CENG443
Heterogeneous Parallel Programming

CUDA Kernel Performance

Işıl ÖZ, IZTECH, Fall 2023

24 November 2023

Computing Performance

The computing performance of a CPU or a GPU is usually measured in Flops

$$Floprate = \frac{\text{number of floating-point operations [Flop]}}{\text{time [s]}}$$

Usually the number of floating-point additions and multiplications the hardware can perform per second

Theoretical Peak Floprate

Quad-core Intel Skylake CPU:

$\sim$ 200 GFlops

14-core Intel Xeon Gold 6132 CPU:

$\sim$ 1200 GFlops

Nvidia Tesla V100 GPU:

$\sim$ **7 000** GFlops

AX Example

```
__global__ void ax_kernel ( int n , double alpha , double * x )
{
  int thread_id = blockIdx.x * blockDim.x + threadIdx.x ;
  int thread_count = gridDim.x * blockDim.x ;
  for ( int i = thread_id ; i < n ; i += thread_count )
    x [i] = alpha * x [i];
}
```

AX Example – Actual Floprate

Nvidia Tesla V100 GPU (~ 7 000 GFlops):

Time = 0.010582 s

Floprate = 47 Gflops

Less than 1% of the peak floprate

Memory Performance

The memory performance of a CPU or a GPU is usually measured in terms of memory throughput

$$\text{Throughput} = \frac{\text{number of bytes moved [Byte]}}{\text{time [s]}}$$

Usually the bandwidth is measured between the CPU cores and the main memory; or the CUDA cores and the global memory

Theoretical Memory Bandwidth

Quad-core Intel Skylake CPU:

$\sim$ 35 GB/s

14-core Intel Xeon Gold 6132 CPU:

$\sim$ 100 GB/s

Nvidia Tesla V100 GPU:

$\sim$ **900** GB/s

AX Example – Actual Memory Throughput

Nvidia Tesla V100 GPU (~ 900 GB/s):

Floprate = 47 GFlops

Memory throughput = 756 GB/s

84% of the memory bandwidth

AX Example - Performance

CUDA cores are idling but the memory bus is busy

--> Memory-bound

GEMM Example

```
void gemm (int n, int ldA, int ldB, int ldC, double *A, double *B, double *C)
{
   for ( int i = 0; i < n ; i ++) {// columns
      for ( int j = 0; j < n ; j ++) {// rows
         double dot = 0.0;
         for ( int k = 0; k < n ; k ++)
            dot += A [k * ldA + j] * B [i * ldB + k];
         C [i * ldC + j] = dot ;
      }
   }
}
```

CublasDgemm in CUDA

GEMM Example – Actual Floprate

Nvidia Tesla V100 GPU (~ 7 000 GFlops):

Floprate was 6077 Gflops

87% of the peak floprate

CUDA cores are busy but the memory bus is partly idle

--> Compute-bound

Arithmetical Intensity

$$\text{Arithmetical intensity} = \frac{\text{number of floating-point operations [Flop]}}{\text{number of bytes moved [Byte]}}$$

Arithmetical Intensity - Example

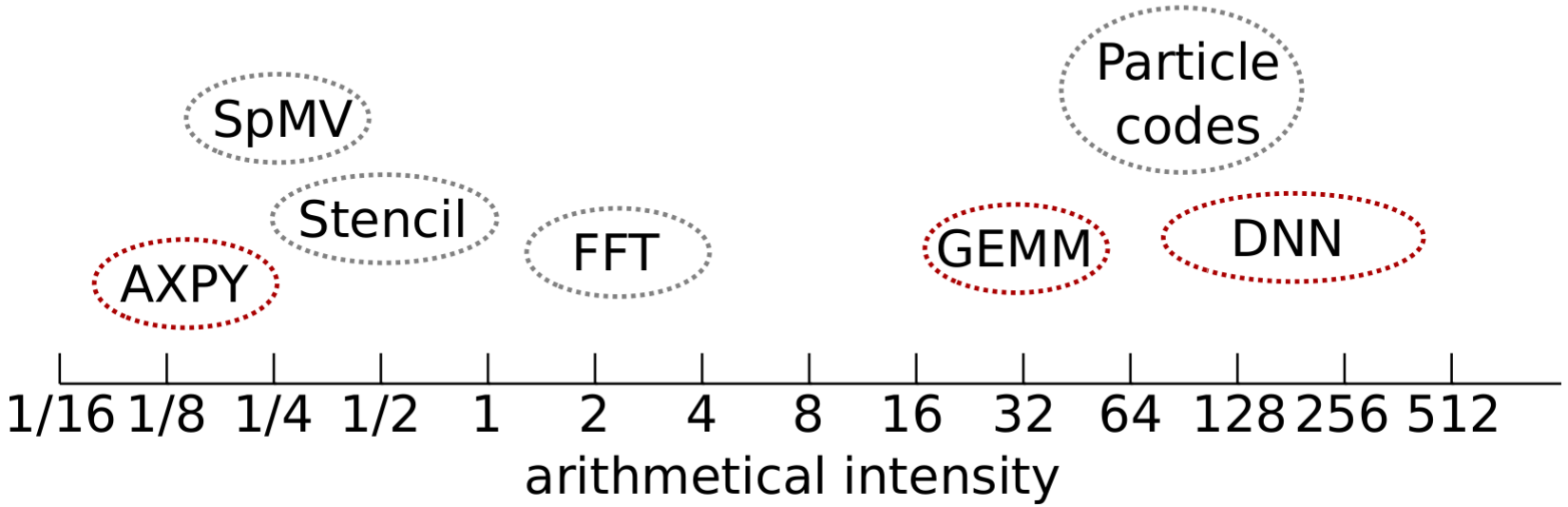
Double precision AX has the arithmetical intensity

