High Energy Physics

Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

GPU programming module

Week 4 : Introduction to CUDA

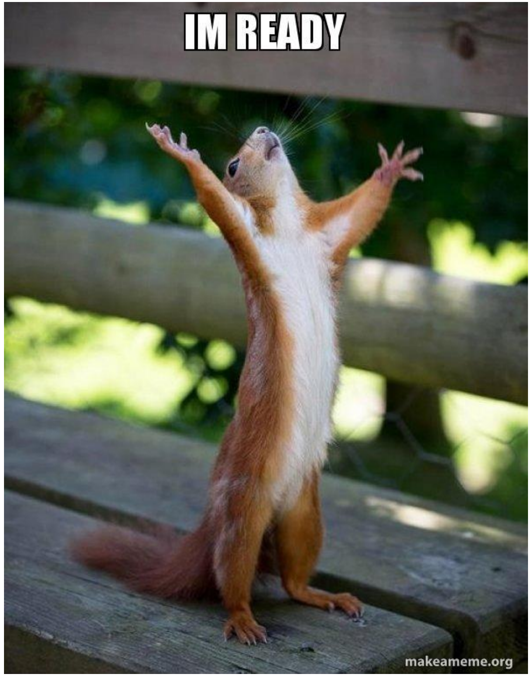
Lecture 7 - October 1st 2024

What we learnt last week

- Learnt about the Nvidia GPU architecture and explored the GPU characteristics
- Learnt about threads / blocks / grid
- Discussed about the CUDA core syntax
- Went over basic memory management
- Learnt how to look out for errors

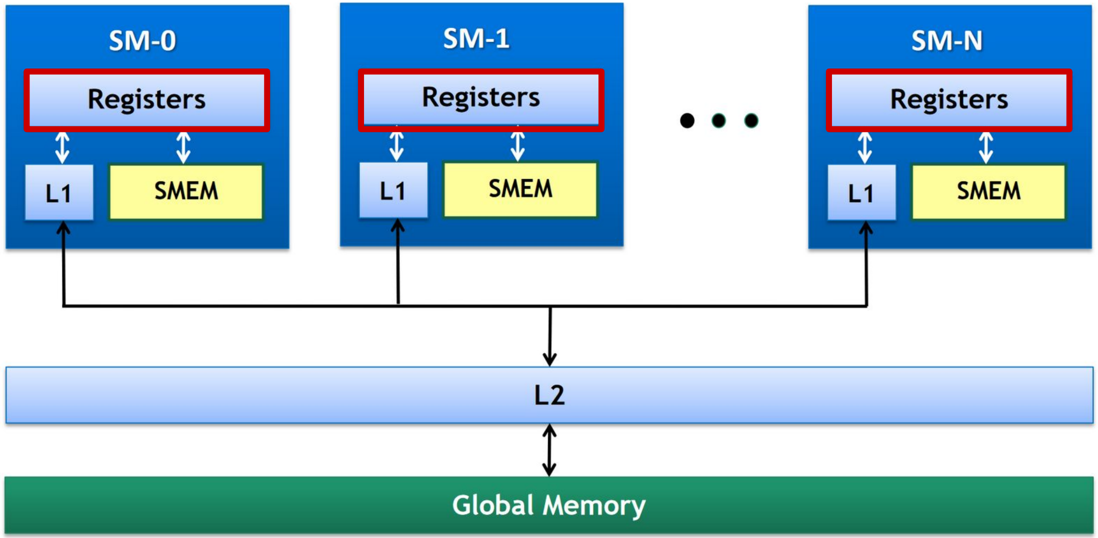
Today

- We will learn more on memory management :
  - Why is data caching important?
  - What is the coalesced memory access pattern?
  - Why is coalesced memory access an important efficiency consideration?



Reminder of memory hierarchy

Reminder of memory hierarchy

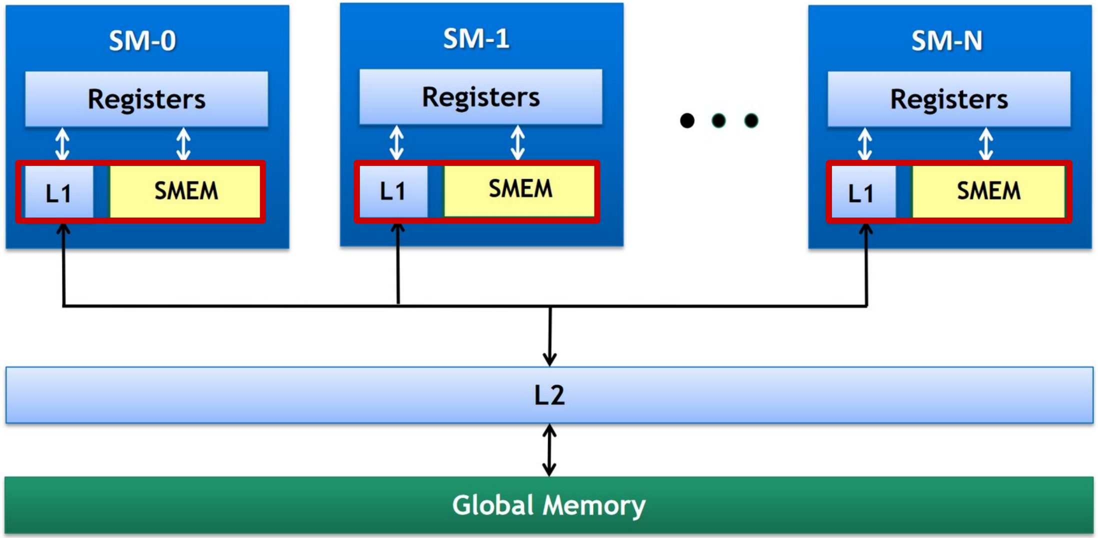


Registers

On-chip
Accesses by a single thread

Image source [1]

