Harold Molina-Bulla DTSC-UC3M

HETEROGENEOUS COMPUTATION BASED ON GPU

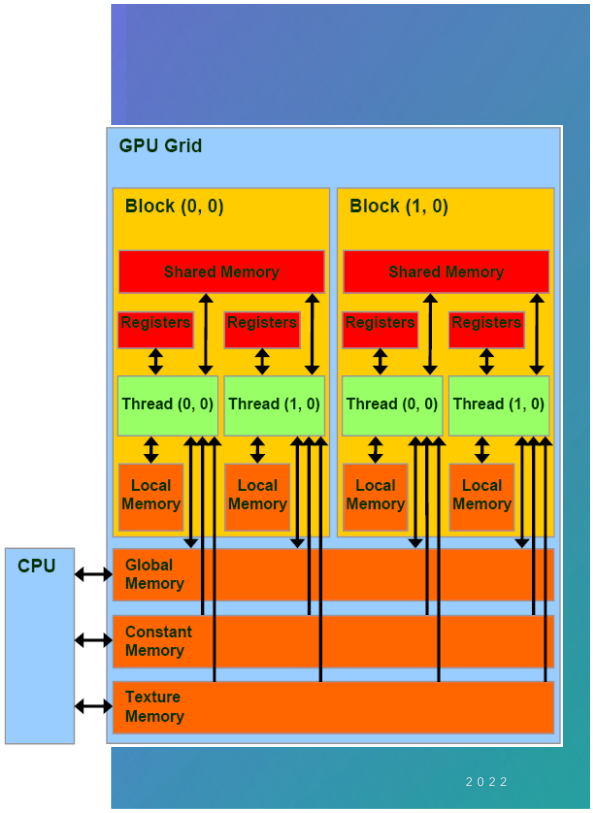
TILED MEMORY AND SYNCHRONIZED THREADS

2022

Harold Molina-Bulla DTSC-UC3M

MEMORY IN CUDA DEVICES

- The memory in CUDA devices is organized at 3 levels:
  - Global Memory (included Constant and Texture), it is the memory accessible by all the processors chips.
    - I will write information by burst of data (size of burst depends on the Memory kind)
  - Shared Memory: Placed ON the Processor chip (The GPUs can have several processors, each one with several execution cores)
    - The Shared Memory is accessible by all the execution cores at same time, but logical restricted to the block
  - Local Memory: The memory only accessible by the executional core.



2022

MEMORY ACCESS IN GPUS

- The global memory access in the GPUs is by burst:
  - Access to the global GPU memory is slow (in comparison with other memory available)
  - Fetch not only the requested memory position, but also the adjacent memory data.
  - It reads DWORDS (Data Words), 32 bits length

Harold Molina-Bulla DTSC-UC3M

SHARED MEMORY

- Direct access from all computational cores in the same processor
- Fast access to this memory place
- Access controlled and restricted to all threads in the same logical block.

LOCAL MEMORY

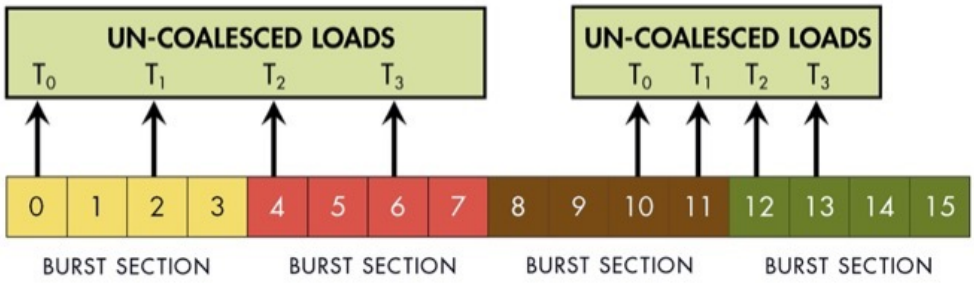
- Only available in the executional core
- Difference between thread and executional core:
  - Core is the hardware
  - Thread execute the program in the assigned core

MEMORY ACCESS OPTIMIZATION

- If the data is not continuous in the memory you will need the same number of memory access as the number of DWORDS you want to access for each one of the executional cores.
- But, if the data is continuous for a continuous set of logical threads, the other treads will have access to the fetched extra data.
- This is Memory Coalescing.