$$\frac{1 \text{ Flop}}{2 \cdot 8 \text{ Byte}} = \frac{1}{16} \text{ Flop/Byte}$$

Well-implemented double-precision GEMM has the arithmetical intensity

$$\sim 32 \text{ Flop/Byte}$$

Arithmetical Intensity - Example



Performance Bottlenecks

Managing the interaction between parallel code and hardware resource constraints

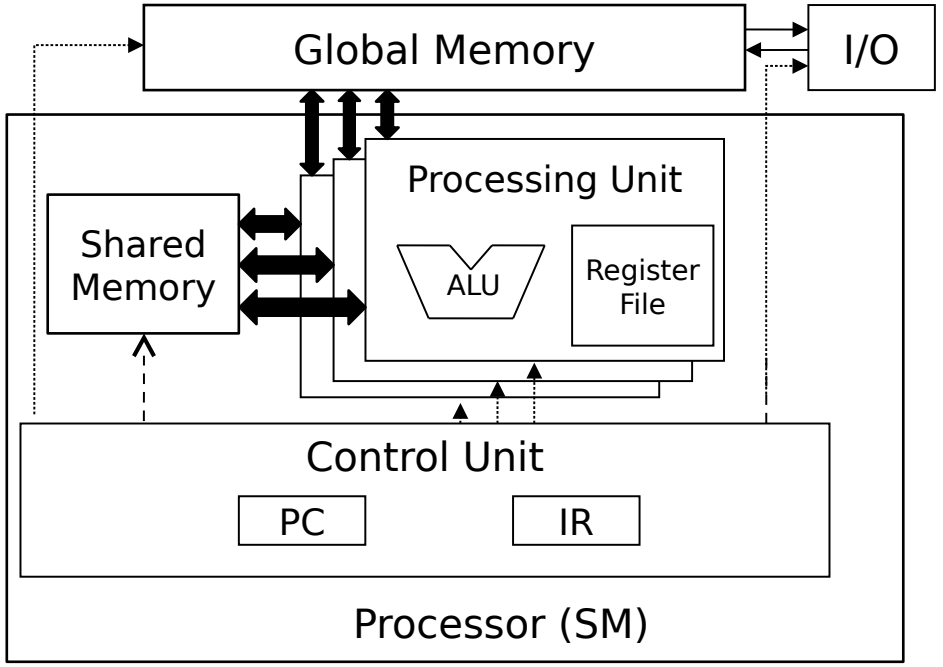
In different applications, different constraints may dominate and become the limiting factors, commonly referred to as bottlenecks

CUDA Kernel Performance

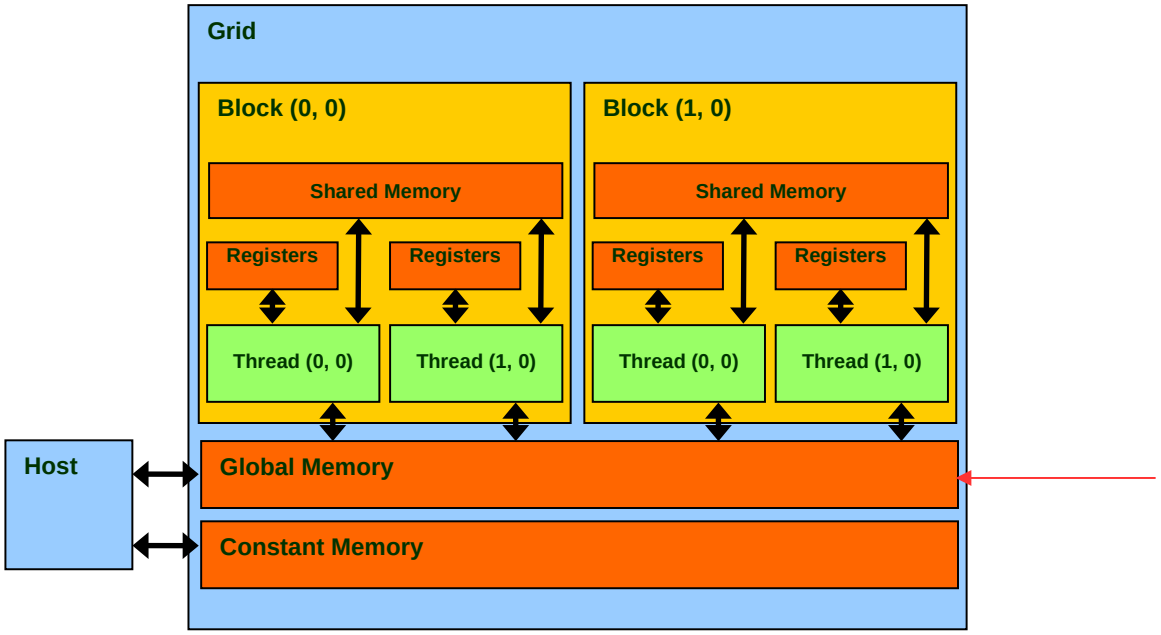
Global memory bandwidth

Warp scheduling

Hardware View of CUDA Memories



Global Memory



- Typically implemented in DRAM
- High access latency:
  400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/s

