COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management Introduction to CUDA C: Part II

Hairong Wang

School of Computer Science,
University of the Witwatersrand, Johannesburg

Semester one 2025

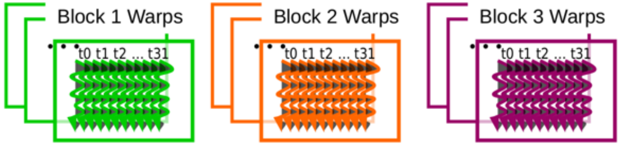
Contents

Outline

WITS UNIVERSITY

CUDA Thread Organization

- All threads in a grid execute the same kernel;
- All threads in a grid rely on coordinates to distinguish themselves from each other;
- All threads in a grid rely on coordinates to identify the appropriate portion of the data to process.
- The threads are organized into two level hierarchy: grid and block
- All threads in a block share the same block index, which can be accessed through `blockIdx`.
- Each thread has a thread index, accessed through `threadIdx`.
- Execution configuration parameters: `dim3` type parameters.



WITS UNIVERSITY

CUDA Thread Organization Contd.

- For the vector addition example, assume the vector size is a variable `n`. Then the kernel execution parameters can be determined by `n` as the following.

  ```
  dim3 dimGrid(ceil(n/256.0), 1, 1);
  dim3 dimBlock(256, 1, 1);
  vecAddKernel<<<dimGrid, dimBlock>>>(...);
  ```

- This will allow the number of blocks to vary with the size of vectors so that the grid will have enough threads to cover all vector elements.
- The kernel can also be launched as:

```
vecAddKernel<<<ceil(n/256.0), 256>>>(...);
```

Mapping Threads to Multi-Dimensional Data

To process a image of size 62×76:

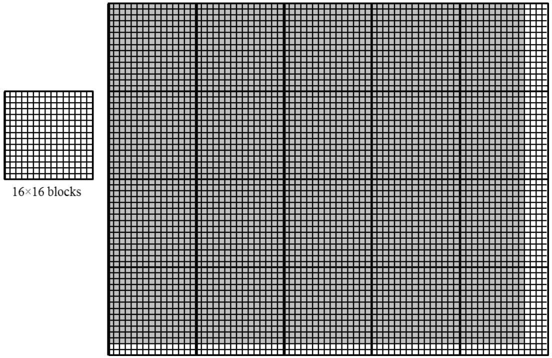


Figure: Using a block of threads to process a picture

PictureKernel Code

```
__global__ void PictureKernel(float* d_Pin,float* d_Pout,int n,int m) {
  //calculate the row index of the d_Pin, d_Pout element to process
  int Row = blockIdx.y*blockDim.y + threadIdx.y;
  //calculate the column index of the d_Pin, d_Pout element to process
  int Col = blockIdx.x*blockDim.x + threadIdx.x;
  //each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
  }
}
```

Mapping Threads to Multi-Dimensional Data Contd.

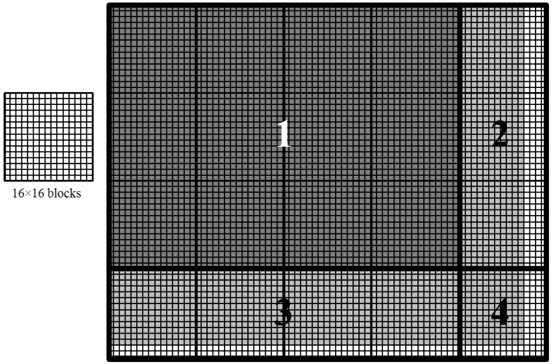


Figure: Covering a 62 $\times$ 76 picture with 16 $\times$ 16 blocks

Outline

CUDA Device Memory Types

CUDA variable type qualifiers			
Variable declaration	Memory	Scope	Lifetime
Automatic scalar variables	Register or local	Thread	Kernel
Automatic array variables	Register or local	Thread	Kernel
`__shared__`	Shared	Block	Kernel
`__device__`	Global	Grid	Application
`__constant__`	Constant	Grid	Application