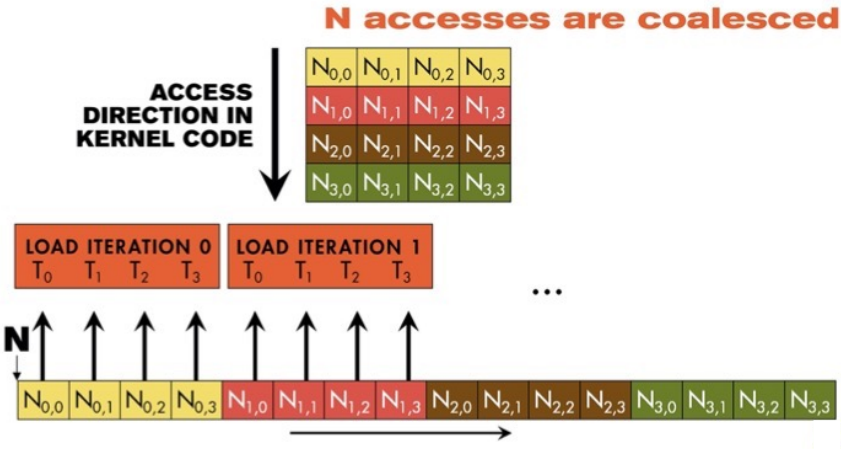
MEMORY COALESCING

MEMORY UNCOALESCING



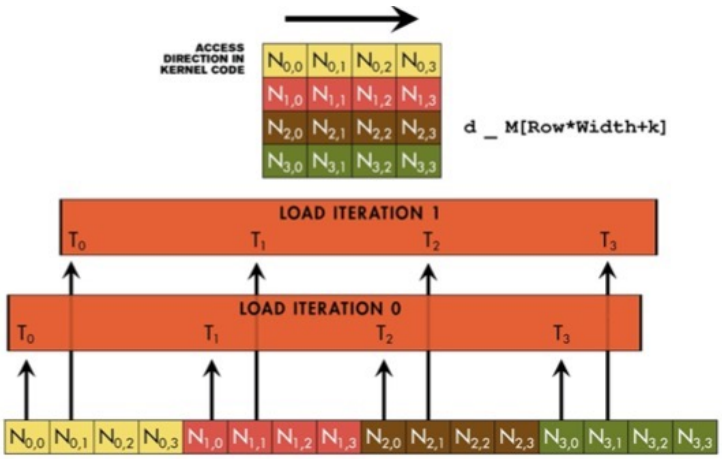
Harold Molina-Bulla DTSC-UC3M

ROW ACCESS ROW-WISE MATRIX



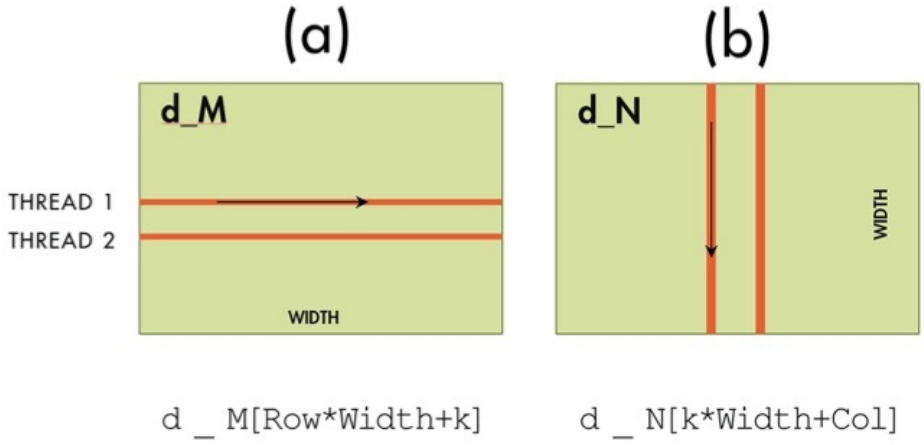
Harold Molina-Bulla DTSC-UC3M

COLUMN ACCESS ROW WISE MATRIX



Harold Molina-Bulla DTSC-UC3M

CLASSICAL MATRIX MULTIPLICATION ACCESS



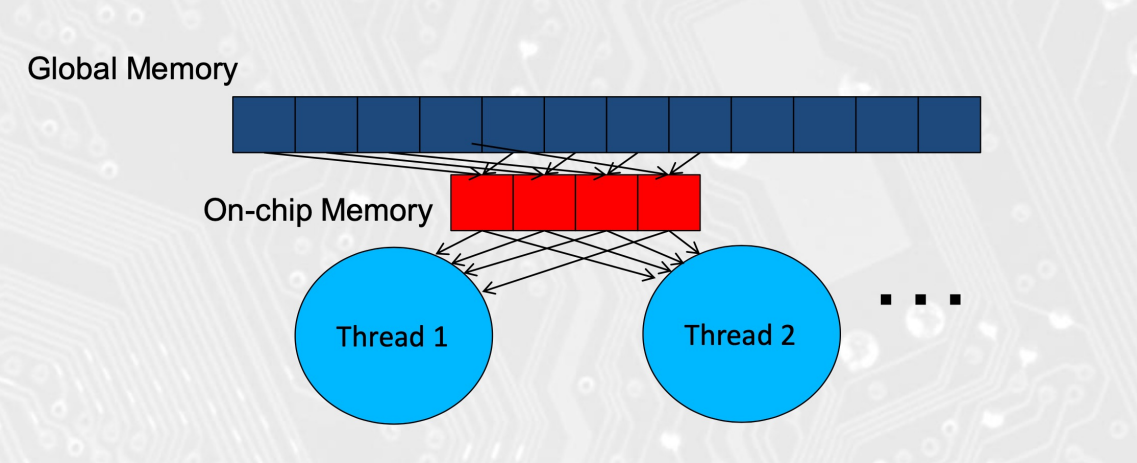
```
kernel  =  SourceModule ("""
__global__ void matrix_mult(float* a,
                            float* b,
                            float* c,
                            int m,
                            int n,
                            int o)
{
  // a is the vector which represents the matrix_a
  // b is the vector which represents the matrix_b
  // c is the vector where we will store the resulting matrix
  // m is number of rows of matrix a
  // n is the number of columns of matrix a, and number of rows of matrix b
  // o is the number of columns of matrix b
  // First task: Using threadIdx.x, threadIdx.y, blockDim.x, blockDim.y,
  // blockIdx.x, blockDim.y identify the coordinates x and y in the result matrix
  // implements the matrix multiplication, cell by cell using the conventional code and analyze

  int idxX;
  int idxY;
  int idxZ;
  int offA;

  float s;

  idxY = blockIdx.y*blockDim.y+threadIdx.y; //With this we calculate the row address in target matrix
  idxX = blockIdx.x*blockDim.x+threadIdx.x; //Here we calculate the column address in target matrix