Global Memory

GPU device with the global memory bandwidth 1,000 GB/s, or 1 TB/s

4 bytes in each single-precision floating-point value

1000/4=250 giga single-precision operands per second to be loaded

- no more than 250 giga floating-point operations per second (GFLOPS)

Peak single-precision performance of 12 TFLOPS

Tiny fraction = 250/12000 = 2%

Need to find ways of reducing global memory accesses!

DRAM Burst

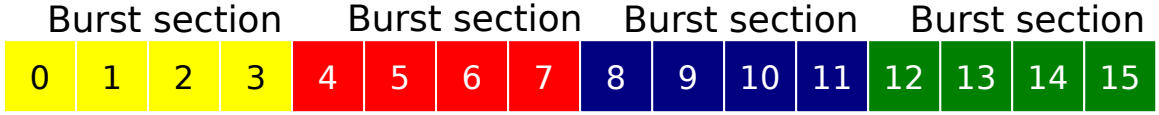
Each time a DRAM location is accessed, a range of consecutive locations that includes the requested location are actually accessed

The data from all these consecutive locations can be transferred at very high-speed to the processor

These consecutive locations accessed and delivered are referred to as DRAM *bursts*

If an application makes focused use of data from these bursts, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed

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 processor

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

Memory Coalescing

Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns

Takes advantage of the fact that threads in a warp execute the same instruction at any given point in time

Memory Coalescing

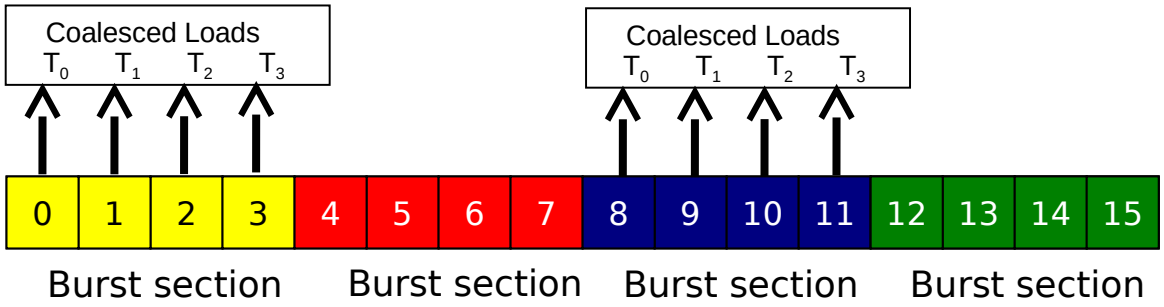
When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations

The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations

The hardware combines, or *coalesces*, all these accesses into a consolidated access to consecutive DRAM locations

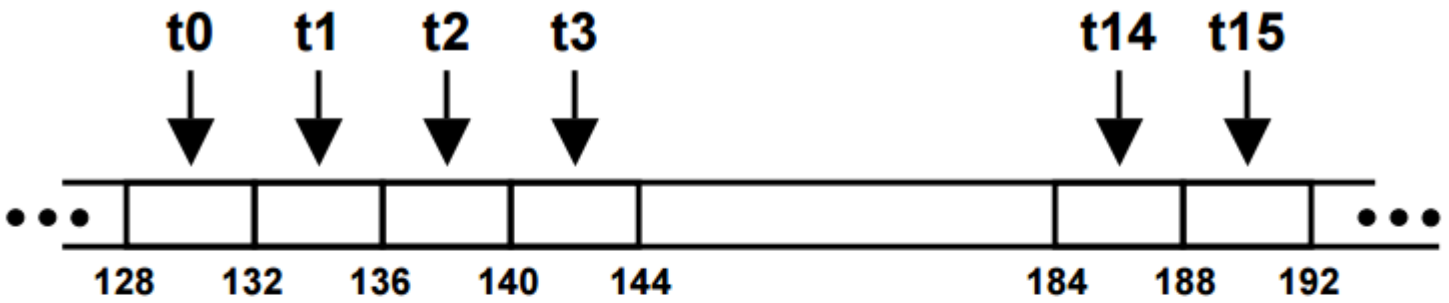
Such coalesced access allows the DRAMs to deliver data as a burst

