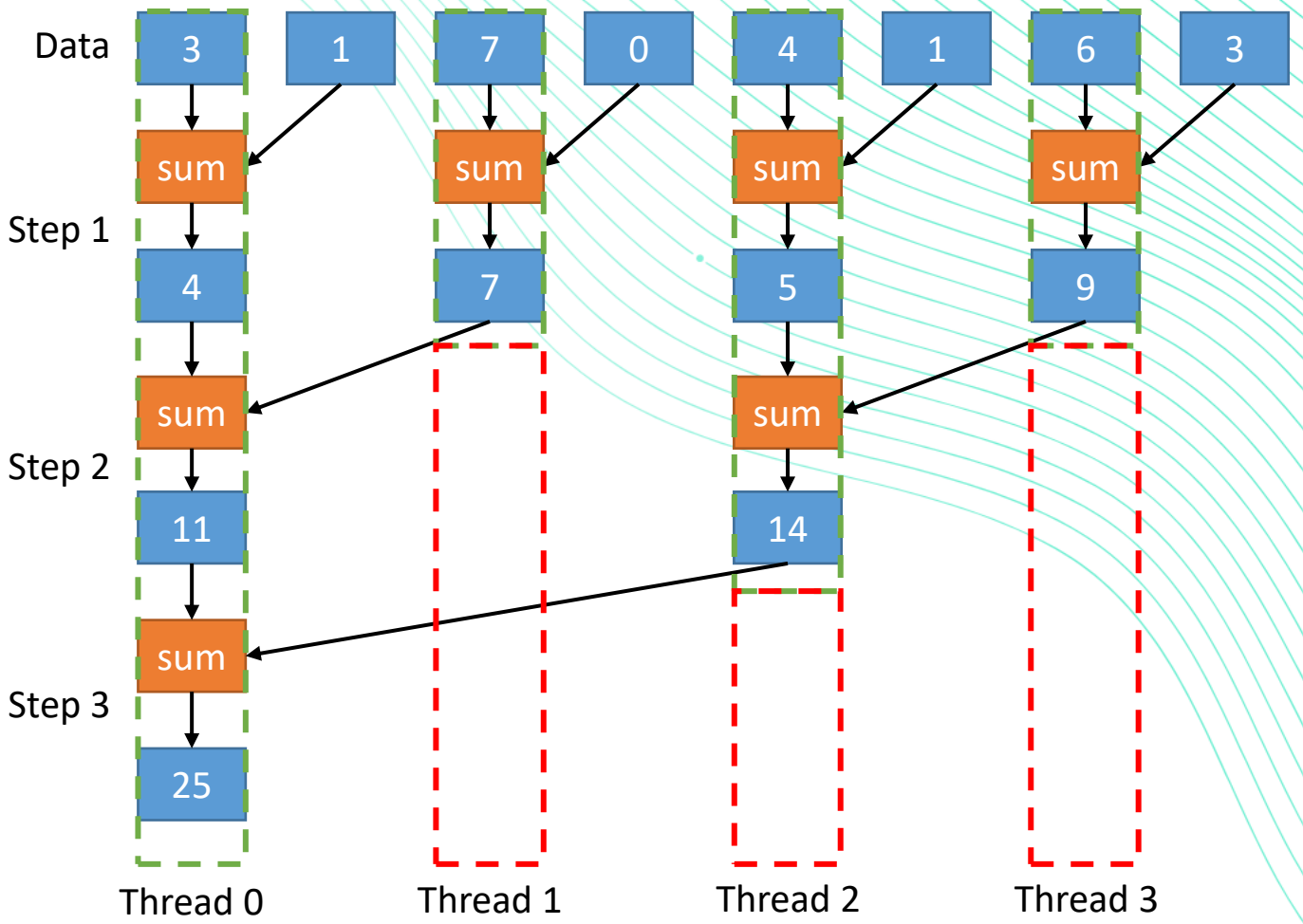
Parallel Computation Patterns
Reduction

Naive Thread to Data Mapping

- each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- after each step, half of the threads are no longer needed
- one of the inputs is always from the location of responsibility
- in each step, one of the inputs comes from an increasing distance away

Control Divergence of Naïve Kernel

- in each iteration, two control flow paths will be sequentially traversed for each warp
  - threads that perform addition and threads that do not
- threads that do not perform addition still consume execution resources
- half or fewer of threads will be executing after the first step
- all odd-index threads are disabled after first step
- after the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
- this can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire



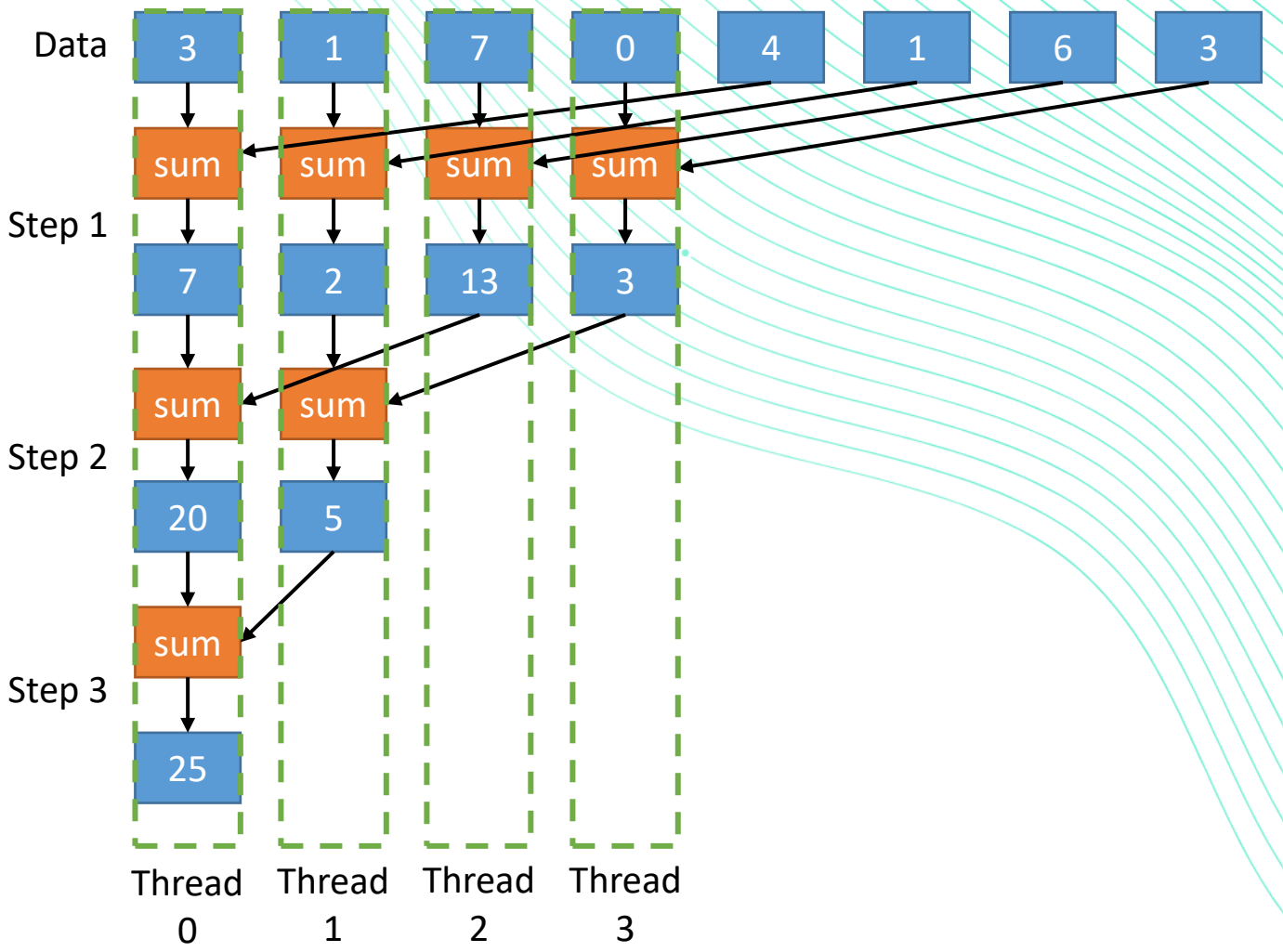
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Reduction

Better Thread to Data Mapping

- in some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators
- always compact the partial sums into the front locations in the partialSum[ ] array
- keep the active threads consecutive

```
for (unsigned int stride = blockDim.x;
                 stride > 0;
                 stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
  }
```



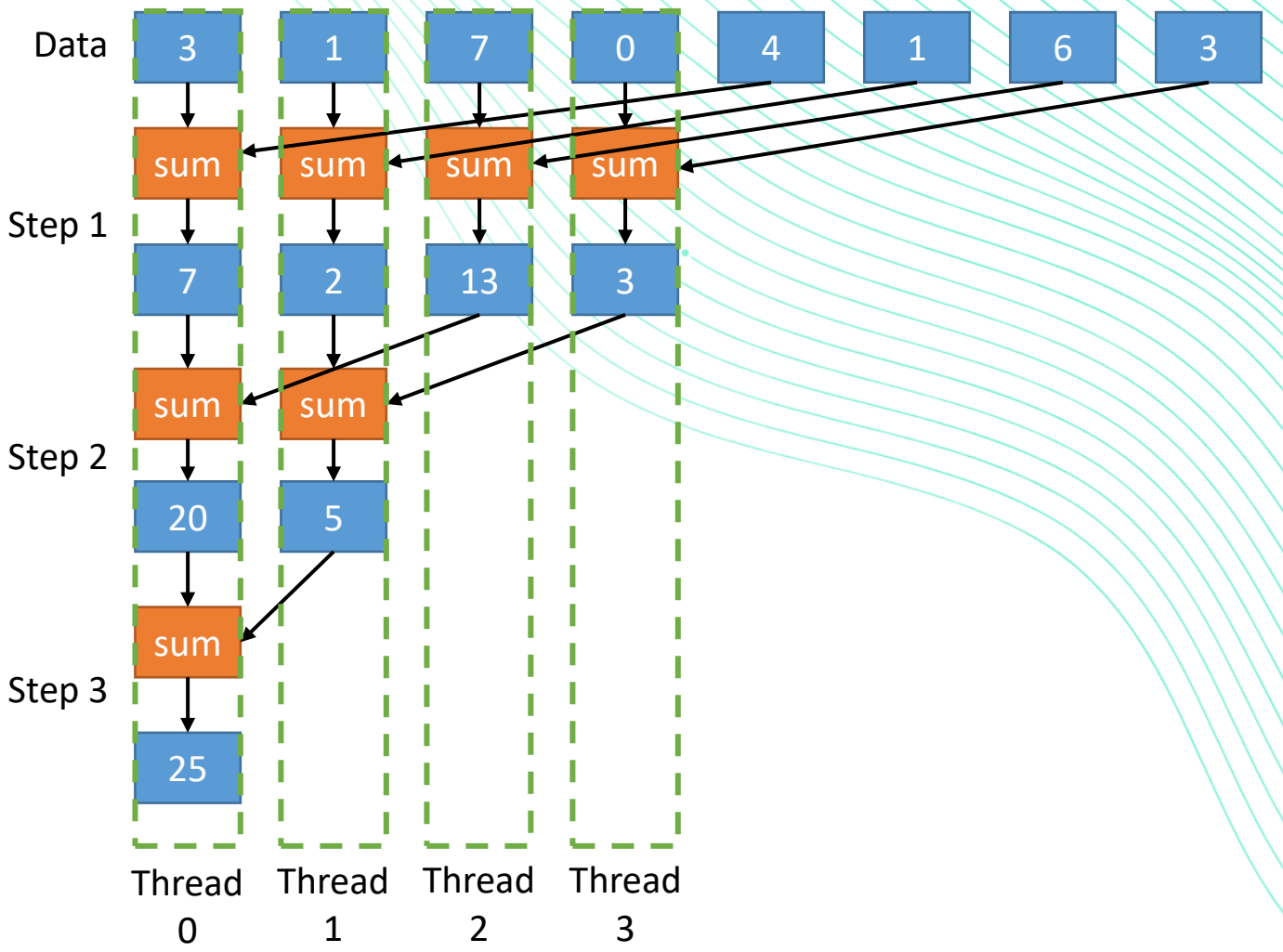
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Reduction