  if ( idxX < o && idxY < m ){    //Here we verify the row address and column address are valid
    idxZ = idxY*o + idxX;         //Here we calculate the target vector address,
                                  //asuming it is a row wise matrix representation
    s = 0;                        //Initialize the s acumulator

    offA = idxY*n;                //We calculate the offset of row in matrix a row wise representation
                                  //This is to reduce the number of calculae in the next for

    for ( int i =0; i<n; i++)     //Here we run through the a columns, b rows
      s += a[offA+i]*b[(i*o)+idxX];

    c[idxZ]=s;
  }

}
""")
```


Harold Molina-Bulla DTSC-UC3M

OPTIMIZE MEMORY USE

- Try to fetch the data coalesced
- One trick:
  - USE MEMORY TILING AND SHARED MEMORY
- Memory Tiling: use shared memory to store temporally the data
- Load the date from the global memory, fetching it coalesced
  - To load the data use coalesced threads to read the data from the burst
  - Each thread save his assigned data in the shared memory
  - Now, all the threads (in the block) have access to the data in the shared memory

SHARED MEMORY TILING

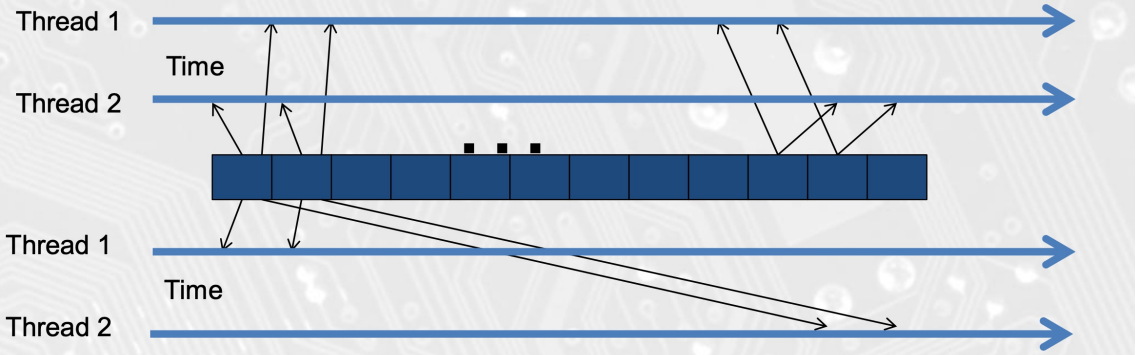


PROBLEMS WITH TILING MEMORY

- Synchronization
  - The threads can have small time difference in execution
  - Needs to coordinate and synchronize the thread
- There are special commands un CUDA C to synchronize the execution:
  - __synchthreads();