Memory Coalescing

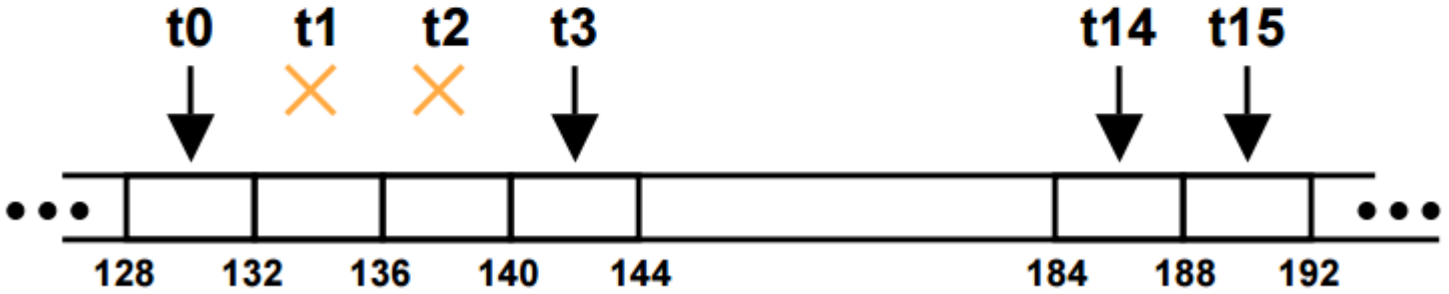


When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced

Coalesced Accesses

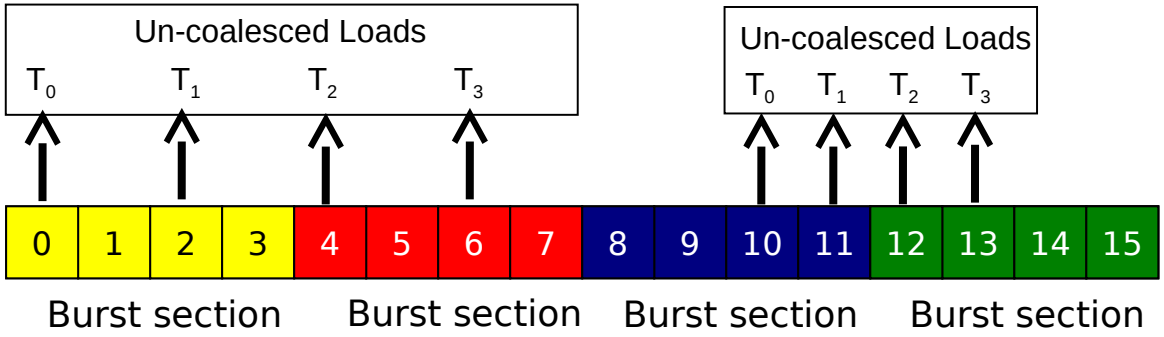


All threads participate



Some Threads Do Not Participate

Uncoalesced Accesses



When the accessed locations spread across burst section boundaries:

- Coalescing fails
- Multiple DRAM requests are made
- The access is not fully coalesced

Some of the bytes accessed and transferred are not used by the threads

Warp Coalescence

Accesses are to consecutive locations if the index in an array access is in the form

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

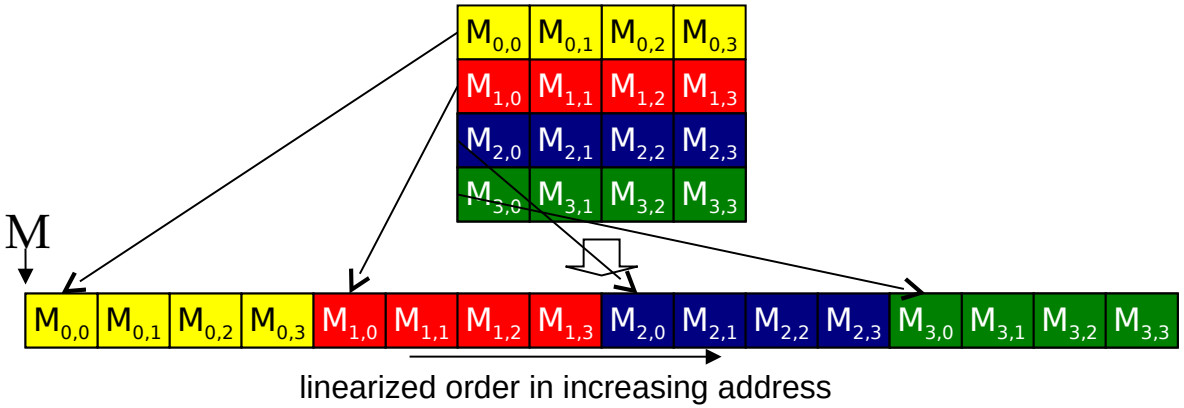
T0 → A[0]

T1 → A[1]