Reminder of memory hierarchy



Registers

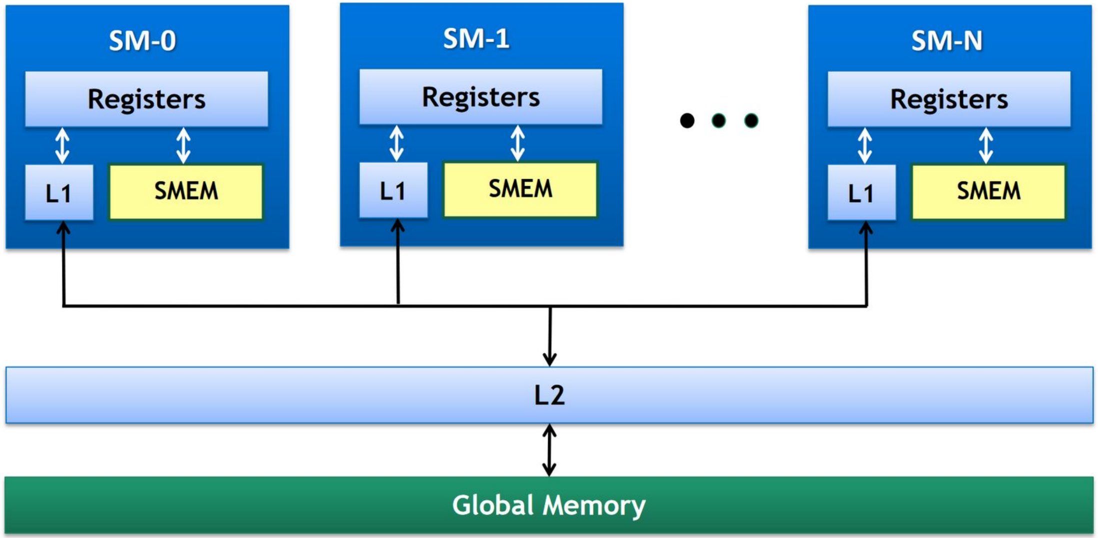
On-chip
Accesses by a single thread

L1 shared

On-chip
Accesses by all threads in block

Image source [1]

Reminder of memory hierarchy



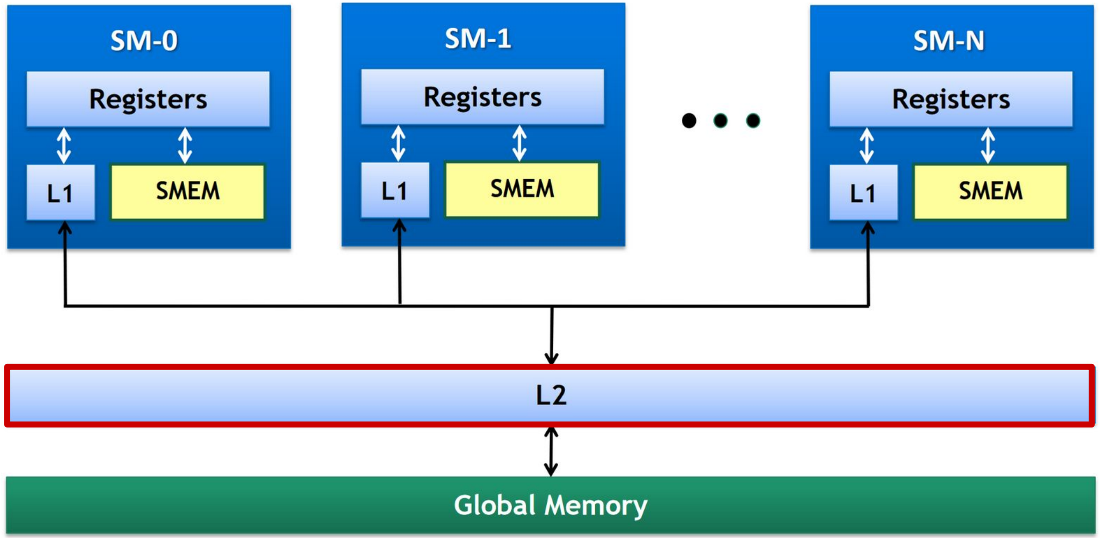
Registers
On-chip
Accesses by a single thread

L1 shared
On-chip
Accesses by all threads in block

Constant/ texture
Off-chip
Read-only

Image source [1]

Reminder of memory hierarchy



Registers
On-chip
Accesses by a single thread

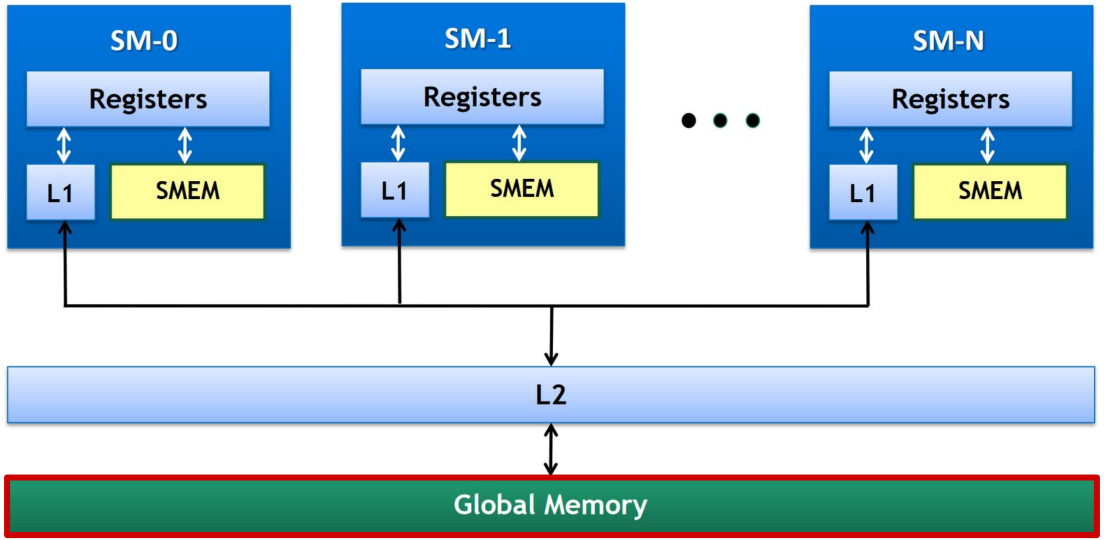
L1 shared
On-chip
Accesses by all threads in block

Constant/ texture
Off-chip
Read-only

L2 Cache
Off-chip
Accesses by all threads all blocks

Image source [1]

Reminder of memory hierarchy



Registers
On-chip
Accesses by a single thread

L1 shared
On-chip
Accesses by all threads in block

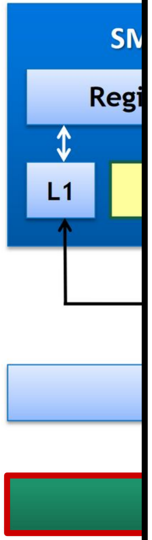
Constant/ texture
Off-chip
Read-only

L2 Cache
Off-chip
Accesses by all threads all blocks

Global
DRAM / large
Accessed by device & host

Image source [1]

Rem



Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.					