A Quick Analysis for a 1024 thread block

- **no divergence in the first 5 steps**
  - 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  - All threads in each warp either all active or all inactive
- **the final 5 steps will still have divergence**

```
for (unsigned int stride = blockDim.x;
          stride > 0;
          stride /= 2)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
  }
```



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

Unlocking European-level HPC Support

Hands-On Reduction

Hands-On Reduction

- `tasks/reduction`
- Complete the TODO1, the rest is a bonus task for you now
- Write the implementation of the reduction sum kernel
  - Inside a block, use the described parallel reduction
  - Add the block result to the total result using atomicAdd
    - atomicAdd(destination_pointer, value)
- Launch the kernel in main()
- Compile with additional flag `-arch=native` (or `-arch=sm_80` for A100)

Expected output:

```
Shared memory sum reduction
  Correct result is  10432810085616533504.0
  Computed result is 10432810086381977600.0
  Relative error is 7.337e-11
  The results are close enough
  Kernel time: 36.642815 ms
```

EPICURE

Unlocking European-level HPC Support

Parallel Computation Patterns: Histogram (Atomic Operations)

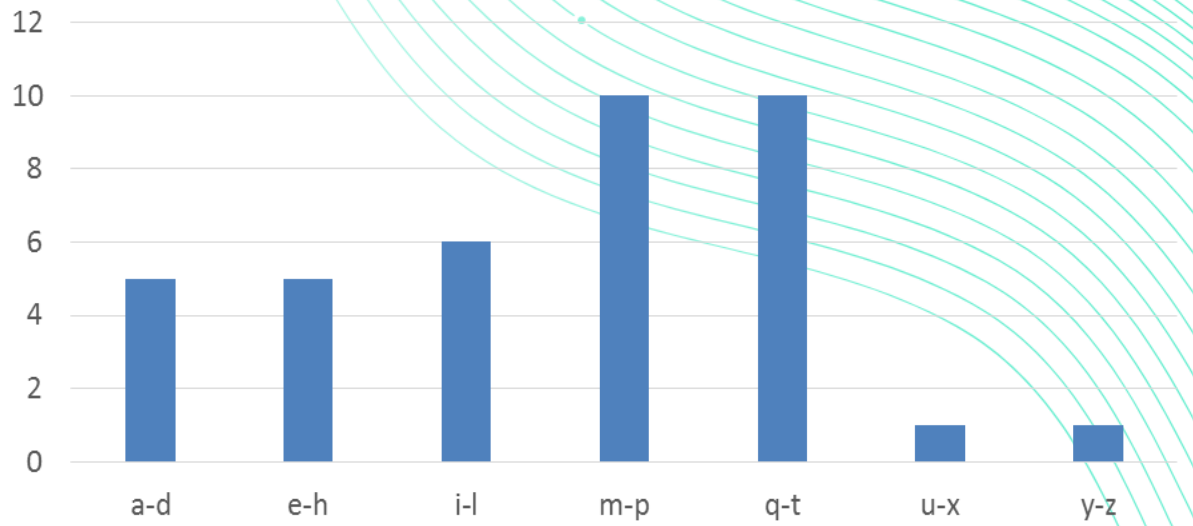
Parallel Computation Patterns Histogram

Histogram

- A method for extracting notable features and patterns from large data sets
- Basic histograms - for each element in the data set, use the value to identify a “bin counter” to increment

A Text Histogram Example

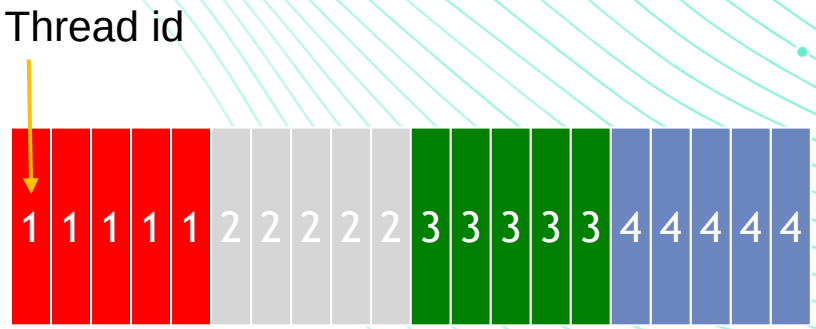
- define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, …
- for each character in an input string, increment the appropriate bin counter.
- in the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



Parallel Computation Patterns Histogram

A simple parallel histogram algorithm

- partition the input into sections
- have each thread to take a section of the input
- each thread iterates through its section.
- for each letter, increment the appropriate bin counter



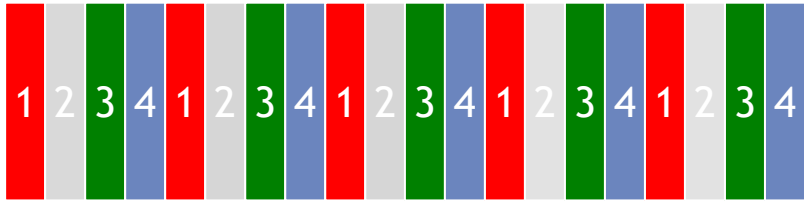
Input Partitioning Affects Memory Access Efficiency

Sectioned partitioning

- results in poor memory access efficiency
- adjacent threads do not access adjacent memory locations
- accesses are not coalesced
- DRAM bandwidth is poorly utilized

Interleaved partitioning

- all threads process a contiguous section of elements
- they all move to the next section and repeat
- the memory accesses are coalesced