Outline

CUDA Error Handling

- Almost all function calls in CUDA return the error type `cudaError_t`, which is an integer.
- Any value other than `cudaSuccess` indicates a fatal error.
- Every function returns an error code that should be checked and some handler written.

```
#define CUDA_CALL(x) {\
  const cudaError_t a = (x);\
  if (a != cudaSuccess) {\
    printf("\nCUDA Error: %s (err_num= %d) \n",\
    cudaGetErrorString(a), a);\
    cudaDeviceReset(); assert(0);}}
```

Outline

WITS UNIVERSITY

Outline

A Simple Running Example 1: Histogram

Objectives

- A simple illustration of the basic features of memory and thread management in CUDA programs
  - Thread index usage
  - Memory layout
  - Register usage
  - Shared memory usage

Histogram

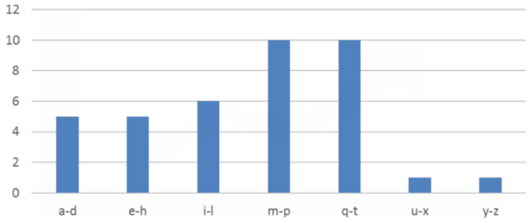
Problem: Count the distribution of data over a number of “bins”. If a data point is associated with a given bin, the value of the bin is incremented.

- Serial

```
for(int i = 0; i < max; i++)
    bin[array[i]]++;
```

A text Histogram Example

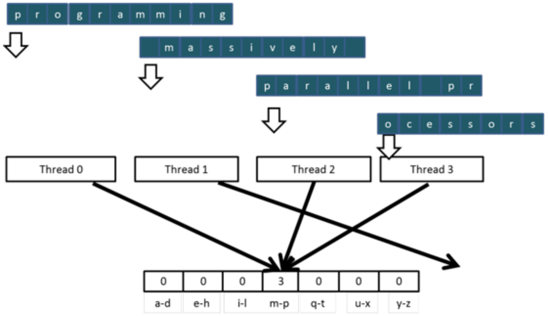
- Define the bins as four-letter sections of the alphabet: a - d, e - h, i - l ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase“Programming Massively Parallel Processor” the output histogram is shown below:



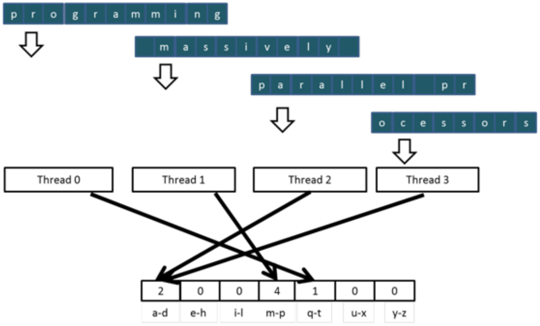
Outline

WITS UNIVERSITY

Sectioned Partitioning (Iteration #1)



Sectioned Partitioning (Iteration #2)



Sectioned Partitioning Kernel Function

```
__global__ void histoGPU_1(uchar *input, uint *d_histo, long size){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int section_size = (size - 1) / (blockDim.x * gridDim.x) + 1;
  int start = i * section_size;
  int pos, k;
  for(k = 0; k < section_size; k++){
    if(start + k < size){
      pos = input[start+k];
      if(pos >=0 && pos < 256)
        atomicAdd(&(d_histo[pos]), 1);
    }
  }
}
```

- `atomicAdd(addr,y)` - generates an atomic sequence of operations that read the value at address `addr`, adds `y` to that value, and stores the result back to the memory address `addr`.

Sectioned Partitioning Kernel Function Cont.

blockIdx.x	threadIdx.x												
0	0	1	2	3	4	5	6	7	8	9	...	14	15
1	0	1	2	3	4	5	6	7	8	9	...	14	15
2	0	1	2	3	4	5	6	7	8	9	...	14	15
3	0	1	2	3	4	5	6	7	8	9	...	14	15
4	0	1	2	3	4	5	6	7	8	9	...	14	15
5	0	1	2	3	4	5	6	7	8	9	...	14	15

- `blockDim.x = 16; gridDim.x = 6;`
- The linearized index of the element in blue (`threadIdx.x=7, blockIdx.x=2`) is $2 * 16 + 7 = 39$.