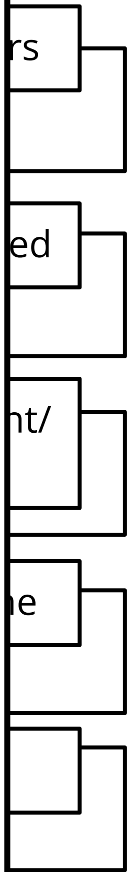


Image source [7]

Global memory and data caching

Global memory

- Accessible by all GPU threads
- Location where memory allocated with cudaMalloc() comes from.
- Has high latency
  - It takes a relatively long time for data to be loaded into registers
  - Can be a performance limiter

Global memory and data caching

Global memory

- Accessible by all GPU threads
- Location where memory allocated with cudaMalloc() comes from.
- Has high latency
  - It takes a relatively long time for data to be loaded into registers
  - Can be a performance limiter

Caching Data

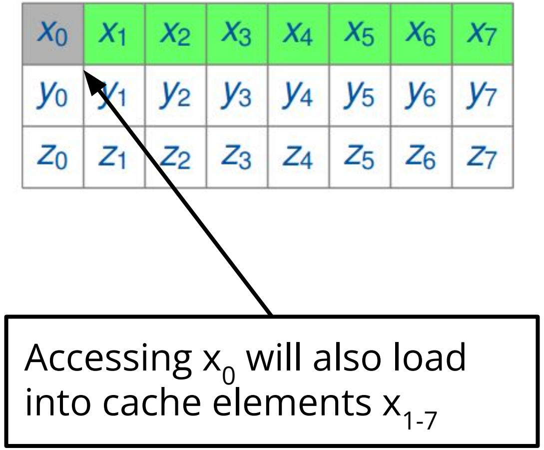
- Process that stores multiple copies of data or files in a temporary storage location
- Future requests for that data are served up faster compared to accessing the primary storage location.
- Caching allows you to efficiently reuse previously retrieved or computed data

Data locality

Data locality : Computation is performed where the data resides

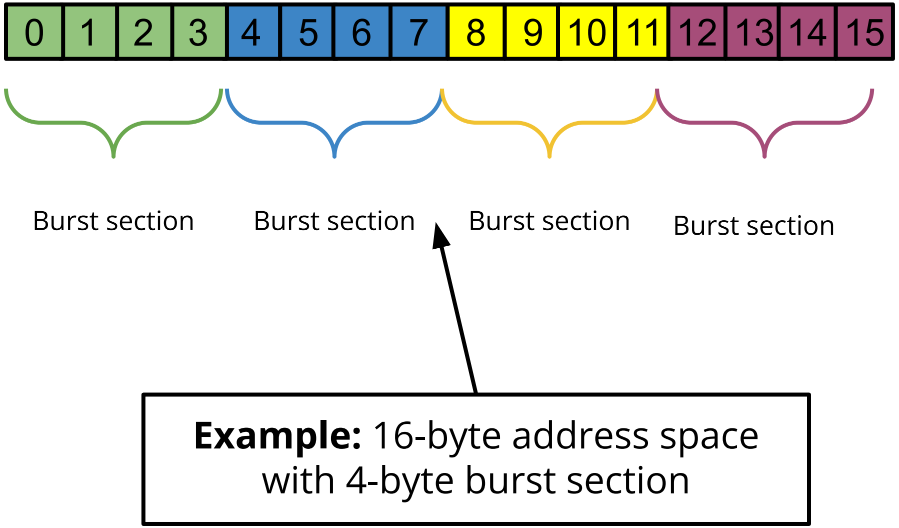
<u>Two types of data locality :</u>

- **Spatial locality**
  - If a program accesses one memory address, neighbouring memory locations likely to be accessed
- **Temporal locality**
  - If a program accesses one memory address, the same memory locations likely to be accessed



Data locality and DRAM burst

- The devices DRAM is organized in **burst sections**
  - Successive bytes that can be accessed simultaneously
  - These are read into cache memory
- Typical burst section is 128 bytes



Coalesced memory access

Coalesced access to global memory

- **Threads in a warp execute the same instruction at any given point in time.**
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations.
  - Global memory loads and stores data in as few as possible transactions