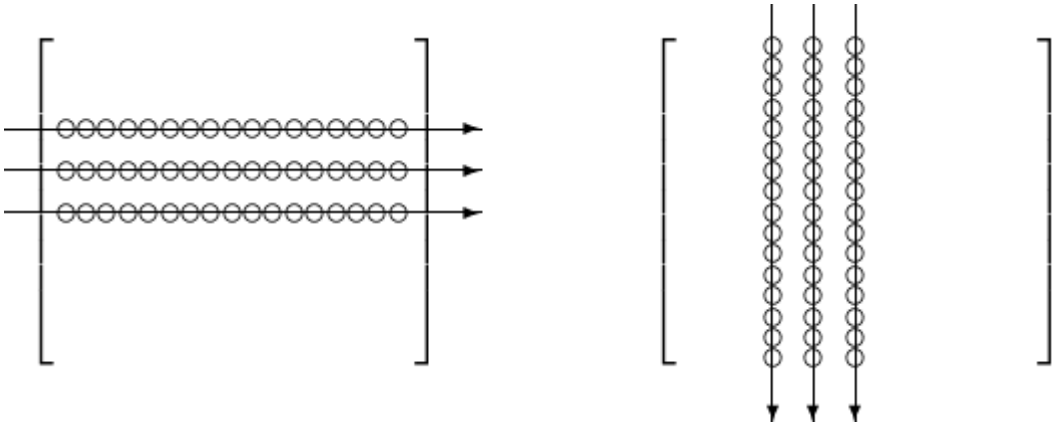
T2 → A[2]