Outline

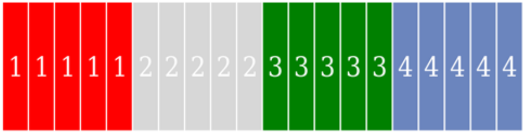
Atomic Operations

- Performed by calling functions that are translated into single instructions
- 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`. The function returns `old`.
- The implementation of atomic functions ensures that no other threads will access the value at `address` when a thread is updating its value. Hence, a predictable result is guaranteed.

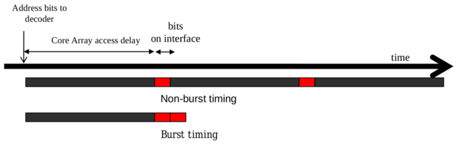
WITS UNIVERSITY

Sectioned Partitioning

- 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

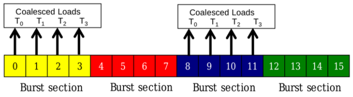


Coalesced Memory Access



Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.

Coalesced Memory Access



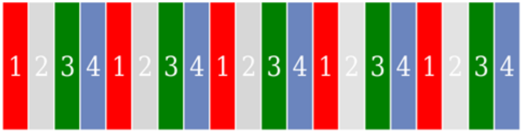
When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

Outline

WITS UNIVERSITY

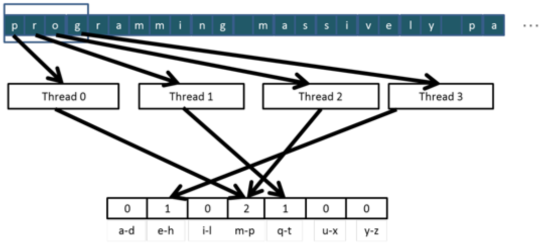
Interleaved Partitioning

- Interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced

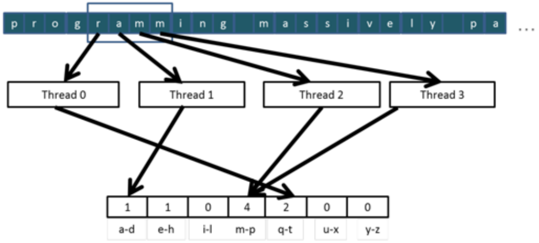


Interleaved Partitioning of Input (Iteration #1)

For coalescing and better memory access performance

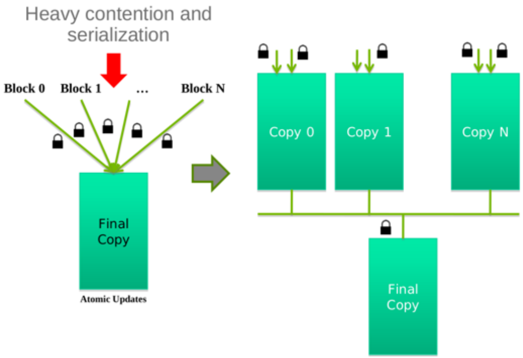


Interleaved Partitioning of Input (Iteration #2)



Outline

Using Shared Memory



Using Shared Memory Cont.

- Each block computes its own histogram.
- Since each block can compute independently, we can use shared memory.
- Using shared memory usually involves the following
  1. Allocate a shared memory buffer to hold each block's intermediate histogram;
  2. The threads in each block compute the histogram in shared memory;

```
  __shared__ unsigned int local_histo[256];
  if threadIdx.x < 256
    local_histo[threadIdx.x]=0;
  __syncthreads();
```

```
int i=threadIdx.x + blockIdx.x * blockDim.x;
int stride=blockDim.x * gridDim.x;
while(i<size){
  atomicAdd(&local_histo[data[i]],1);
  i=i+stride;
}
__syncthreads();
```

Using Shared Memory Cont.