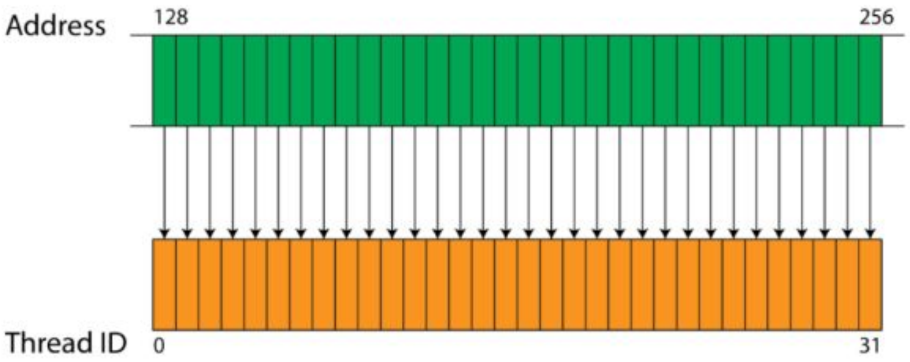
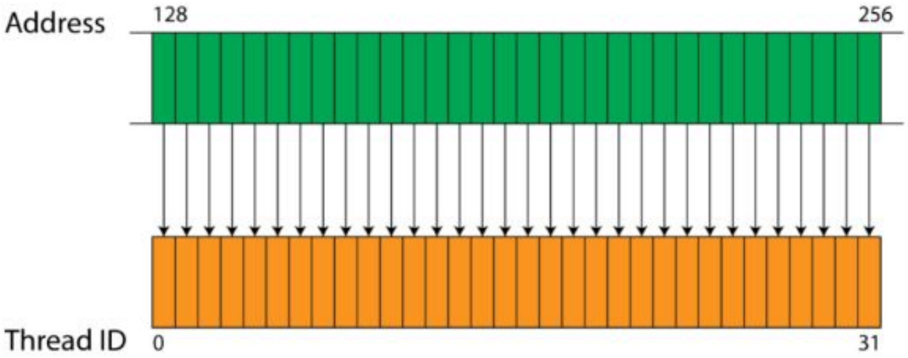


Image source [1]

Coalesced access to global memory

- When threads make a memory request and the request falls under the same burst, **the access is coalesced**
- Important performance consideration as it can affect the time needed to access data



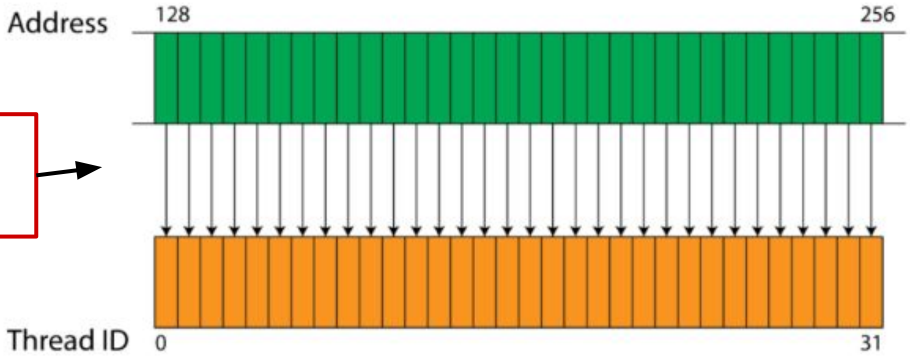
Every successive 128 bytes (DRAM burst) can be accessed by a warp

Image source [i]

Coalesced access to global memory

- When threads make a memory request and the request falls under the same burst, **the access is coalesced**
- Important performance consideration as it can affect the time needed to access data

Single transaction



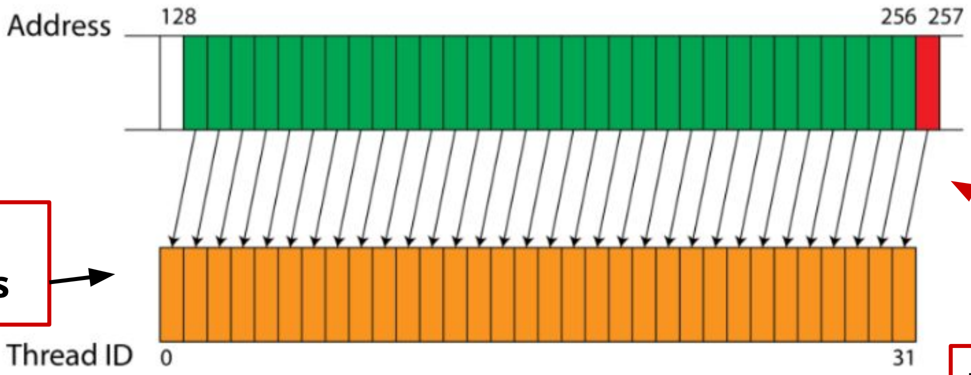
Every successive 128 bytes (DRAM burst) can be accessed by a warp

Image source [i]

Coalesced access to global memory

- If the data accessed by the threads in a warp are not in the same burst section, the data access will take twice as long

Two transactions



This is an undesired behaviour, which impacts performance.

Image source [i]

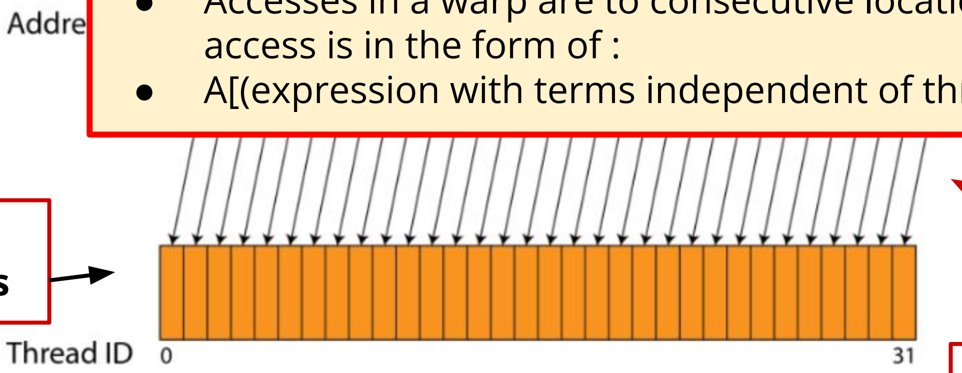
Coalesced access to global memory

- If the data accessed by the threads in a warp are not in the same burst section, the data access will take twice as long

How can we conclude that an access patten 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]

Two transactions



This is an undesired behaviour, which impacts performance.