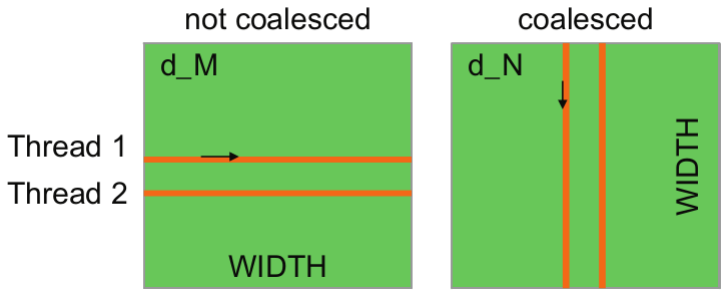
…

A 2D C Array in Linear Memory Space



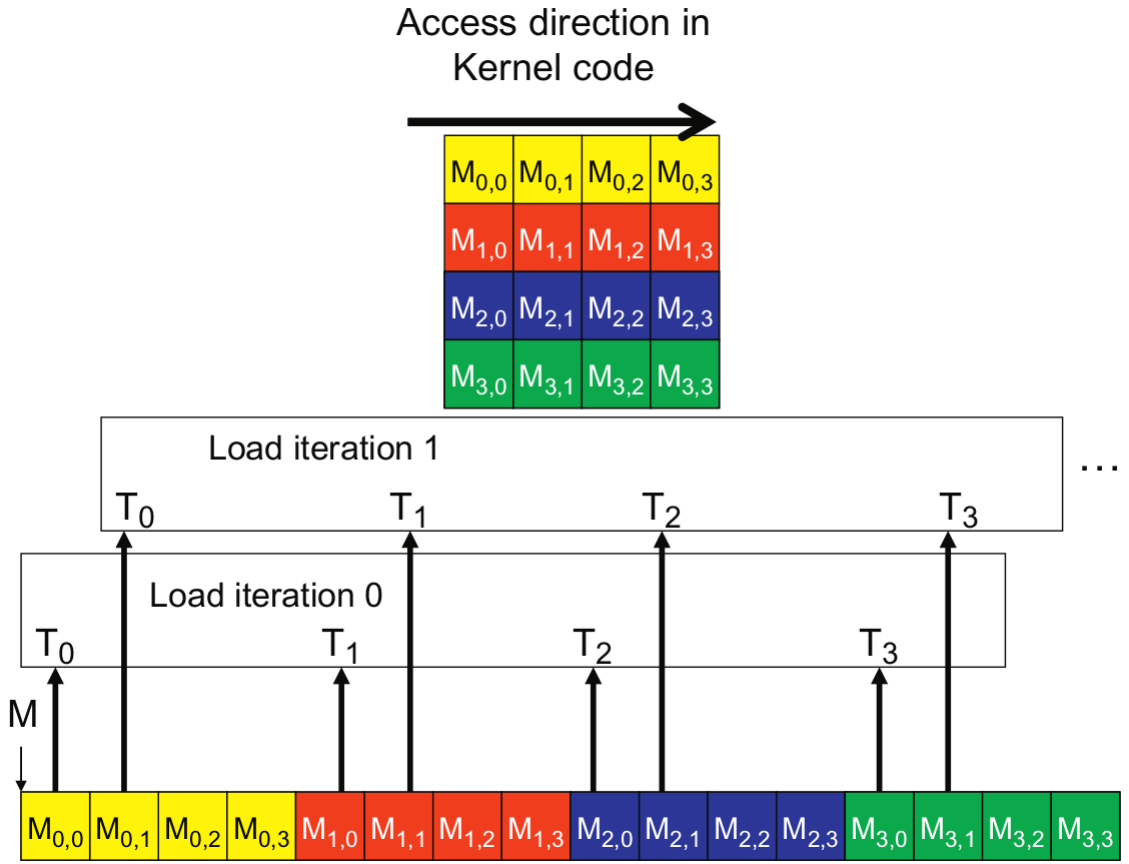
Accessing Elements in a Matrix





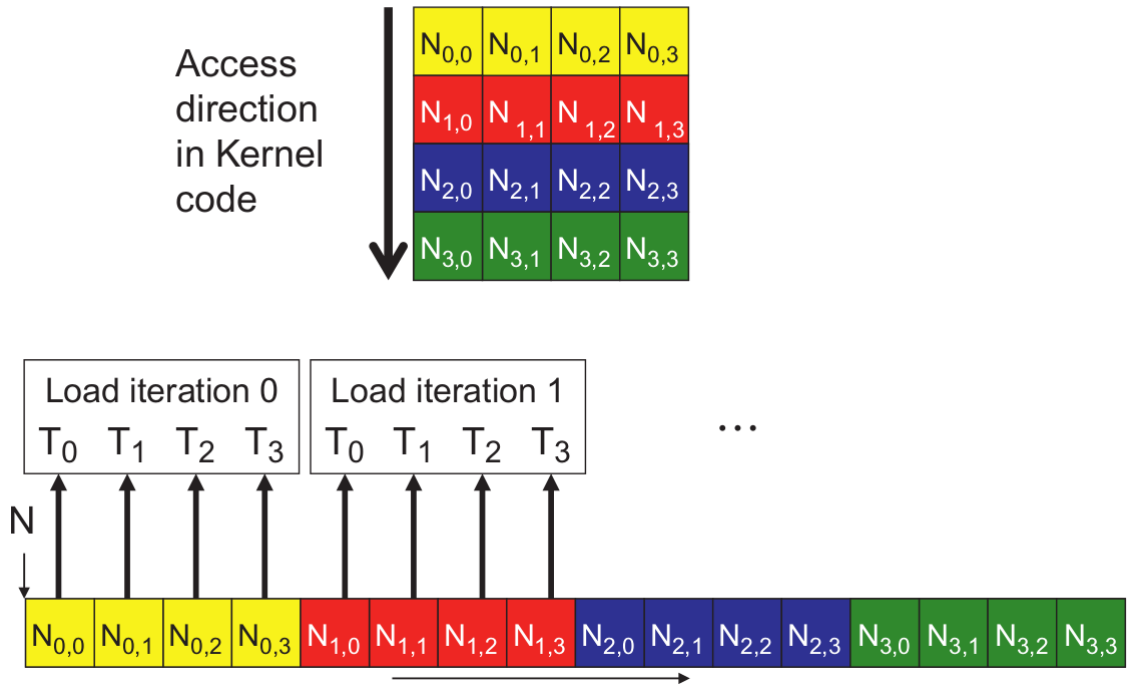
Accesses are not Coalesced

Threads T_0,T_1,T_2, and T_3 access the elements on the first two columns



Accesses are Coalesced

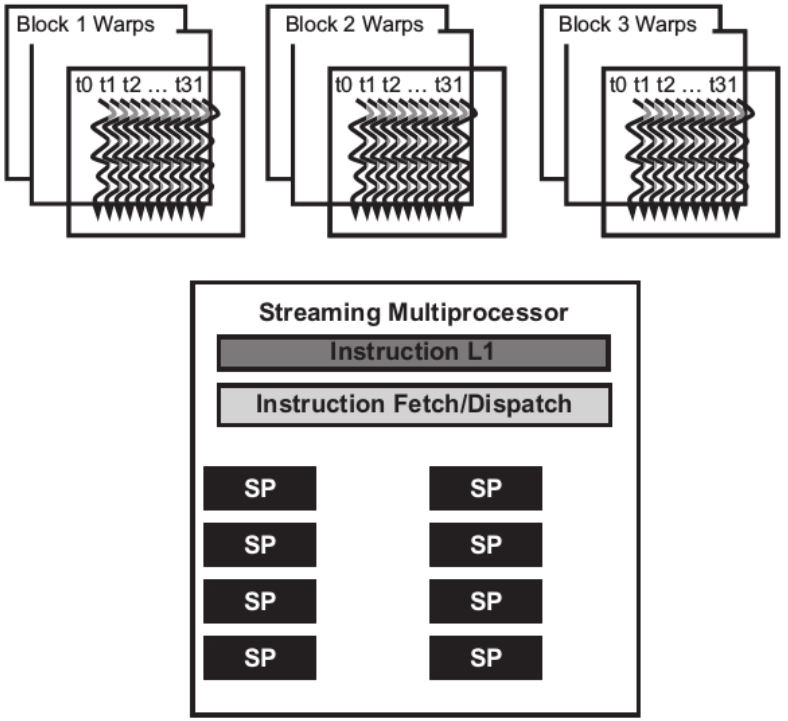
Threads T_0,T_1,T_2, and T_3 access the elements on the first two rows