1. Allocate a shared memory buffer to hold each block's intermediate histogram;
2. The threads in each block compute the histogram in shared memory;
3. Merge the histogram results from all the blocks.

```
__syncthreads();
atomicAdd(&(global_histo[threadIdx.x]),local_histo[threadIdx.x]);
```

Note we assume the number of bins and the number of threads in a block are the same in the last `atomicAdd`. If this is not the case, we can not use it as such.

Outline

Outline

Square Matrix Multiplication Example

Matrix multiplication, `C = AB`

- Each thread calculates one element of `C`
- Each row of `A` is loaded `B.width` times from global memory
- Each column of `B` is loaded `A.height` times from global memory

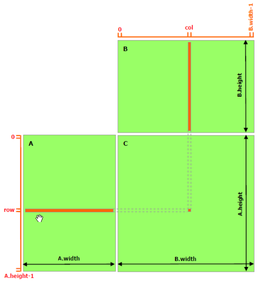


Figure: Matrix multiplication example

Square Matrix Multiplication: A Host Version

```
void MatrixMulOnHost(float* M, float* N, float* P, int width){
  double a, b;
  for(int i = 0; i<width, ++i){
    for (int j = 0; j < width; ++j){
      double sum = 0.0;
          for (int k = 0; k < width; ++k){
            a = M[i * width + k];
            b = N[k * width + j];
            sum += a * b;
          }
          P[i * width + j] = sum;
      }
  }
}
```

Square Matrix Multiplication Example Contd.

Mapping the 2D array to a linear array:

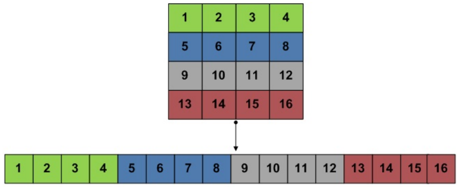


Figure: Row-major layout of a 2-D array

Parallelizing Square Matrix Multiplication

- Compute $P = MN$
- Have each 2-D thread block to compute a `BLOCK_WIDTH * BLOCK_WIDTH` sub-matrix (block) of the result matrix. Each block has `BLOCK_WIDTH * BLOCK_WIDTH` threads.
- Generate a 2-D Grid of `width/BLOCK_WIDTH * width/BLOCK_WIDTH` blocks.
- **Example:** `width = 4, BLOCK_WIDTH = 2`. Then `width/BLOCK_WIDTH = 2`. We have $2 \times 2 = 4$ blocks, with $2 \times 2 = 4$ threads each.

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

$\color{blue}{P_{0,0}}$	$\color{blue}{P_{0,1}}$	$\color{red}{P_{0,2}}$	$\color{red}{P_{0,3}}$
$\color{blue}{P_{1,0}}$	$\color{blue}{P_{1,1}}$	$\color{red}{P_{1,2}}$	$\color{red}{P_{1,3}}$
$\color{green}{P_{2,0}}$	$\color{green}{P_{2,1}}$	$P_{2,2}$	$P_{2,3}$
$\color{green}{P_{3,0}}$	$\color{green}{P_{3,1}}$	$P_{3,2}$	$P_{3,3}$

Square Matrix Multiplication Example: A Bigger Example

- **Example:** `width = 8, BLOCK_WIDTH = 4`. Then `width/BLOCK_WIDTH = 2`. We have $2 \times 2 = 4$ blocks, with $4 \times 4 = 16$ threads each.

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

Square Matrix Multiplication: Kernel Invocation

- Setup the execution configuration

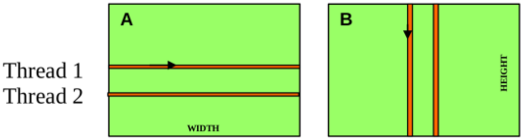
```
// BLOCK_WIDTH is a #define constant
int numBlocks=width/BLOCK_WIDTH;
if(width % BLOCK_WIDTH) numBlocks++;
dim3 dimGrid(numBlocks, numBlocks, 1);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);
......
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);
```

A Simple Matrix Multiplication Kernel

A Simple Matrix Multiplication Kernel using one thread to compute one output element