Image source [i]

Linear representation of a matrix

A(0,0)	**A(0,1)**	**A(0,2)**	**A(0,3)**
A(1,0)	**A(1,1)**	**A(1,2)**	**A(1,3)**
A(2,0)	**A(2,1)**	**A(2,2)**	**A(2,3)**
A(3,0)	**A(3,1)**	**A(3,2)**	**A(3,3)**

A[row][column] → A[row,column]

Linear representation of a matrix

A(0,0)	**A(0,1)**	**A(0,2)**	**A(0,3)**
A(1,0)	**A(1,1)**	**A(1,2)**	**A(1,3)**
A(2,0)	**A(2,1)**	**A(2,2)**	**A(2,3)**
A(3,0)	**A(3,1)**	**A(3,2)**	**A(3,3)**

A(0,0)	**A(0,1)**	**A(0,2)**	**…**	**A(3,3)**

- **Row major order**
- Matrix represented in 1-D by concatenating one row after the other :
- If $size_A$ = rows*columns :
    - A(i,j) = i*columns+j

Linear representation of a matrix

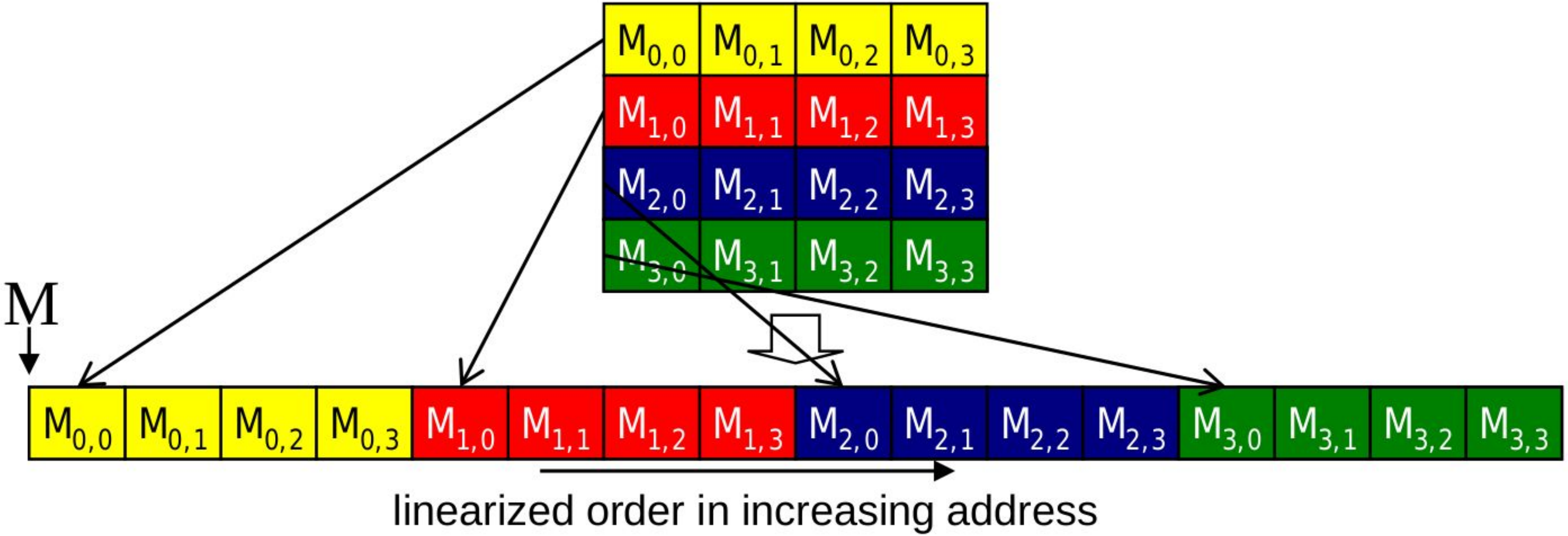


Image source [7]

Linear representation of a matrix

A(0,0)	**A(0,1)**	**A(0,2)**	**A(0,3)**
A(1,0)	**A(1,1)**	**A(1,2)**	**A(1,3)**
A(3,0)	**A(3,1)**	**A(3,2)**	**A(3,3)**

- Default way 2-d arrays are stored in C/C++
- Lets try out this script to check the memory location of the matrix elements!

A(0,0)	**A(0,1)**	**A(0,2)**	**…**	**A(3,3)**

Row major order
Matrix represented in 1-D by concatenating one row after the other :

- If $size_A$ = rows*columns :
  - A(i,j) = i*columns+j

Linear representation of a matrix

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)
A(3,0)	A(3,1)	A(3,2)	A(3,3)

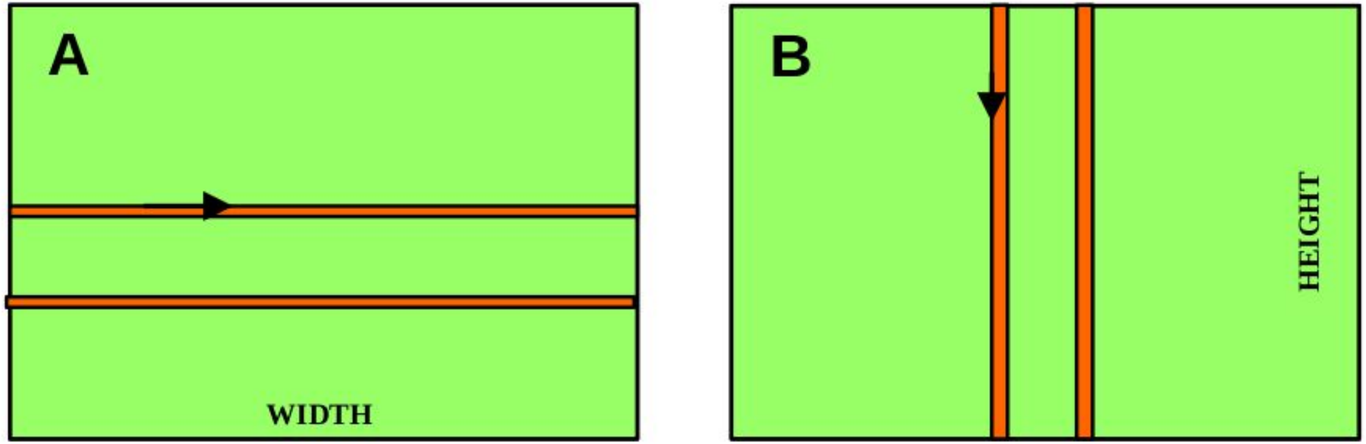
A(0,0)	A(1,0)	A(2,0)	…	A(3,3)