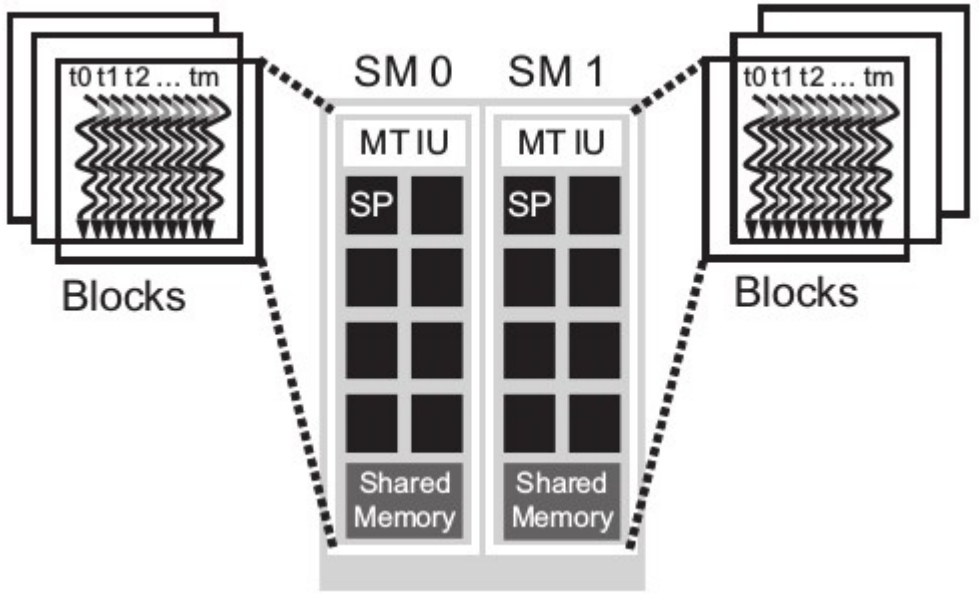


Warps

Each block is divided into 32-thread warps

Warps are scheduling units in SM





Block Partitioning into Warps

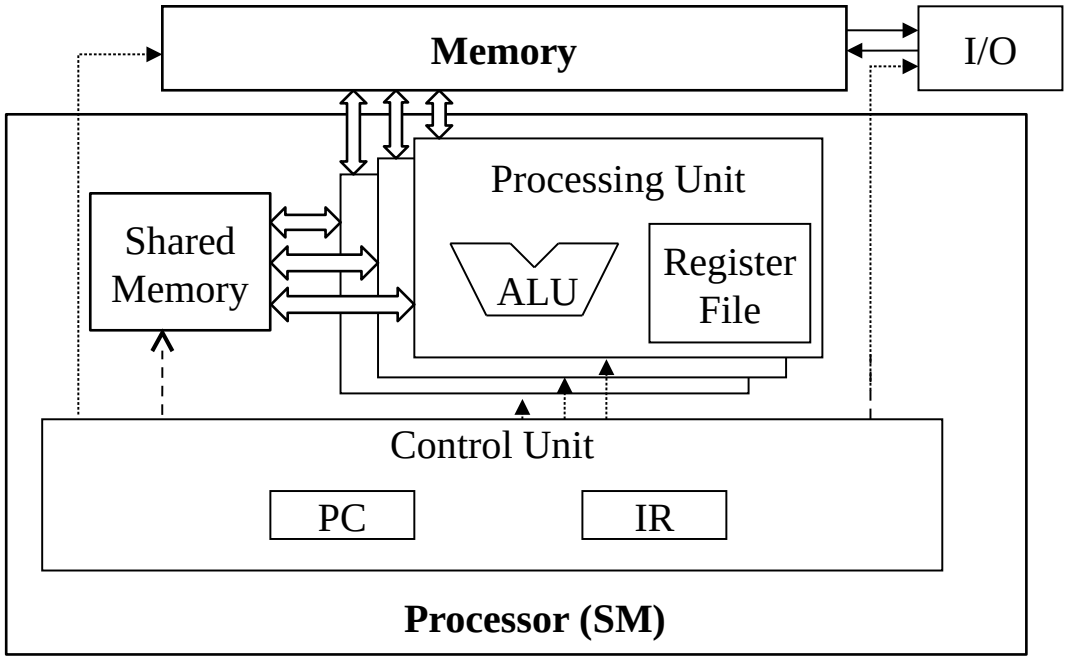
Thread blocks are partitioned into warps based on thread indices

For warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63

For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linearized row-major order before partitioning into warps

SMs are SIMD Processors

Control unit for instruction fetch, decode, and control is shared among multiple processing units



SIMD Among Threads in a Warp

All threads in a warp must execute the same instruction at any point in time

This works efficiently if all threads follow the same control flow path

- All if-then-else statements make the same decision
- All loops iterate the same number of times

Control Divergence

Control divergence occurs when threads in a warp take different control flow paths by making different control decisions

- Some take the then-path and others take the else-path of an if-statement
- Some threads take different number of loop iterations than others

The execution of threads taking different paths are serialized in current GPUs

- The control paths taken by the threads in a warp are traversed one at a time until there is no more
- During the execution of each path, all threads taking that path will be executed in parallel
- The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

Divergence can arise when branch or loop condition is a function of thread indices