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int
     width) {
  // calculate the row index of the d_P and d_M
  int row = blockIdx.y*blockDim.y+threadIdx.y;
  // calculate the column idenx of d_P and d_N
  int col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((row < width) && (col < width)) {
    float Pvalue = 0.0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < width; ++k)
      Pvalue += d_M[row*width+k] * d_N[k*width+col];
    d_P[row*width+col] = Pvalue;
  }
}
```

Two Access Patterns of Basic Matrix Multiplication

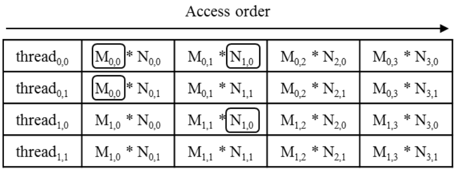


A[Row*n+i]   B[i*k+Col]

Outline

A Strategy for Reducing Global Memory Traffic – Tiling

- Global memory resides in device memory (DRAM) - slow access
- A profitable way of performing computation on the device is to tile the input data to take advantage of fast shared memory:
    - Partition data into subsets that fit into shared memory
    - Handle each data subset with one thread block by:
        - Loading the subset from global memory to shared memory using multiple threads;
        - Performing the computation on the subset from shared memory;
        - Copying results from shared memory to global memory
- Note that not all data structures can be partitioned into tiles.


Access order

$\text{thread}_{0,0}$	$\boxed{M_{0,0}} * N_{0,0}$	$M_{0,1} * \boxed{N_{1,0}}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$\text{thread}_{0,1}$	$\boxed{M_{0,0}} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$\text{thread}_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * \boxed{N_{1,0}}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$\text{thread}_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

Outline

Matrix Multiplication Using Shared Memory

Outline of the technique

- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile

Tiled Matrix Multiplication

Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of M and N.

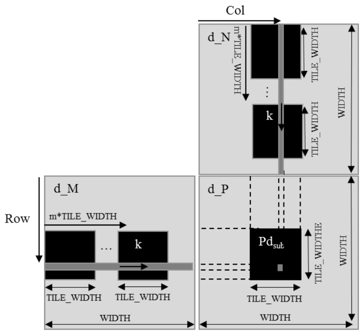


Figure: Tiled matrix multiplication

Sqaure Matrix Multiplication: Kernel Function Contd.

Thread $(0, 0)$ compute $P_{0,0}$:

```
col = blockIdx.x * blockDim.x + threadIdx.x
    = 0 * 2 + threadIdx.x
row = blockIdx.y * blockDim.y + threadIdx.y
    = 0 * 2 + threadIdx.y
```

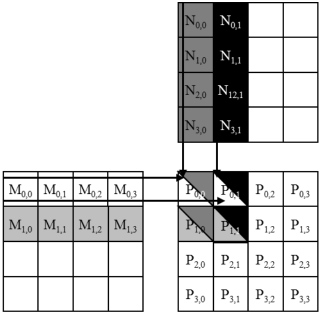


Figure: Matrix multiplication

Square Matrix Multiplication: Kernel Function Contd.

Thread $(1, 0)$ compute $P_{0,1}$:

```
col = blockIdx.x * blockDim.x + threadIdx.x
    = 0 * 2 + threadIdx.x
row = blockIdx.y * blockDim.y + threadIdx.y
    = 0 * 2 + threadIdx.y
```

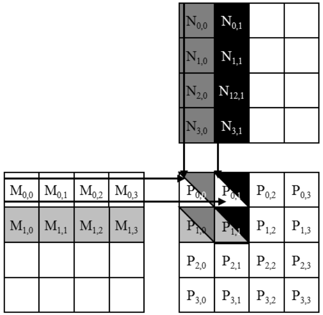


Figure: Matrix multiplication

Loading a Tile

- All threads in a block participate
    - Each thread loads one element from M and one element from N
- Assign the loaded element to each thread such that the accesses within each warp is coalesced.

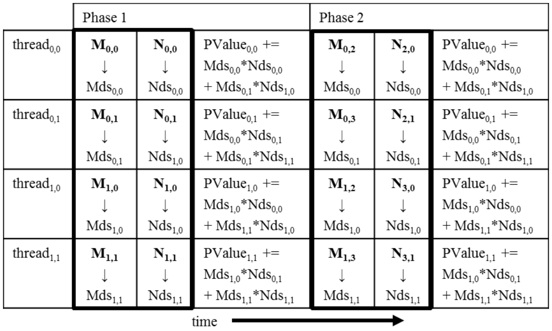
Let `tx=threadIdx.x, ty=threadIdx.y`
`bx=blockIdx.x, by=blockIdx.y`

- `row=by*TILE_WIDTH+ty,`
- `col=bx*TILE_WIDTH+tx`
- Loading a tile: Phase `m=0` to `WIDTH/TILE_WIDTH`
- 2-D index: `M[row][m*TILE_WIDTH+tx],`
  1-D index: `M[row*WIDTH+m*TILE_WIDTH+tx]`
- 2-D index: `N[m*TILE_WIDTH+ty][col],`
  1-D index: `N[(m*TILE_WIDTH+ty)*WIDTH+col]`

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_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;
  //identify the row and column of the d_P element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;
  //loop over the tiles
  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph){
  //colaborative loading of d_M and d_N tiles into shared memory
    ds_M[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH+tx];
    ds_N[ty][tx] = d_N[Col+(ph*TILE_WIDTH+ty)*Width];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += ds_M[ty][k] * ds_N[k][tx];
    __synchthreads();
  }
  d_P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Contd.



	Phase 1			Phase 2		
thread$_{0,0}$	$\mathbf{M_{0,0}}$ $\downarrow$ Mds$_{0,0}$	$\mathbf{N_{0,0}}$ $\downarrow$ Nds$_{0,0}$	PValue$_{0,0}$ += Mds$_{0,0}$*Nds$_{0,0}$ + Mds$_{0,1}$*Nds$_{1,0}$	$\mathbf{M_{0,2}}$ $\downarrow$ Mds$_{0,0}$	$\mathbf{N_{2,0}}$ $\downarrow$ Nds$_{0,0}$	PValue$_{0,0}$ += Mds$_{0,0}$*Nds$_{0,0}$ + Mds$_{0,1}$*Nds$_{1,0}$
thread$_{0,1}$	$\mathbf{M_{0,1}}$ $\downarrow$ Mds$_{0,1}$	$\mathbf{N_{0,1}}$ $\downarrow$ Nds$_{1,0}$	PValue$_{0,1}$ += Mds$_{0,0}$*Nds$_{0,1}$ + Mds$_{0,1}$*Nds$_{1,1}$	$\mathbf{M_{0,3}}$ $\downarrow$ Mds$_{0,1}$	$\mathbf{N_{2,1}}$ $\downarrow$ Nds$_{0,1}$	PValue$_{0,1}$ += Mds$_{0,0}$*Nds$_{0,1}$ + Mds$_{0,1}$*Nds$_{1,1}$
thread$_{1,0}$	$\mathbf{M_{1,0}}$ $\downarrow$ Mds$_{1,0}$	$\mathbf{N_{1,0}}$ $\downarrow$ Nds$_{1,0}$	PValue$_{1,0}$ += Mds$_{1,0}$*Nds$_{0,0}$ + Mds$_{1,1}$*Nds$_{1,0}$	$\mathbf{M_{1,2}}$ $\downarrow$ Mds$_{1,0}$	$\mathbf{N_{3,0}}$ $\downarrow$ Nds$_{1,0}$	PValue$_{1,0}$ += Mds$_{1,0}$*Nds$_{0,0}$ + Mds$_{1,1}$*Nds$_{1,0}$
thread$_{1,1}$	$\mathbf{M_{1,1}}$ $\downarrow$ Mds$_{1,1}$	$\mathbf{N_{1,1}}$ $\downarrow$ Nds$_{1,1}$	PValue$_{1,1}$ += Mds$_{1,0}$*Nds$_{0,1}$ + Mds$_{1,1}$*Nds$_{1,1}$	$\mathbf{M_{1,3}}$ $\downarrow$ Mds$_{1,1}$	$\mathbf{N_{3,1}}$ $\downarrow$ Nds$_{1,1}$	PValue$_{1,1}$ += Mds$_{1,0}$*Nds$_{0,1}$ + Mds$_{1,1}$*Nds$_{1,1}$

time $\longrightarrow$

Figure: Execution phases of a tiled matrix multiplication

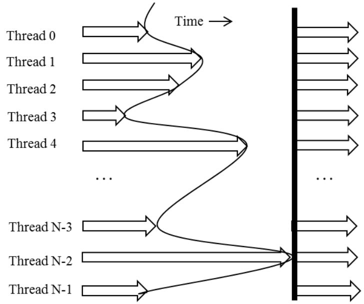
Tiled Matrix Multiplication Contd.

- In general, if an input matrix is of size `N` $\times$ `N` and the tile size is `TILE_WIDTH` $\times$ `TILE_WIDTH`, the multiplication will be performed in `N/TILE_WIDTH` phases.
- The benefit of the tiled algorithm is substantial. For matrix multiplication, the global memory accesses are reduced by a factor of `TILE_WIDTH`. This increases the CGMA (Compute to Global Memory Access) ratio from 1 to `TILE_WIDTH`.

Barrier Synchronization

How to coordinate the execution of the CUDA threads?

- Barrier synchronization function: `__syncthreads();`
- When called, all threads in a block will be held at the calling location until every thread in the block reaches the location.



Outline

WITS UNIVERSITY

Memory as a Limiting Factor to Parallelism

- Cuda registers and shared memory are effective in reducing the number of accesses to global memory.
- Their sizes are limited and they are shared among the thread blocks reside in a SM.
- This also limit the number of threads can reside in each SM.
- An application can dynamically determine these properties of a device.
- Done by calling the `cudaGetDeviceProperties()` function.

Outline

A Simple Running Example 3: Reduction

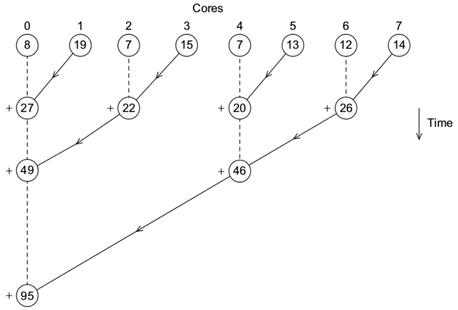


Figure: A tree structured global sum

A Simple Running Example 3: Reduction

Use the following tip to map the threads to the addition of one pair of numbers at each reduction step. Note that not all the threads participate in the additions of every reduction steps.

```
int t = threadIdx.x;
/*declare shared memory for array sum and initialize it here*/

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

A Simple Running Example 3: Reduction

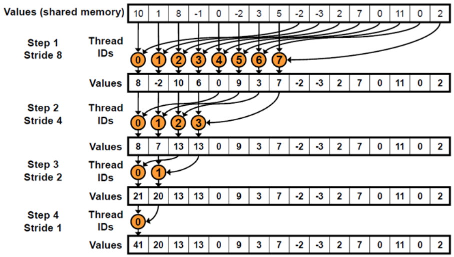


Figure: Another tree structured global sum

A Simple Running Example 3: Reduction

Use the following tip to map the threads to the addition of one pair of numbers at each reduction step. Similar to the previous approach, only a part of the threads participate in the additions at some of the reduction steps.

```
int t = threadIdx.x;
/*declare shared memory for array sum and initialize it here*/

for (int stride=blockDim.x/2;stride>0;stride= stride >> 1) {
  if (t < stride)
    sum[t] += sum[t + stride];
}
```

A Simple Running Example 3: Reduction

- Implement the two approaches for reduction, respectively. For this, you may consider computing a local sum using each block of threads first, then obtain the global sum from these local sums.

References

- CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` – visited April 13, 2025.
- Chapter 4, Programming Massively Parallel Processors: A Hands-on Approach, third edition, by David B. Kirk and Wen-mei W. Hwu. Morgan Kaufmann Publishers Inc.
- Chapter 5, Cuda by Example: an Introduction to General Purpose GPU Programming, by Jason Sanders and Edward Kandrot, Addison-Wesley, 2011.