- **Column major order**
- Matrix represented in 1-D by concatenating one column after the other :
- If $size_A$ = rows*columns :
    - A(i,j) = j*columns+i

Example - Matrix multiplication

Thread 1

Thread 2

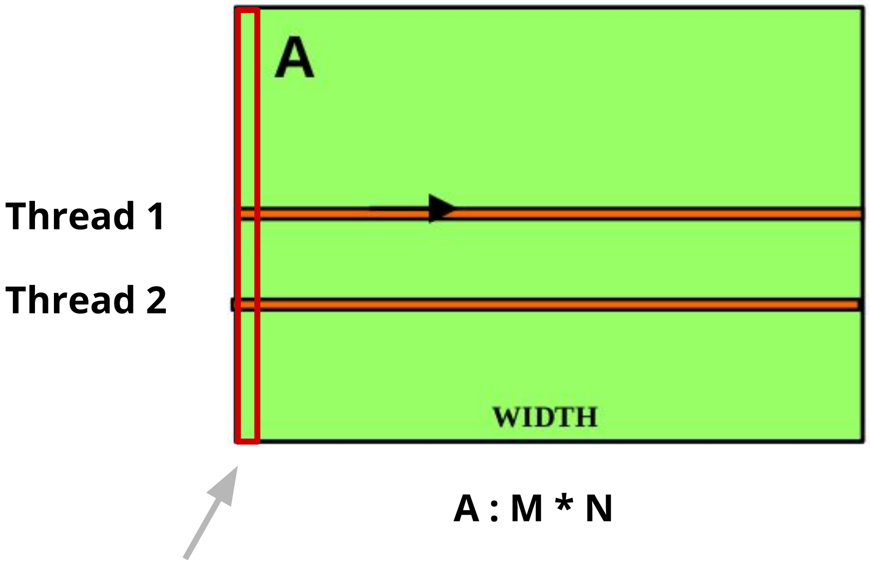


A : M * N

B: N * K

Image source [8]

Example - Matrix multiplication



A : M * N

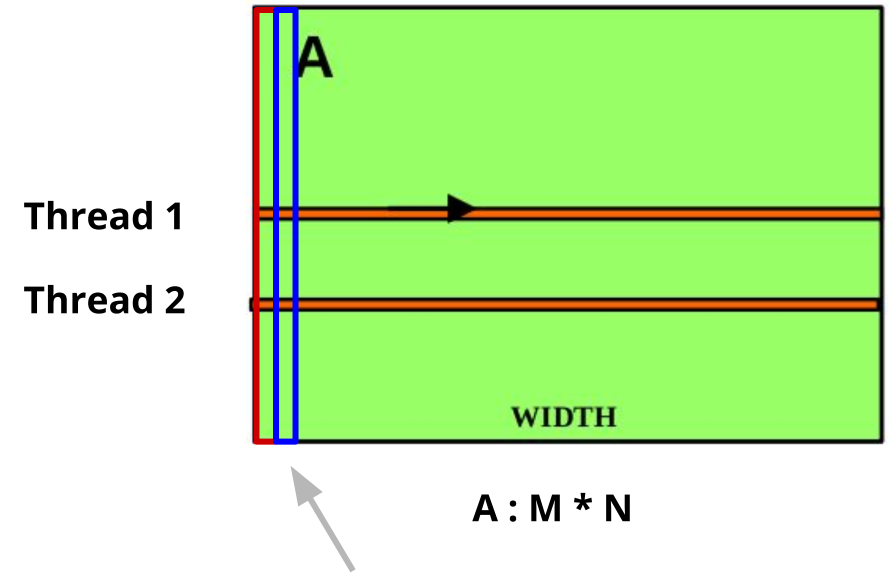
First iteration → a warp of 32 threads reads element 0 of the first 32 rows

Matrix A has an unfavorable data access pattern :

- Threads in a warp read adjacent rows
- During the first iteration, threads in a warp read element 0 of rows 0 through 31.

Image source [8]

Example - Matrix multiplication



Second iteration → the same warp of 32 threads reads element 1 of the first 32 rows

Matrix A has an unfavorable data access pattern :

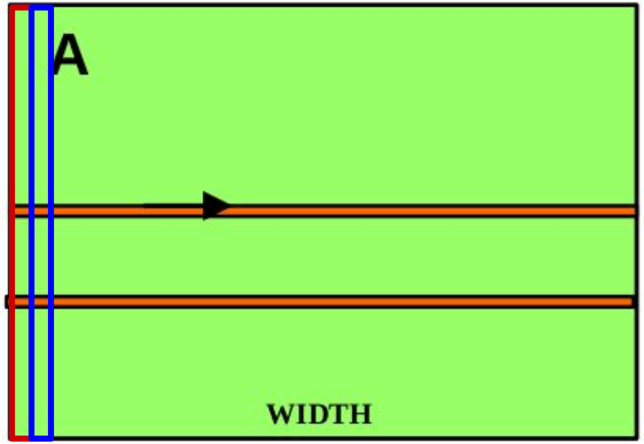
- Threads in a warp read adjacent rows
- During the first iteration, threads in a warp read element 0 of rows 0 through 31.
- During the second iteration the same set of threads read element 1 of rows 0 through 31.

Image source [8]

Example - Matrix multiplication

Thread 1

Thread 2



A : M * N

Matrix A has an unfavorable data access pattern :

- Threads in a warp read adjacent rows
- During the first iteration, threads in a warp read element 0 of rows 0 through 31.
- During the second iteration the same set of threads read element 1 of rows 0 through 31.

None of the accesses will be coalesced!!