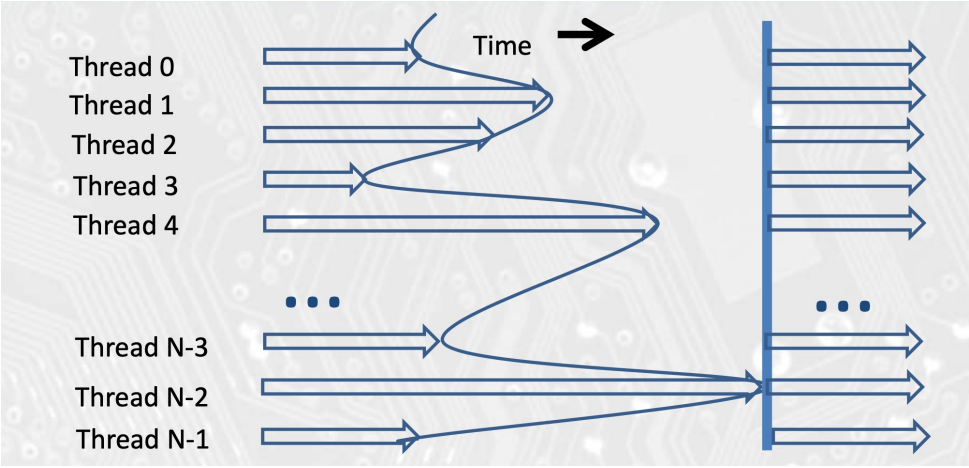
BLOCKING/TILING NEEDS SYNCHRONIZATION

- Good – when threads have similar access timing



- Bad – when threads have very different timing

Harold Molina-Bulla DTSC-UC3M



2022

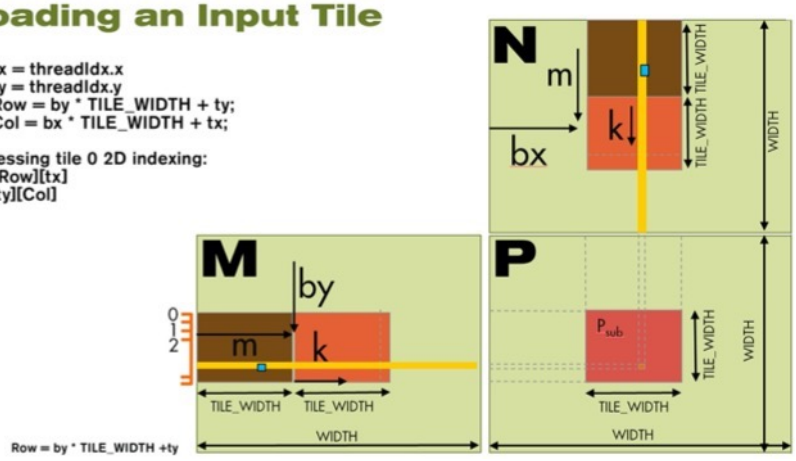
OUTLINE OF TECHNIQUE

- Identify a tile of global memory content that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Wait until all threads ends to copy the data into on-chip memory (sync threads)
- Have the multiple threads to access their data from the on-chip memory
- Wait (again) until each thread ends his own assigned task.
- Move on to the next tile and repeat