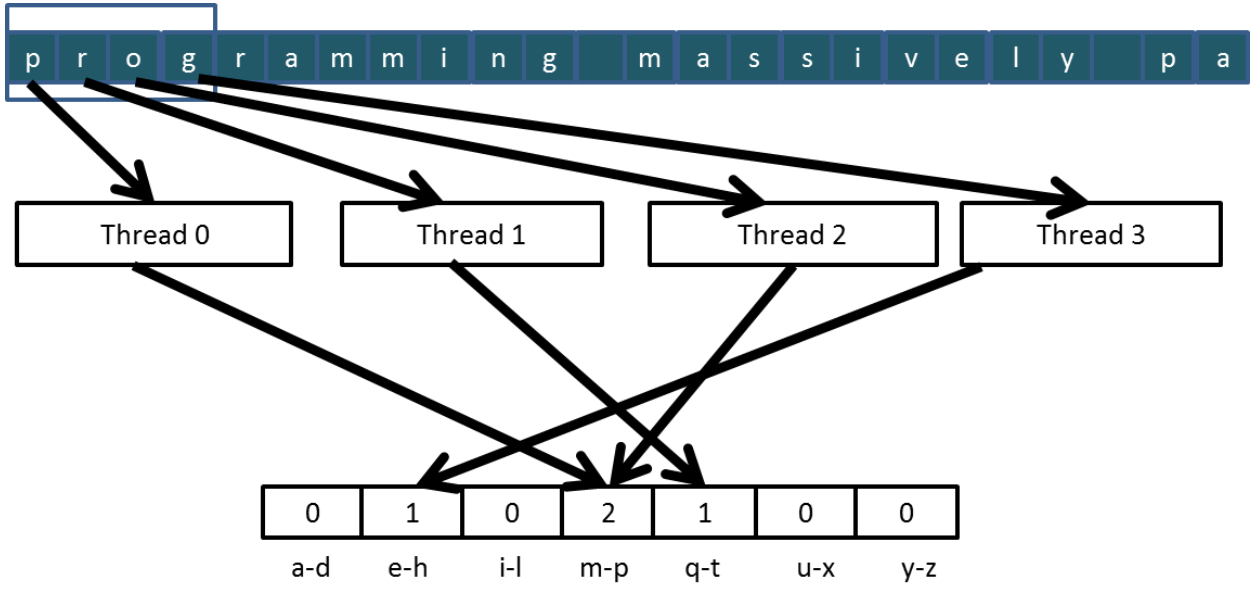


Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

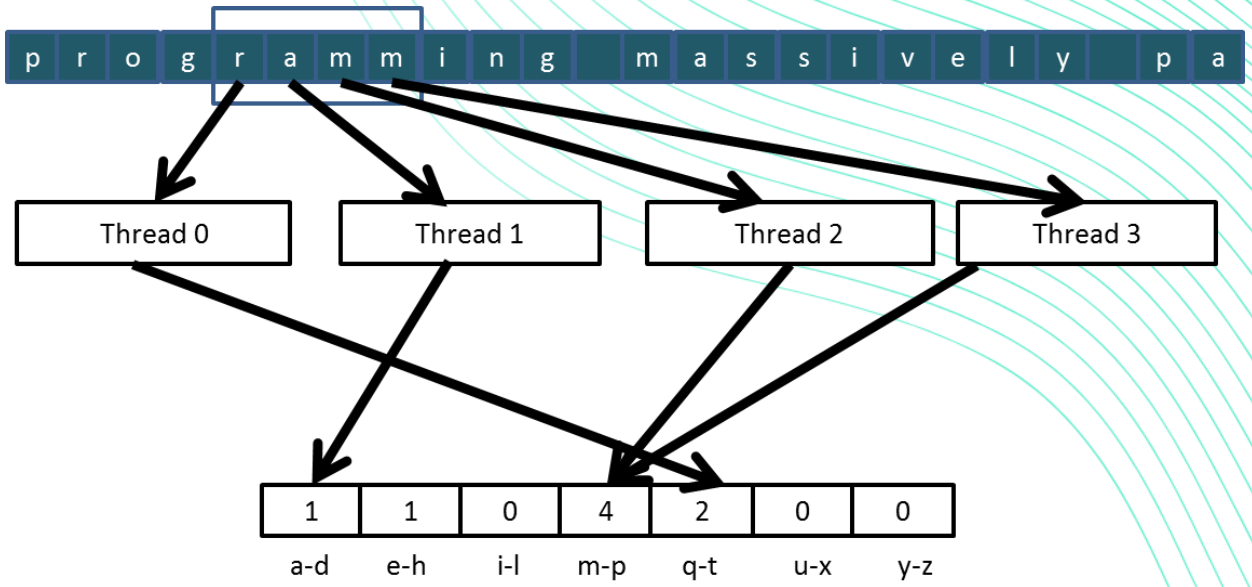
Parallel Computation Patterns Histogram

Interleaved partitioning of input

Iteration 1



Iteration 2

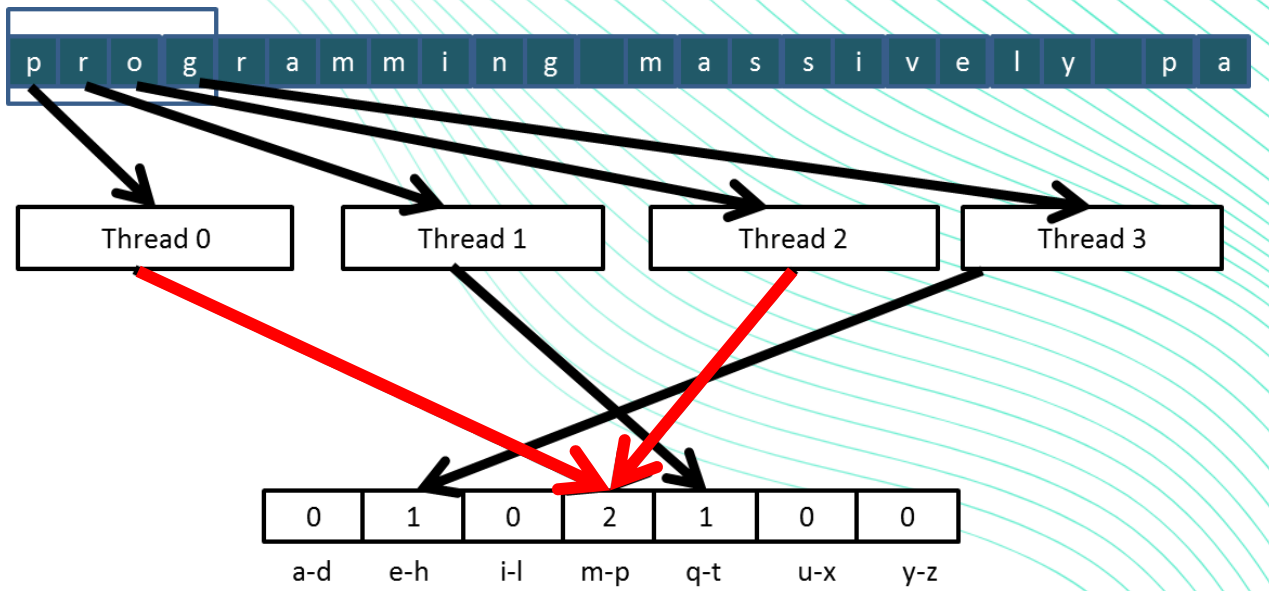


Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Histogram

Interleaved partitioning of input

- for every input element thread increments selected bin
- bin incrementation results in
  - **Read-modify-write** operation
  - **can result in Data Race**