Image source [8]

Example - Matrix multiplication

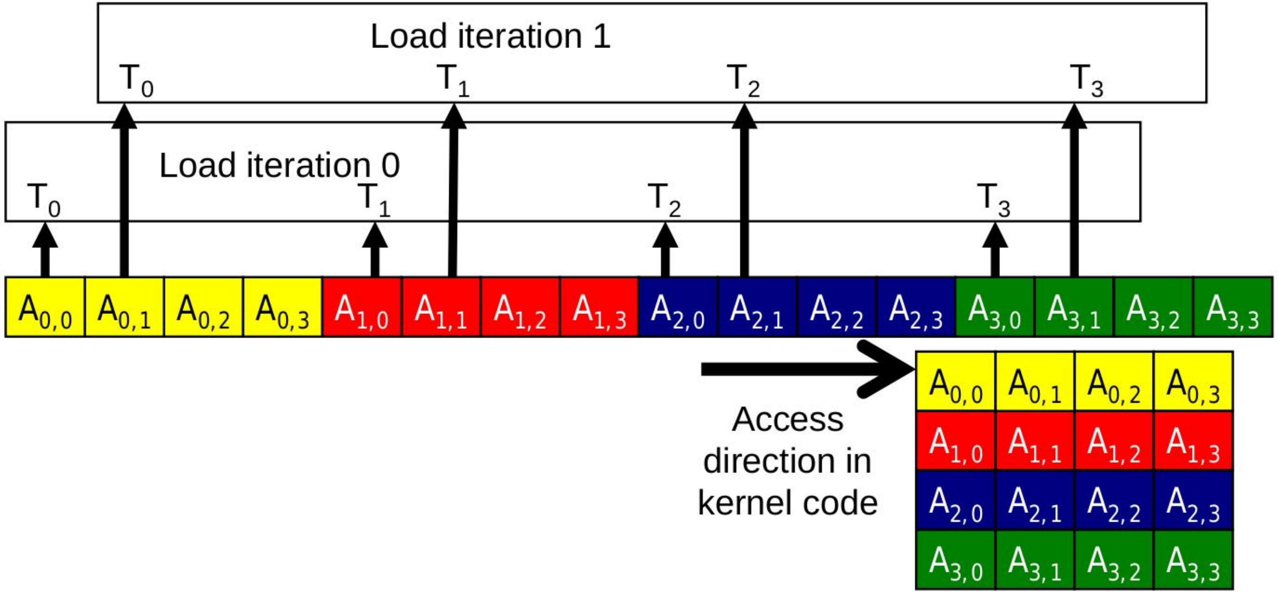
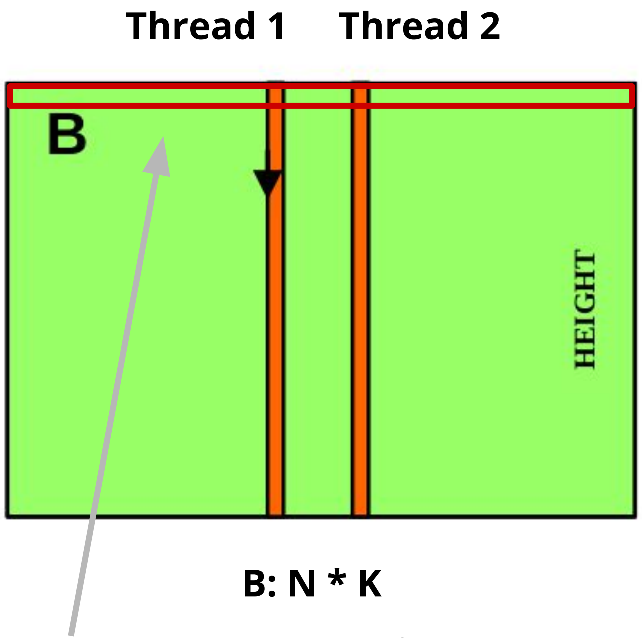


Image source [8]

Example - Matrix multiplication



B: N * K

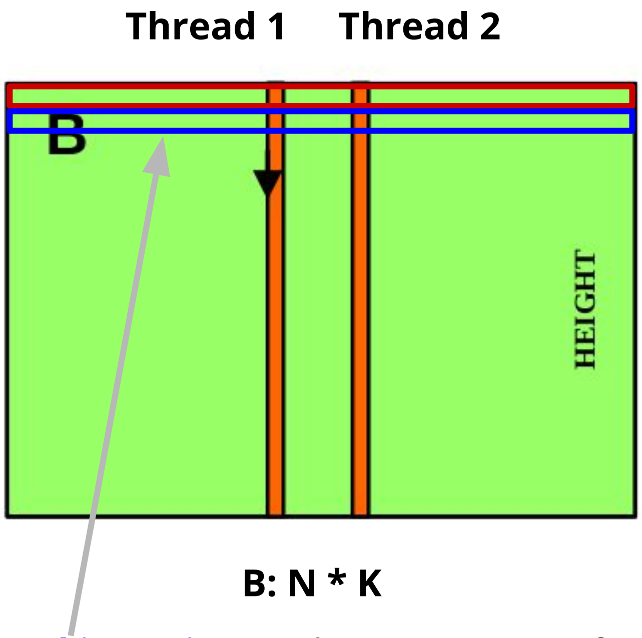
Matrix B has a favorable data access pattern :

- Each thread reads a column of N elements
- During the first iteration, threads in a warp read element 0 of columns 0 to 31

First iteration → a warp of 32 threads reads element 0 of the first 32 columns

Image source [8]

Example - Matrix multiplication



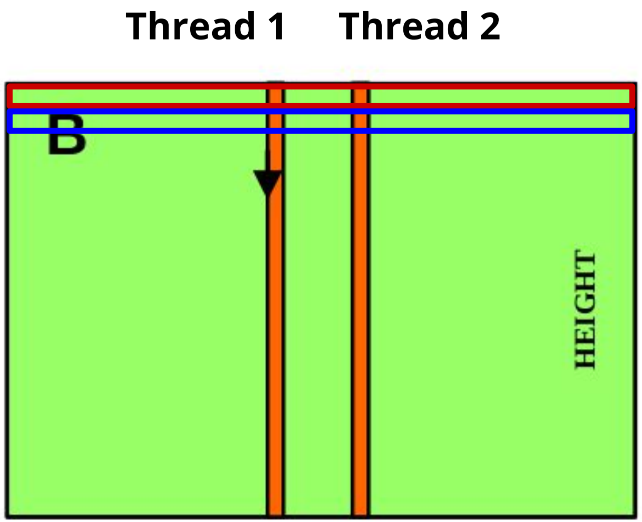
Matrix B has a favorable data access pattern :

- Each thread reads a column of N elements
- During the first iteration, threads in a warp read element 0 of columns 0 to 31
- During the second iteration, threads in a warp read element 1 of columns 0 to 31

Second iteration → the same warp of 32 threads reads element 1 of the first 32 columns

Image source [8]

Example - Matrix multiplication



B: N * K

Matrix B has a favorable data access pattern :

- Each thread reads a column of N elements
- During the first iteration, threads in a warp read element 0 of columns 0 to 31
- During the second iteration, threads in a warp read element 1 of columns 0 to 31

These elements are stored in the same burst section & these accesses will be coalesced!