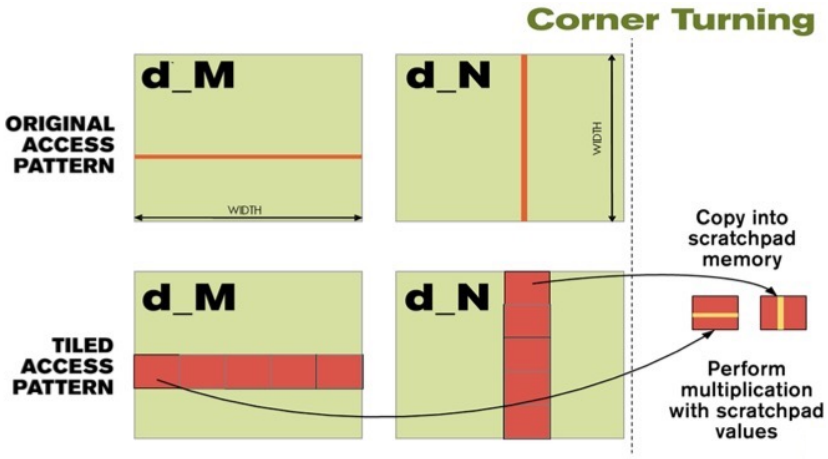
SOLUTION: TILE COLABORATIVE ACCESS

Loading an Input Tile

```
int tx = threadIdx.x
int ty = threadIdx.y
int Row = by * TILE_WIDTH + ty;
int Col = bx * TILE_WIDTH + tx;

Accessing tile 0 2D indexing:
  M[Row][tx]
  N[ty][Col]
```



TILE COLABORATIVE ACCESS



DEFINE SHARED ON-CHIP MEMORY

Here, we define the memory space on-chip for our shared information using meta marker __shared__.
In this case, we defines 2 memory blocks, one for matrix A, and the second for matrix B

```
// Shared memory for the sub-matrix of A
__shared__ float As[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];
// Shared memory for the sub-matrix of B
__shared__ float Bs[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];
```

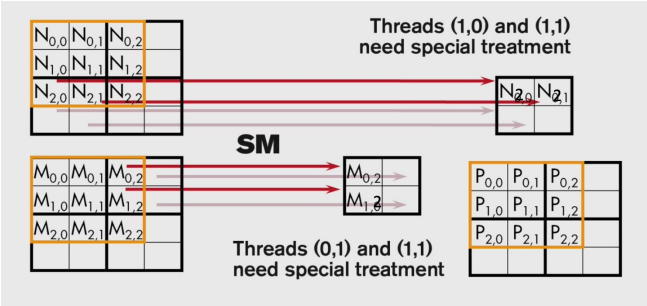
FILL ON-CHIP MEMORY DATA

- This part is a collaborative part: each thread in the block fills the on-chip memory fetching part of the data from global memory, in an ordered steps.
- We need to wait all the threads ends his small task, putting a __syncthreads() waiting function.

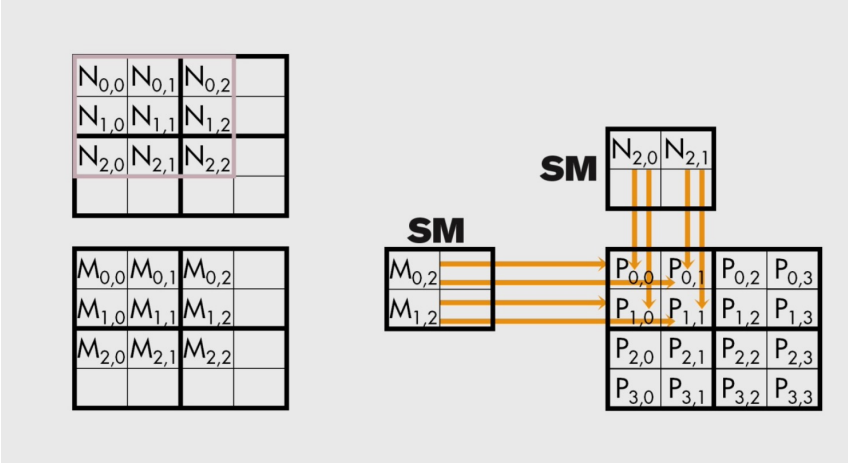
```
// Load the matrices from global memory to shared memory
// each thread loads one element of each matrix
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];
// Synchronize to make sure the matrices are loaded
__syncthreads();
```


SOLUTION OF SECOND PROBLEM

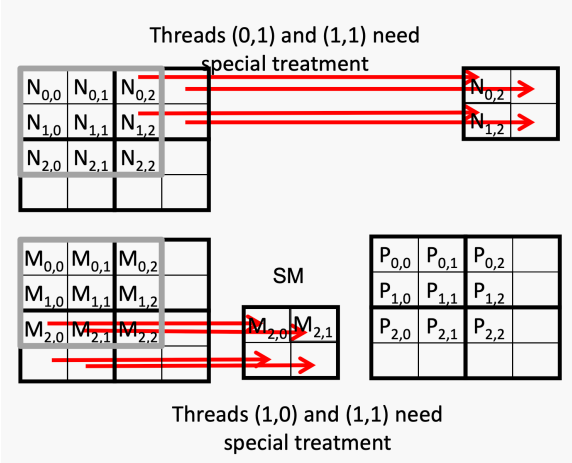
- We can not fill of zeroes the main matrices, BUT, may fill of zeroes the tiled matrices:



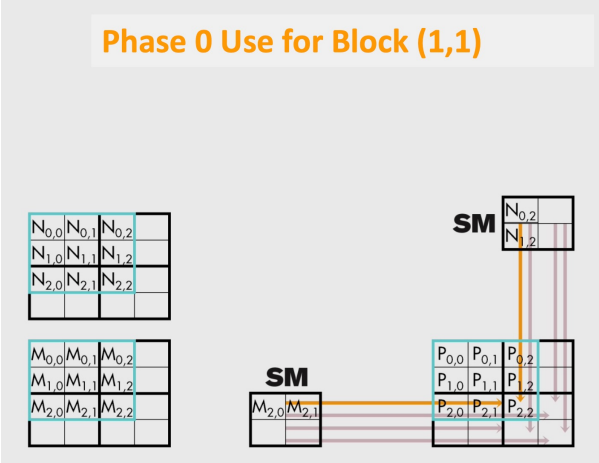
CASE BLOCK 0,0: NO PROBLEM



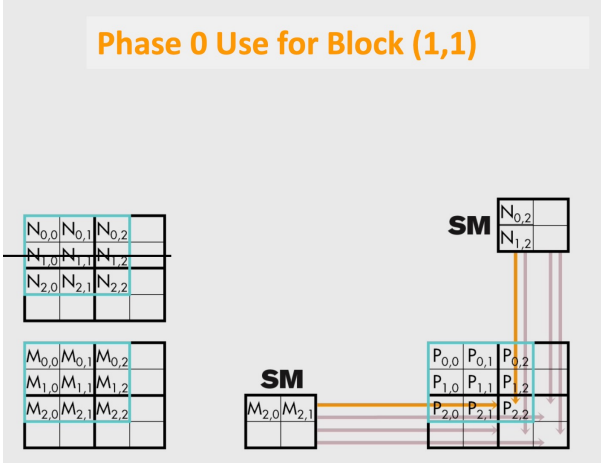
CASE OTHER BLOCKS



CASE IN STEP2: ADDING

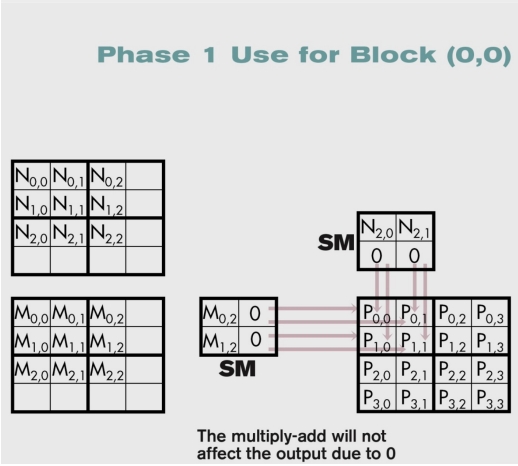


Harold Molina-Bulla DTSC-UC3M

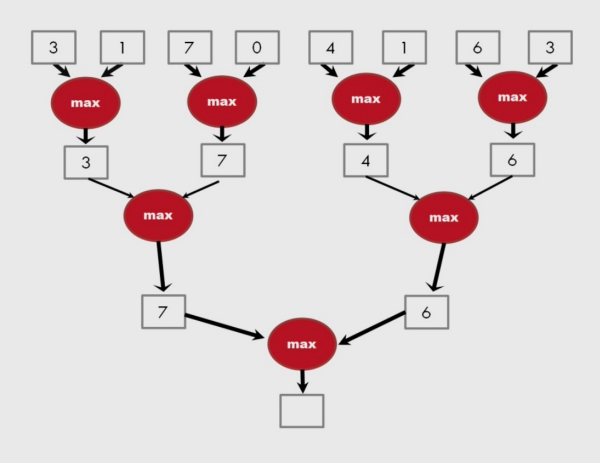


2022

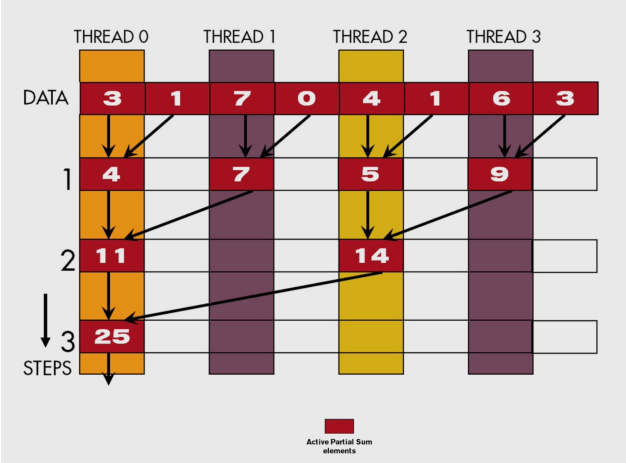
POSSIBLE SOLUTION: FILL WITH ZEROES THE NON USED CELLS



PARALLEL SUM REDUCTION



PARALLEL REDUCTION



Harold Molina-Bulla DTSC-UC3M

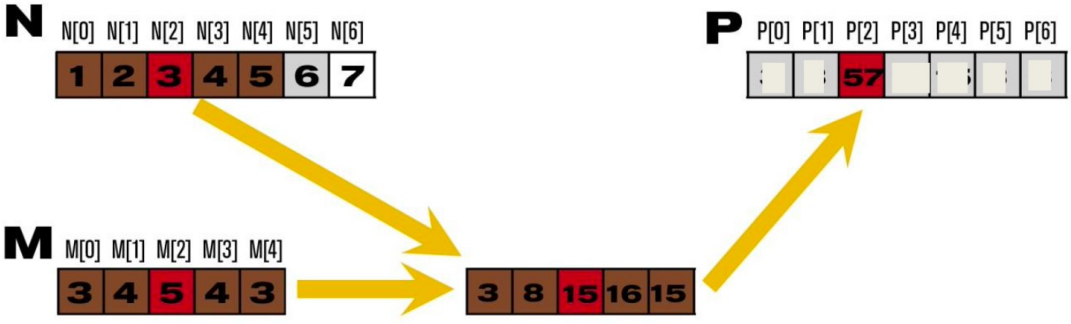
PARALLEL REDUCTION

```
__ 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];
```

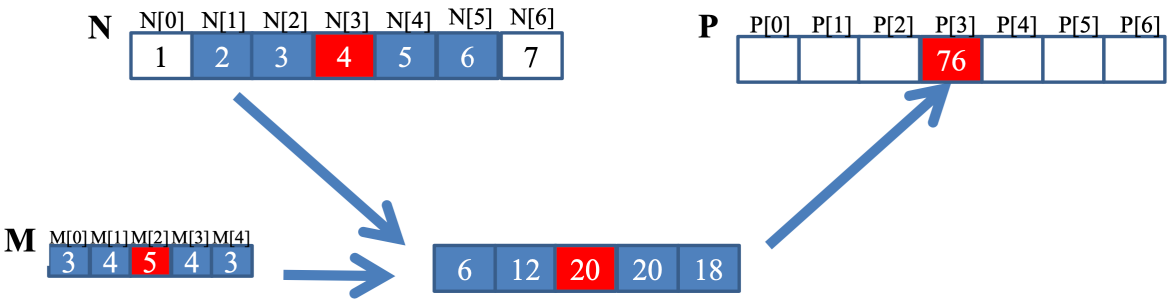
PARALLEL REDUCTION

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


1D CONVOLUTION

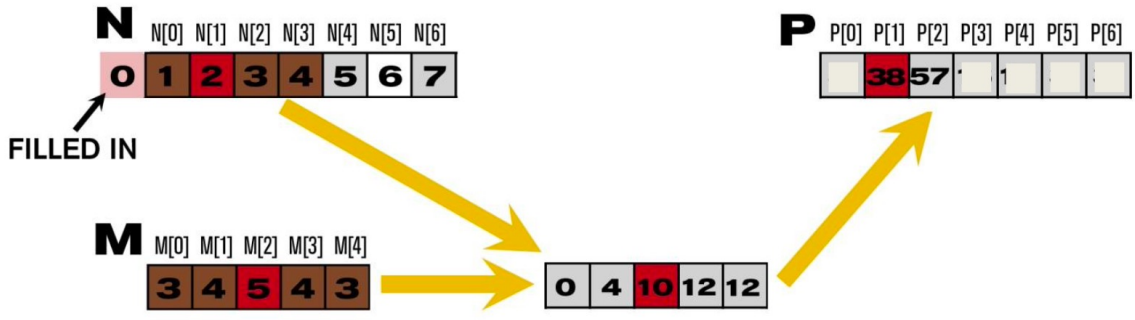


1 D CONVOLUTION



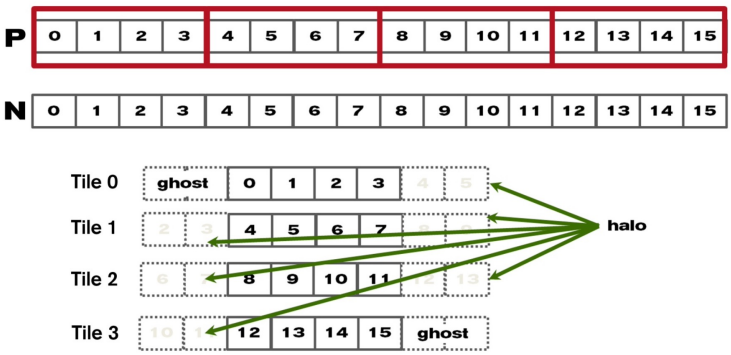
Harold Molina-Bulla DTSC-UC3M

1 D CONVOLUTION



TILED 1D CONVOLUTION

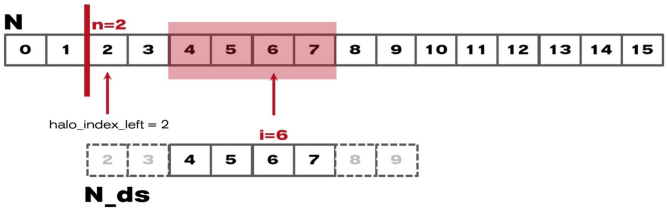
Tiled 1D Convolution Basic Idea



Harold Molina-Bulla DTSC-UC3M

TILED 1D CONVOLUTION

Loading the left halo

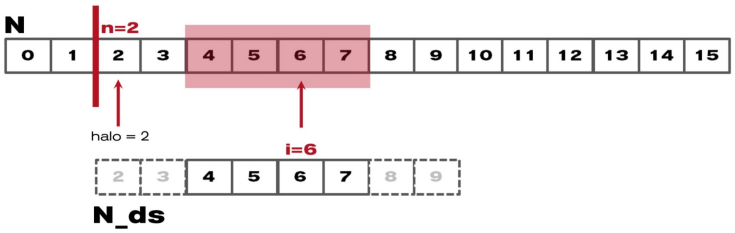



```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
 if (threadIdx.x >= blockDim.x - n) {
   N_ds[threadIdx.x - (blockDim.x - n)] =
     (halo_index_left < 0) ? 0 : N[halo_index_left];
 }
```

2022

TILED 1D CONVOLUTION

Loading the internal elements

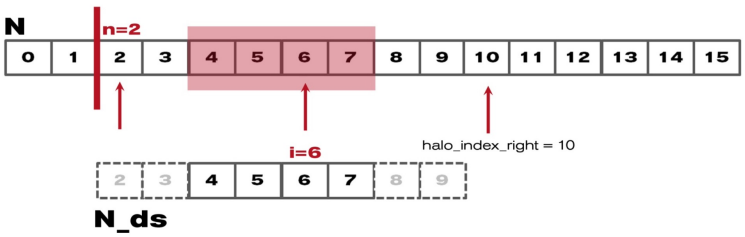



```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
```


Harold Molina-Bulla DTSC-UC3M

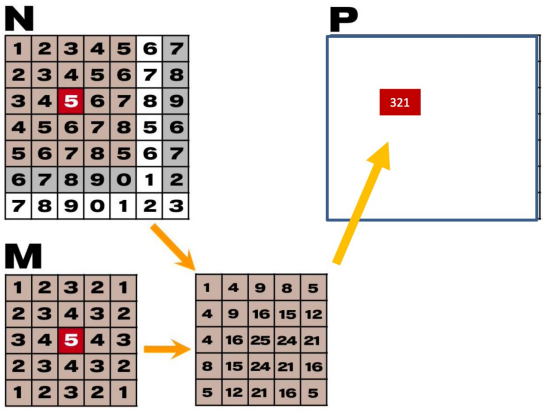
TILED 1D CONVOLUTION

Loading the right halo




```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < n) {
   N_ds[n + blockDim.x + threadIdx.x] =
     (halo_index_right >= Width) ? 0 : N[halo_index_right];
 }
```

2-D CONVOLUTION



Harold Molina-Bulla DTSC-UC3M

2-D CONVOLUTION