Data Race in Parallel Thread Execution

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

- **Old** and **New** are per-thread register variables.

Question 1: If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?

Question 2: What does each thread get in their Old variable?

Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a **data race.**

Parallel Computation Patterns Histogram

Data race examples

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

Timing Scenario #1

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

Timing Scenario #2

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Parallel Computation Patterns Histogram

Data race examples

Timing Scenario #3

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Timing Scenario #4

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Parallel Computation Patterns Histogram

Atomic Operations Ensure Good Outcomes

```
thread1:  Old ← Mem[x]
          New ← Old + 1
          Mem[x] ← New

                          thread2:  Old ← Mem[x]
                                    New ← Old + 1
                                    Mem[x] ← New
```

Or

```
                          thread2:  Old ← Mem[x]
                                    New ← Old + 1
                                    Mem[x] ← New

thread1:  Old ← Mem[x]
          New ← Old + 1
          Mem[x] ← New
```

Timing Scenario #3

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Timing Scenario #4

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Parallel Computation Patterns Histogram

Atomic Operations

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Key Concepts of Atomic Operations

- a read-modify-write operation performed by a single hardware instruction on a memory location address
  - read the old value, calculate a new value, and write the new value to the location
- the hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
  - any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - all threads perform their atomic operations serially on the same location

Parallel Computation Patterns Histogram

Atomic Operations

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Atomic Arithmetic Operations in CUDA

- performed by calling functions that are translated into single instructions (a.k.a. intrinsic functions or intrinsics)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide for details

Example: Atomic Add

```
int atomicAdd(int* address, int val);
```

- reads the 32-bit word old from the location pointed to by address in global or shared memory, computes (old + val), and stores the result back to memory at the same address.
- these three operations are performed in one atomic transaction. The function returns old.

More Atomic Adds in CUDA

- unsigned 32-bit integer atomic add - `unsigned int atomicAdd`
- unsigned 64-bit integer atomic add, single-precision floating-point atomic add, double-precision floating-point atomic add, 16-bit floating-point atomic add, …

Parallel Computation Patterns Histogram

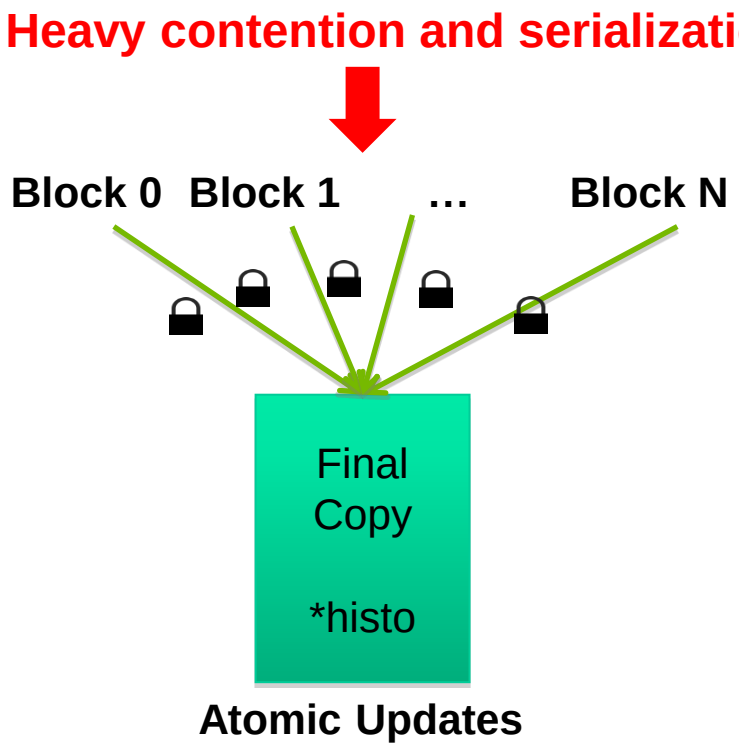
A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread process the input in a strided pattern

```
__global__ void histo_kernel(
            unsigned char *buffer,
            long size,
            unsigned int *histo)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  // stride is total number of threads
  int stride = blockDim.x * gridDim.x;

  // All threads handle blockDim.x * gridDim.x
  // consecutive elements
  while (i < size) {
    int alphabet_position = buffer[i] – “a”;
    if (alphabet_position >= 0 && alpha_position < 26)
    {
      atomicAdd(&(histo[alphabet_position/4]), 1);
    }
    i += stride;
  }
}
```