Image source [8]

Example - Matrix multiplication

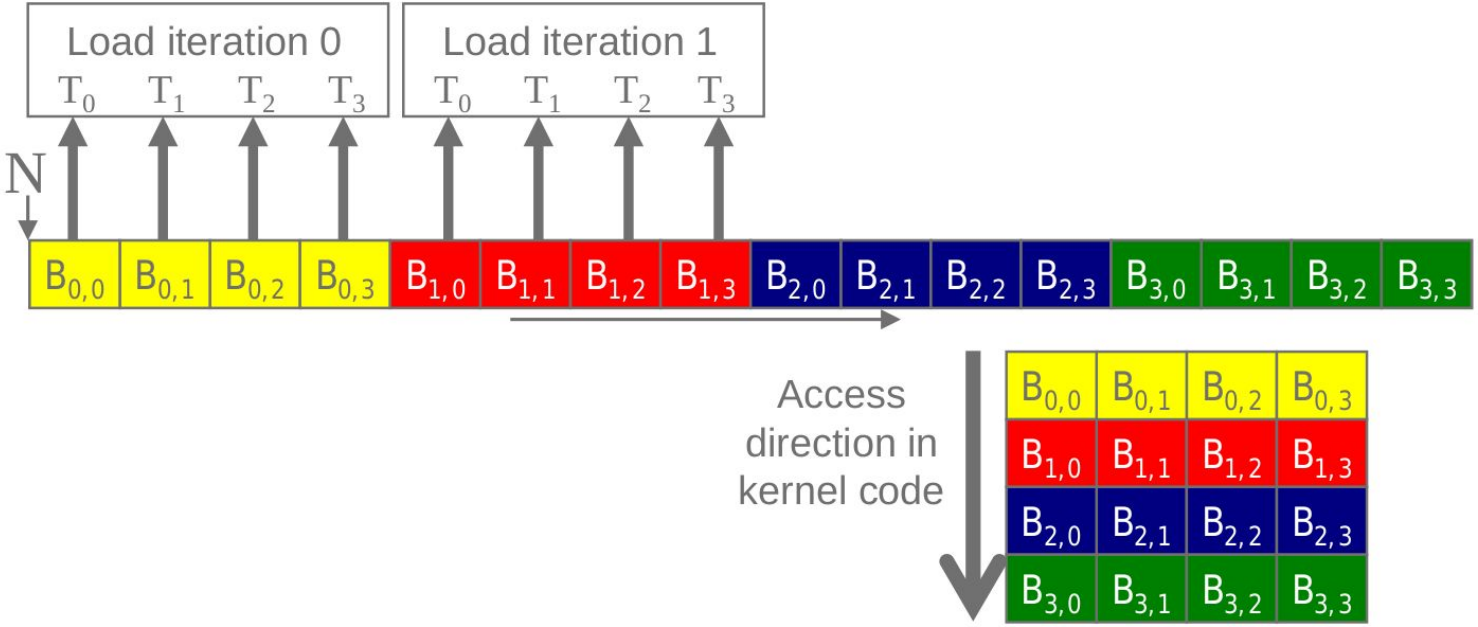


Image source [8]

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

<u>Questions :</u>

- <u>Is memory access of elements of matrix A coalesced?</u>

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

- Is memory access of elements of matrix A coalesced?

REMEMBER

How can we conclude that an access patten 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]

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

NO: row*N+k = blockIdx.y * blockDim.y * N + threadIdx.y *N + k

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int column = blockIdx.x * blockDim.x + threadIdx.x;

   if((row < N) && (column < N)){
       float sum = 0;
       for(int k = 0; k < N; k++) {
           sum += A[row*N + k] * B[k*N + column];
       }
       C[row * N + column] = sum;
   }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

- Is memory access of elements of matrix A coalesced?
- Is memory access of elements of matrix B coalesced?

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int column = blockIdx.x * blockDim.x + threadIdx.x;

   if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
   }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

- Is memory access of elements of matrix A coalesced?

YES: k*N+column = k*N+blockIdx.x*blockDim.x+**threadIdx.x**

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

- Is memory access of elements of matrix A coalesced?
- Is memory access of elements of matrix B coalesced?
- Is memory access of elements of matrix C coalesced?

Example - Matrix multiplication

```
__global__ void matrix_mult (float* A, float* B, float*
C, int N){

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int column = blockIdx.x * blockDim.x + threadIdx.x;

    if((row < N) && (column < N)){
        float sum = 0;
        for(int k = 0; k < N; k++) {
            sum += A[row*N + k] * B[k*N + column];
        }
        C[row * N + column] = sum;
    }
}
```

Let's take a look at this kernel that performs matrix multiplication of two matrices.

Questions :

- Is memory access of elements of matrix A coalesced?
- Is memory access of elements of matrix B coalesced?

YES: row*N+column = N*blockIdx.y * blockDim.y + N*threadIdx.y + blockIdx.x * blockDim.x + **threadIdx.x**

Wrapping-up

Overview of today's lecture

- Today we went deeper into memory management with CUDA
    - Discussed about data locality and caching
    - Understood the coalesced memory data access pattern

Overview of today's lecture

- Today we went deeper into memory management with CUDA
  - Discussed about data locality and caching
  - Understood the coalesced memory data access pattern

Let's take 5 mins to fill-in this mid-training survey!

Tomorrow

We will learn about :

- Shared memory
- Atomic operations
- The default CUDA stream

Back-up

Resources

1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link
7. CUDA/C++ best practices guide link
8. NVidia DLI teaching kit link