Example kernel statement with divergence:

if (threadIdx.x > 2) { }

This creates two different control paths for threads in a block

Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

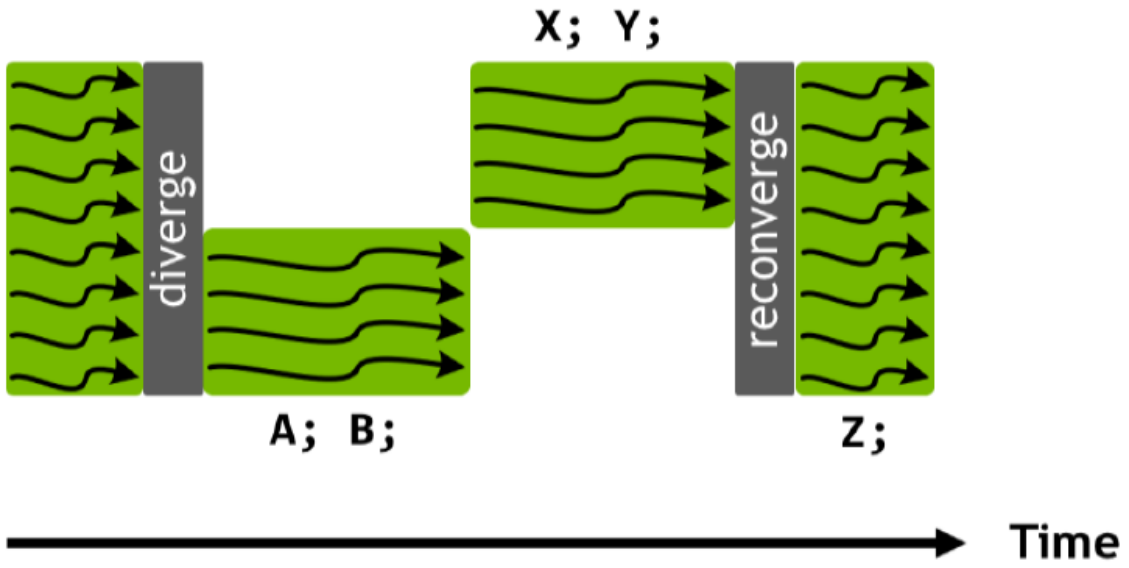
Example without divergence:

If (blockIdx.x > 2) { }

Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Thread Scheduling - Pascal

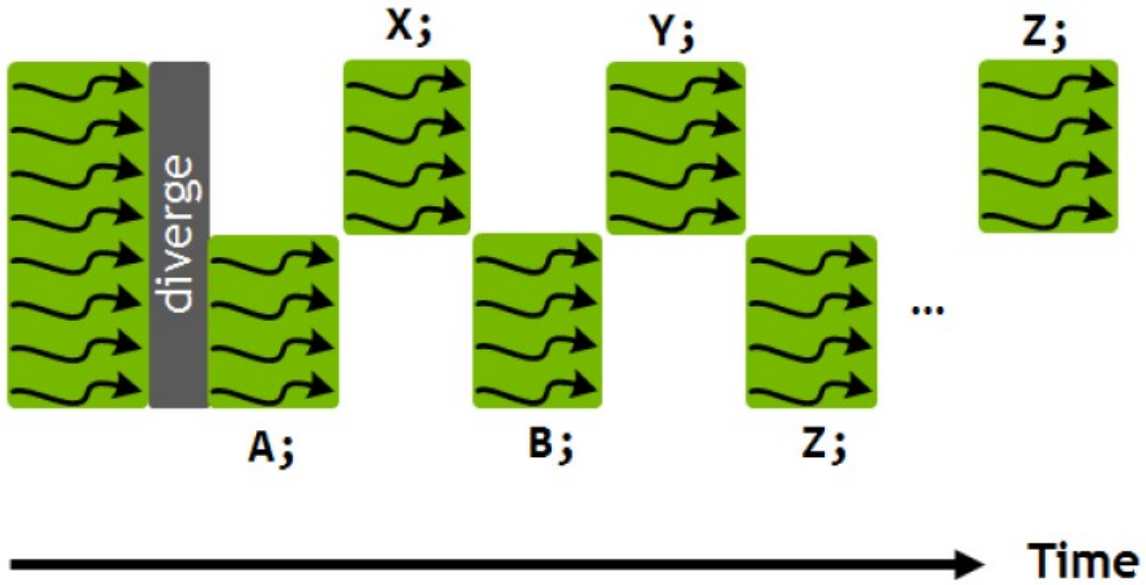
```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Thread scheduling under the SIMT warp execution model of Pascal and earlier NVIDIA GPUs. Capital letters represent statements in the program pseudocode. Divergent branches within a warp are serialized so that all statements in one side of the branch are executed together to completion before any statements in the other side are executed. After the else statement, the threads of the warp will typically reconverge.

Thread Scheduling - Volta

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



Volta independent thread scheduling enables interleaved execution of statements from divergent branches. This enables execution of fine-grain parallel algorithms where threads within a warp may synchronize and communicate.

Control Divergence Example

```
//my_Func_then and my_Func_else are some device functions
if (threadIdx.x <16)
{
  myFunc_then();
  __syncthreads();
}else if (threadIdx.x >=16)
{
  myFunc_else();
  __syncthreads();
}
```

References

Chapter 5

(Programming Massively Parallel Processors : A Hands-on Approach, David B. Kirk, Wen-Mei W. Hwu, Morgan Kaufmann Publishers, 3rd edition)