Parallel Computation Patterns Histogram

A Basic Text Histogram Kernel



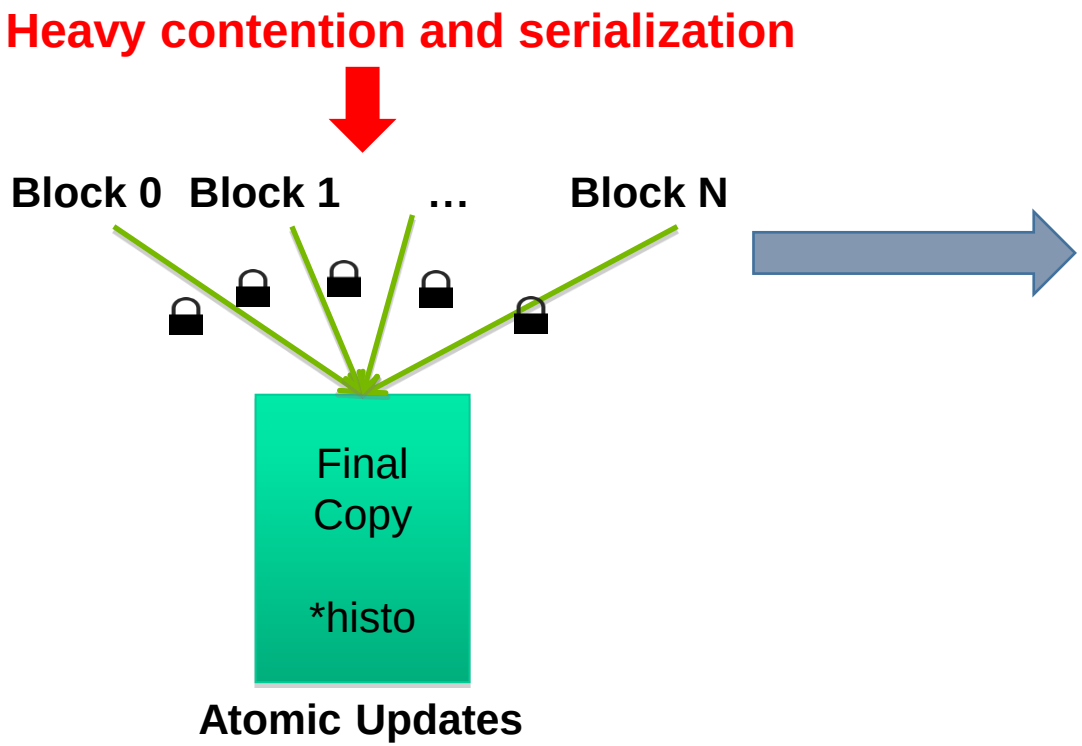

```
__global__ void histo_kernel(
            unsigned char *buffer,
            long size,
            unsigned int *histo)
{
 int i = threadIdx.x + blockIdx.x * blockDim.x;

 // stride is total number of threads
 int stride = blockDim.x * gridDim.x;

 // All threads handle blockDim.x * gridDim.x
 // consecutive elements
 while (i < size) {
  int alphabet_position = buffer[i] – “a”;
  if (alphabet_position >= 0 && alpha_position < 26)
  {
   atomicAdd(&(histo[alphabet_position/4]), 1);
  }
  i += stride;
 }
}
```

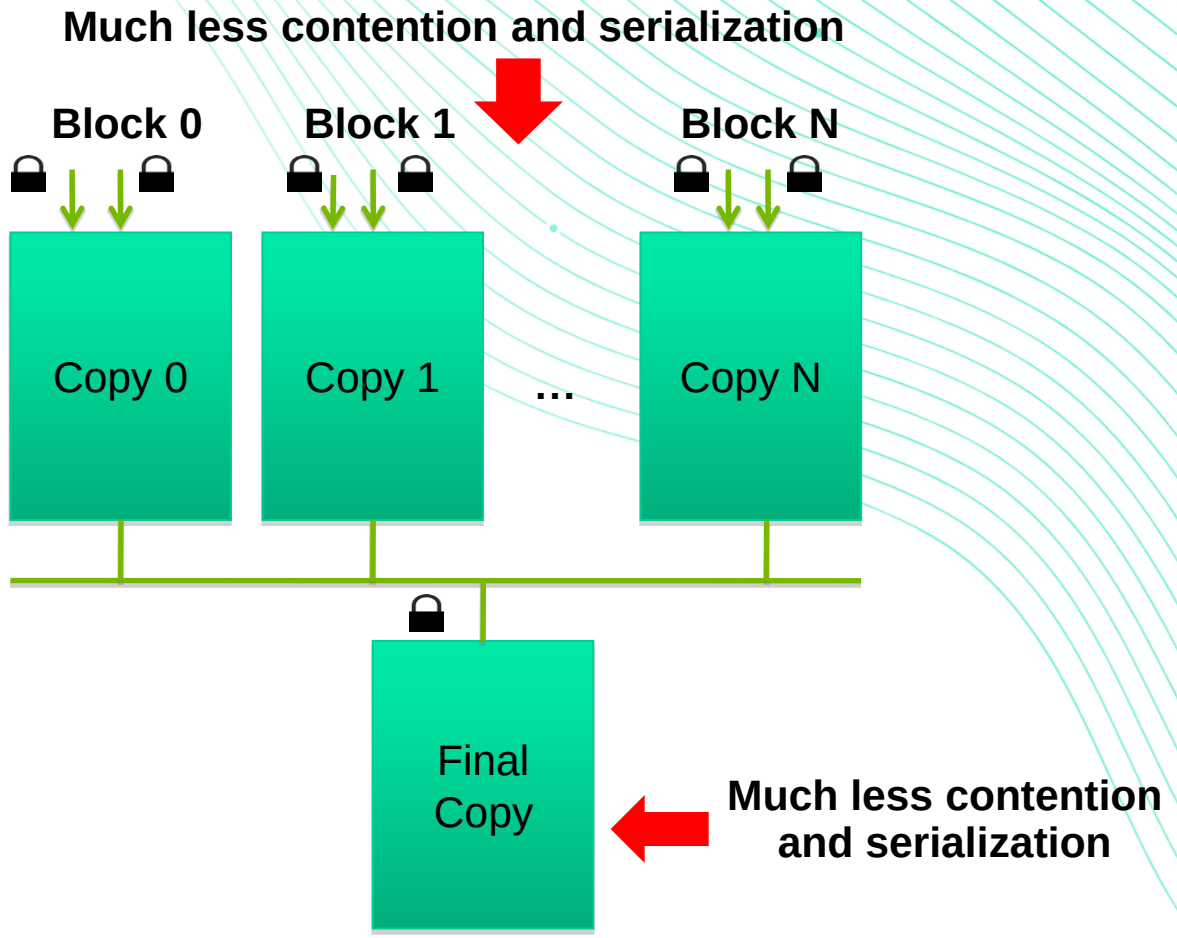
Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Histogram

Privatization

- Privatization is a technique for reducing latency, increasing throughput, and reducing serialization





Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns Histogram

Privatization

- privatization is a technique for reducing latency, increasing throughput, and reducing serialization

Cost and Benefit of Privatization

Cost

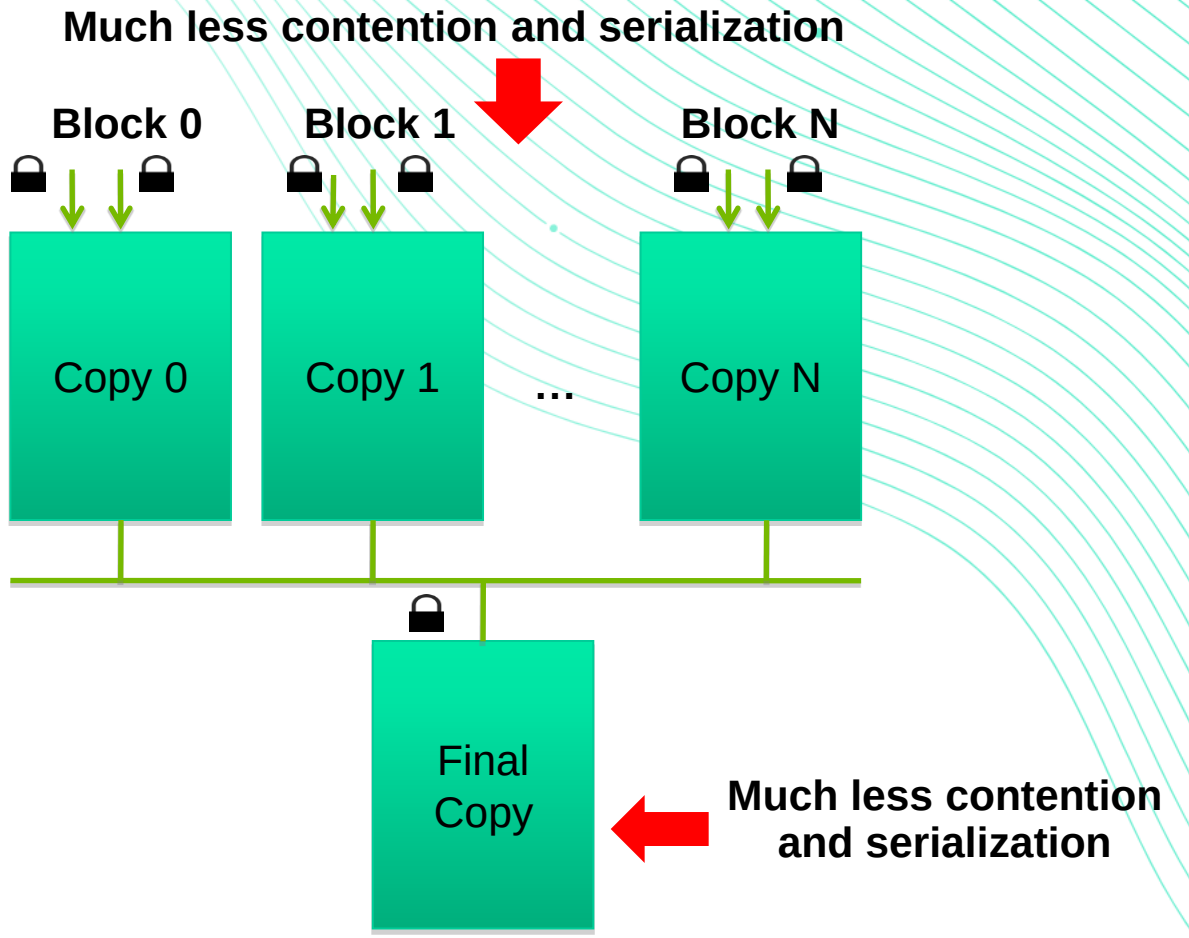
- overhead for creating and initializing private copies
- overhead for accumulating the contents of private copies into the final copy

Benefit

- much less contention and serialization in accessing both the private copies and the final copy
- the overall performance can often be improved more than 10x

Shared Memory Atomics for Histogram

- each subset of threads are in the same block
- much higher throughput than DRAM (100x) or L2 (10x) atomics
- less contention – only threads in the same block can access a shared memory variable
- **this is a very important use case for shared memory!**



Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

Parallel Computation Patterns
Histogram

Privatized Histogram kernel

Create private copies of the histo[] array for each thread block

Initialize the bin counters in the private copies of histo[]

Build Private Histogram

Build Final Histogram

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
  __shared__ unsigned int histo_private[7];

  if (threadIdx.x < 7) histo_private[threadidx.x] = 0;

  __syncthreads();

  int i = threadIdx.x + blockIdx.x * blockDim.x;
  // stride is total number of threads
  int stride = blockDim.x * gridDim.x;
  while (i < size) {
    int alphabet_position = buffer[i] – “a”;
    if (alphabet_position >= 0 && alpha_position < 26) {
      atomicAdd(&(private_histo[alphabet_position/4]), 1); }
    i += stride;
  }

  // wait for all other threads in the block to finish
  __syncthreads();

  if (threadIdx.x < 7) {
    atomicAdd(&(histo[threadIdx.x]), private_histo[threadIdx.x] );
  }
}
```

Slide is partially based on NVIDIA GPU Teaching Kit – Accelerated Computing: https://www.nvidia.com/en-us/training/teaching-kits/

EPICURE

Unlocking European-level HPC Support

Hands On: Histogram

Hands-On Histogram

- `tasks/histogram`
- Finish the TODO1 and TODO2 tasks
  - Naïve implementation – atomicAdd directly to result in global memory
  - Privatization – atomicAdd to shared memory, then atomicAdd the results to global memory
  - TODO3 is a bonus for you
- Again, only the kernel and its launch is up to you
  - Array init and error check already implemented

Expected output:

```
Histogram naive
  Everything seems OK
  Kernel time: 453.779449 ms
Histogram using shared memory
  Everything seems OK
  Kernel time: 24.850431 ms
```

EPICURE

Unlocking European-level HPC Support

Other notable GPU programming models

HIP

- Created by AMD to mimic CUDA
  - To ease users' transition from NVIDIA to AMD GPUs
- Works on both AMD and NVIDIA GPUs
- cuda* functions and types replaced by hip*
- hip* libraries (BLAS etc.)
  - Wrappers around cuda* or roc* functions
- Hipify – convert CUDA source code to HIP code

- ROCm software ecosystem/platform
- roc* libraries (blas, sparse, fft, …)

- Frontier (#1) and LUMI (#5) use AMD GPUs

AMD ROCm

HIP

source.cu

```
__global__ void vector_scale(float * x, float alpha, int count)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < count) x[idx] = alpha * x[idx];
}

int main()
{
    int count = 20 * 256;

    float * h_data = new float[count];
    for(int i = 0; i < count; i++) h_data[i] = i;

    float * d_data;
    cudaMalloc(&d_data, count * sizeof(float));

    cudaMemcpy(d_data, h_data, count * sizeof(float), cudaMemcpyHostToDevice);
    vector_scale<<< 20, 256 >>>(d_data, 10, count);
    cudaMemcpy(h_data, d_data, count * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_data);
    delete[] h_data;
    return 0;
}
```

```
$ nvcc source.cu -o program_cuda.x
```

source.hip.cpp

```
#include <hip/hip_runtime.h>

__global__ void vector_scale(float * x, float alpha, int count)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < count) x[idx] = alpha * x[idx];
}

int main()
{
    int count = 20 * 256;

    float * h_data = new float[count];
    for(int i = 0; i < count; i++) h_data[i] = i;

    float * d_data;
    hipMalloc(&d_data, count * sizeof(float));

    hipMemcpy(d_data, h_data, count * sizeof(float), hipMemcpyHostToDevice);
    vector_scale<<< 20, 256 >>>(d_data, 10, count);
    hipMemcpy(h_data, d_data, count * sizeof(float), hipMemcpyDeviceToHost);

    hipFree(d_data);
    delete[] h_data;
    return 0;
}
```

```
$ hipcc source.hip.cpp -o program_hip.x
```

SYCL

- Open standard, modern C++17 interface
- A way to do parallel programming not only for GPUs
  - CPUs, FPGAs
- Primary way to utilize Intel GPUs
  - Aurora supercomputer (#2)
- Source code portability. Not necessarily performance portability.
- Implementations for all of Intel, AMD and NVIDIA GPUs exist
  - DPC++ (Intel), AdaptiveCPP
- oneAPI – SYCL interface for high performance libraries (BLAS, SPARSE, FFT, …)
  - Also a standard
  - Has implementations for all of Intel, AMD and NVIDIA GPUs
    - Intel's oneAPI, Codeplay

EPICURE

Unlocking European-level HPC Support

Thank you!

Follow us

pmo-epicure@postit.csc.fi

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High Performance Computing Joint Undertaking under grant agreementNo.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union orEuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

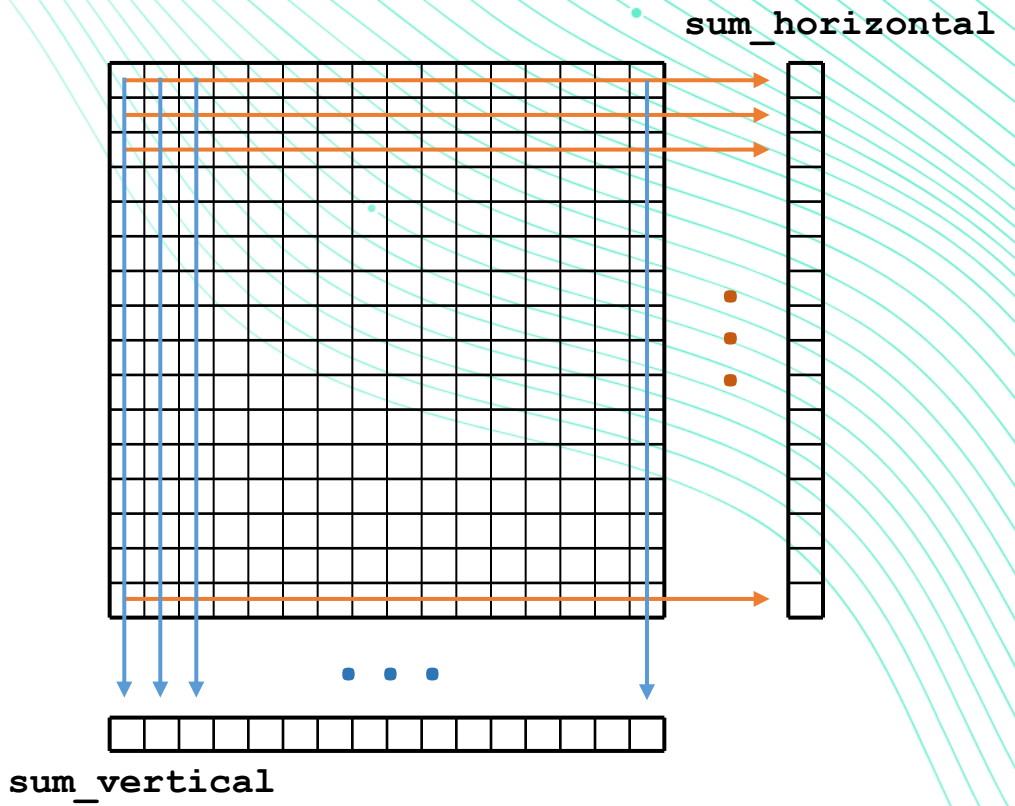
EPICURE

Unlocking European-level HPC Support

Hands-on
Matrix sum

Hands-on
Matrix sum

- `tasks/matrix_sum`
- Sum of values in a matrix
  - Horizontally
  - Vertically
- Complete the TODO tasks
  - Implement the two kernels
  - 1D kernels iterating over the rows/columns
- Think about the memory access pattern
  - Do not think about each thread individually, think about the threadblock (or rather warp) as a whole



Expected output:

```
Horizontal sum seems OK
Vertical sum seems OK

Matrix init:              25.087 ms
Matrix sum horizontal:    61.935 ms
Matrix sum vertical:      30.567 ms
Using coalesced memory accesses was  2.03 times faster
```

https://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf