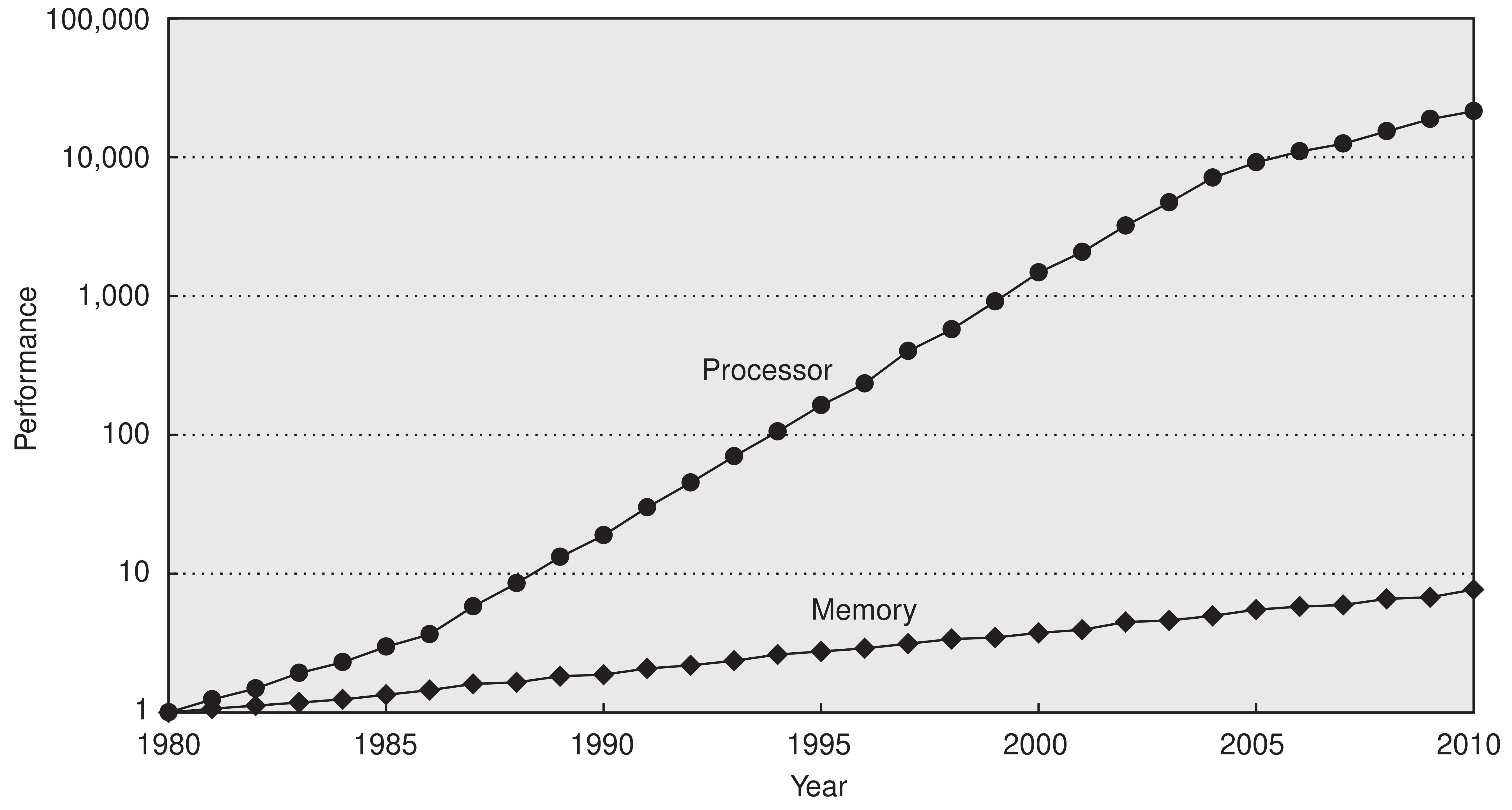
Performant Programming for GPUs

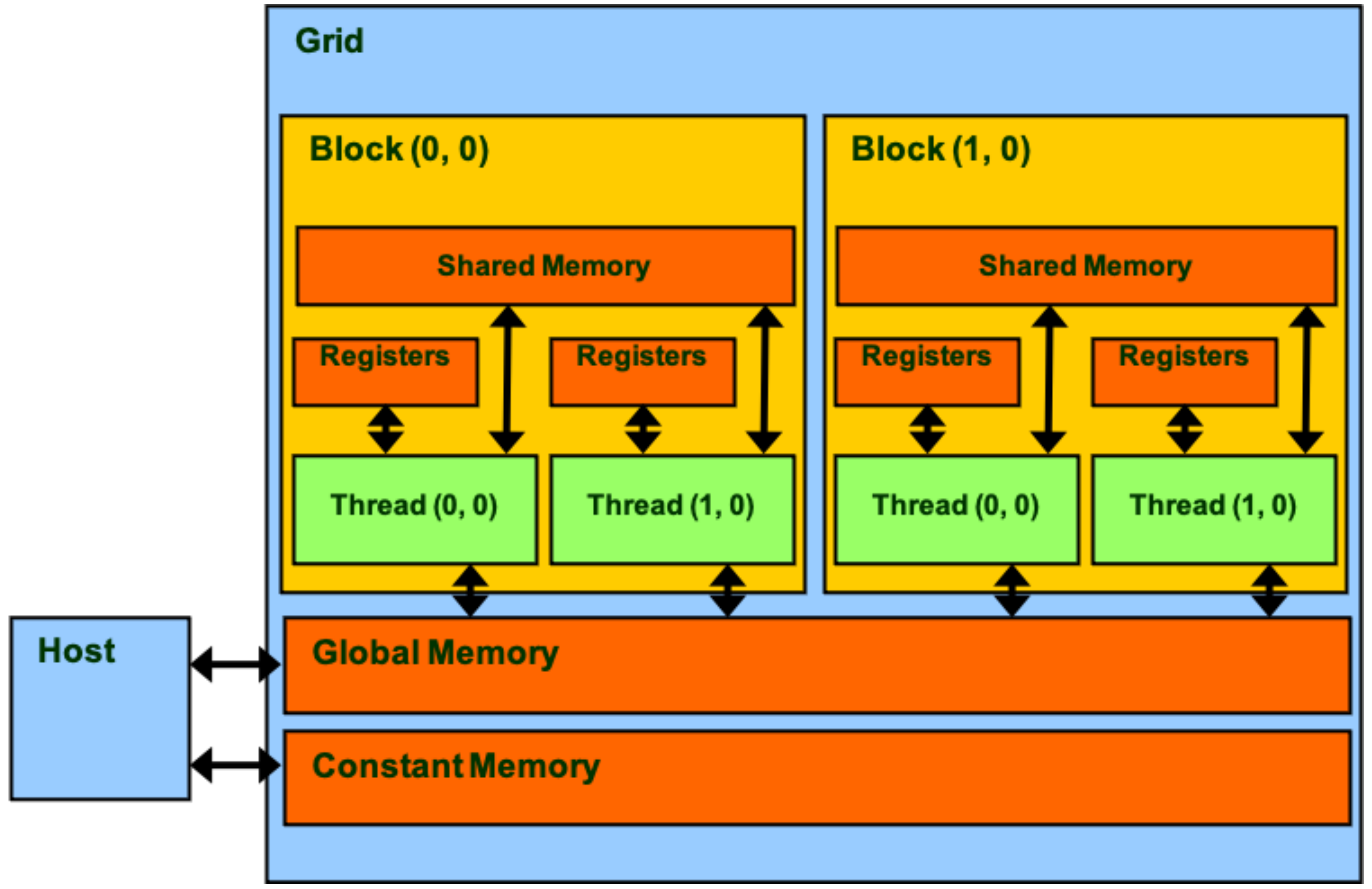
Daniel Cámpora | thematic CERN School of Computing

Table of Contents

Price to Pay for Memory

As you already know, **memory** is a key item to consider when optimizing a program.





Data Locality

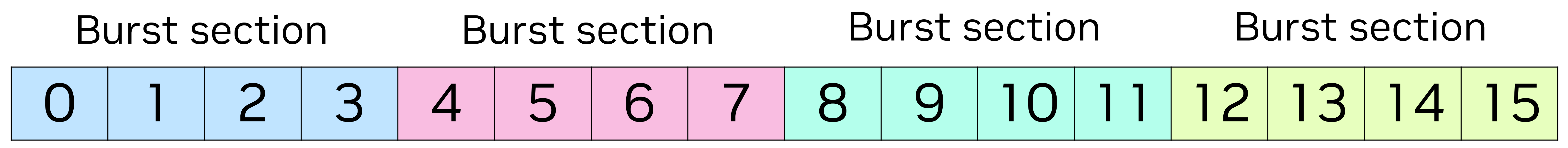
- **Space locality** – Neighbouring memory locations are likely going to be accessed.
- **Temporal locality** – The same memory location is likely going to be accessed again.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7

Accessing x_0 will load into cache all *x* elements.

- DRAM is read into cache memory – each read brings a group of items onto cache memory.

DRAM Burst Sections

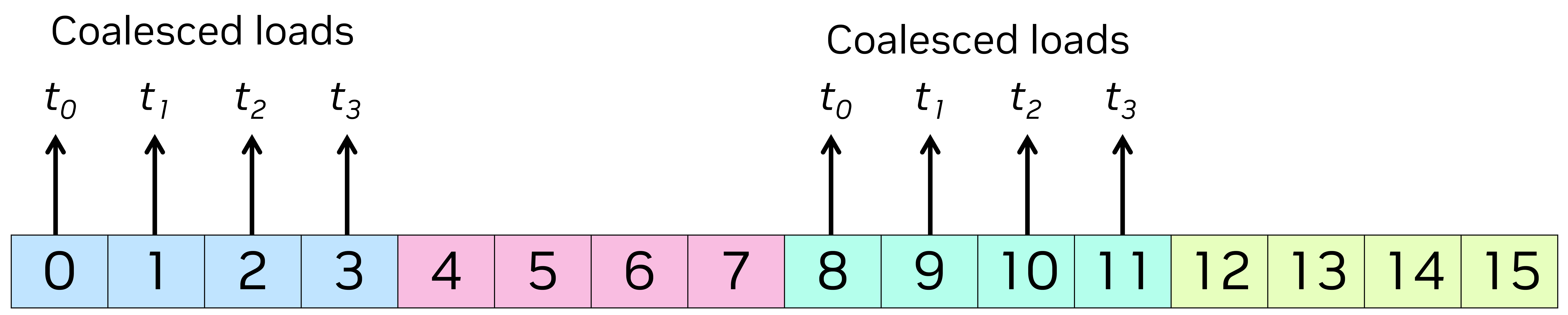


In fact, DRAM is organized in **burst sections**. Let's take a simplified example:

- Each cell represents a byte.
- We have a 16-byte address space, with 4-byte burst sections.

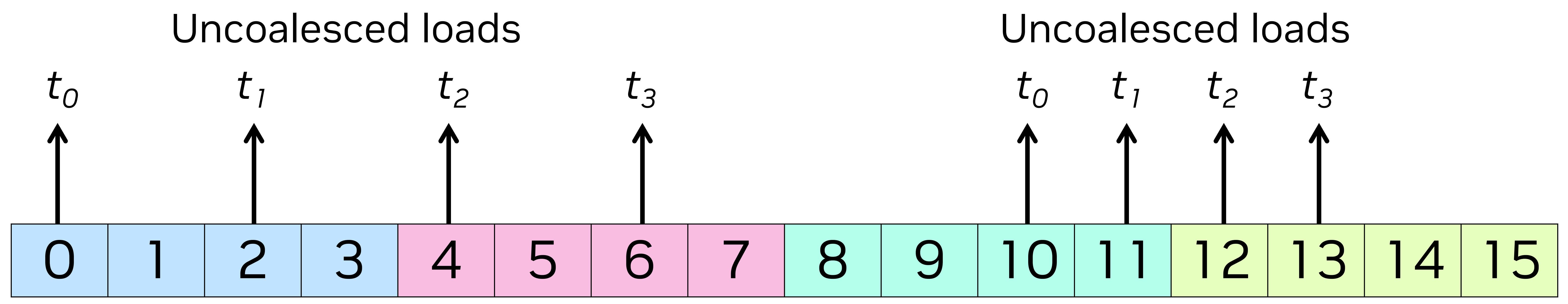
Note that nowadays the address spaces are in the GBs, and a typical burst section is 128 bytes.

Coalesced Memory Accesses



When threads make a memory request and the request falls under the same burst, the access is **coalesced**.

Non-coalesced Memory Accesses



However, if threads request a block of memory and the accesses do not fall under the same burst, the access is **non-coalesced**.

Several access patterns can yield this undesired behaviour, which impacts performance.

Is an Access Coalesced?

As a general rule, look for the following conditions:

- Base address should be a multiple of burst size.
- `threadIdx` should be used as a free term.

Eg. coalesced access of array A:

```
A[(expression independent of threadIdx) + threadIdx.x]
```

Quickbit: Linear Representation of a Matrix

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

is actually stored as

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

Tip: Always store higher order arrays as 1-dimensional arrays!

Matrix-matrix Multiplication

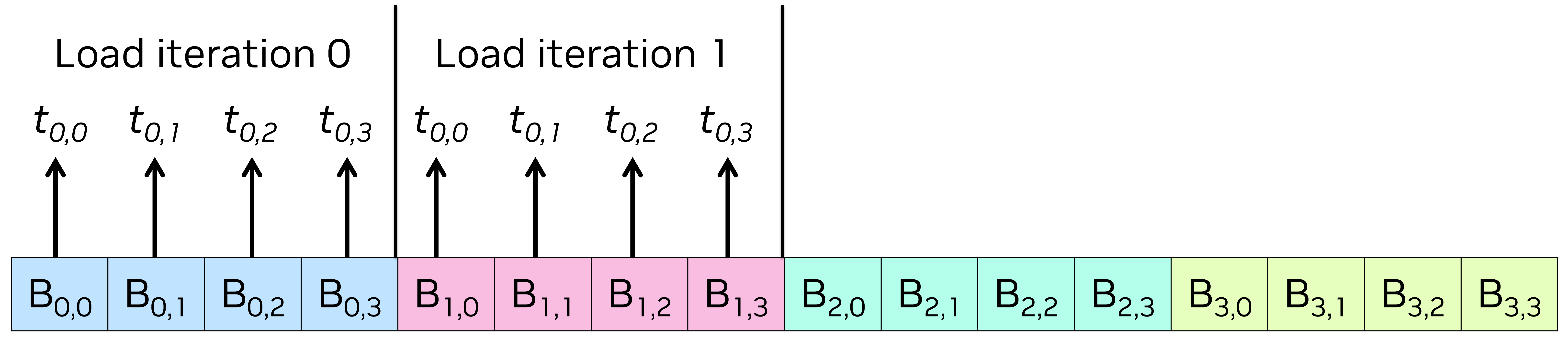
Suppose we want to multiply two arrays:

- **A** of size `m x n`
- **B** of size `n x k`
- Result is **C** of size `m x k`

```
__device__ void multiply_arrays(float* A, float* B, float* C, int m, int n, int k) {
  for (int row = threadIdx.x; row < m; row += blockDim.x) {
    for (int col = threadIdx.y; col < k; col += blockDim.y) {
      float element = 0.f;
      for (int i = 0; i < n; ++i) {
        element += A[row * m + i] * B[i * k + col];
      }
      C[row * k + col] = element;
    }
  }
}
```

Access Patterns

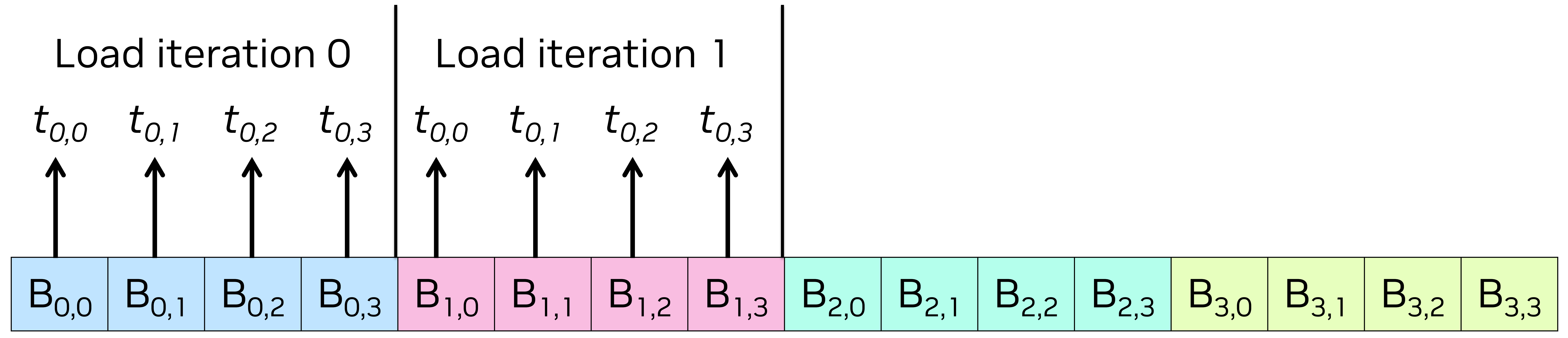
Accesses to B are **coalesced**:



Load iteration 0
Load iteration 1
$t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$ $t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

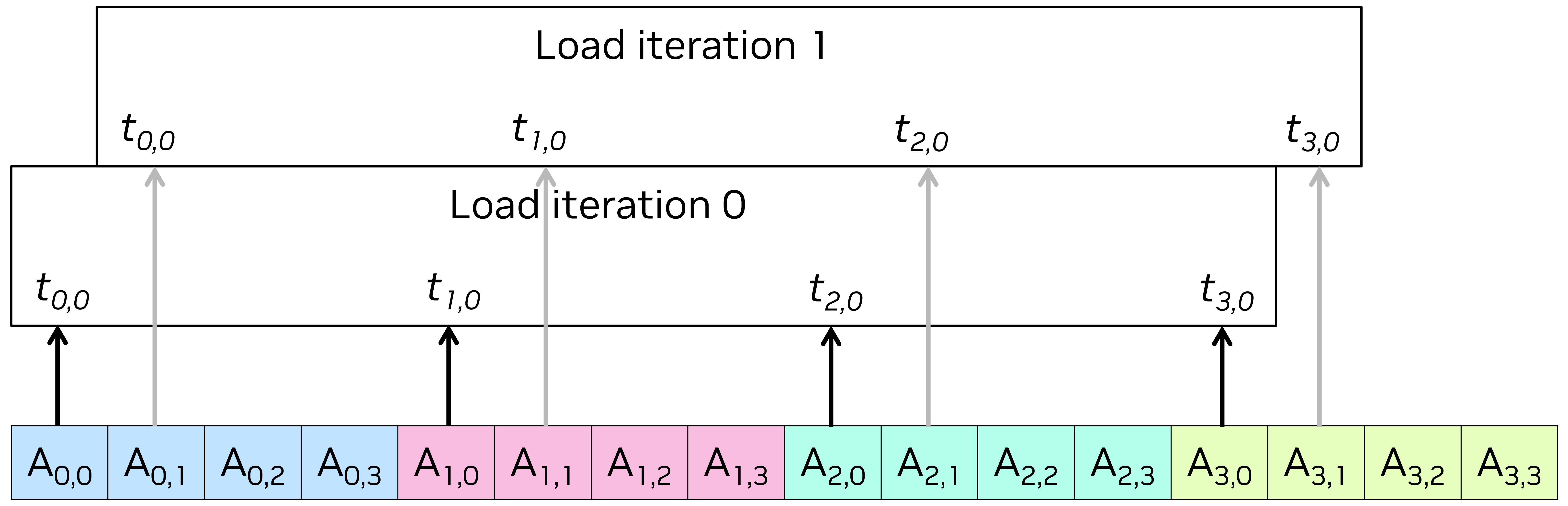
Access Patterns

Accesses to B are **coalesced**:



Load iteration 0
Load iteration 1
$t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$ $t_{0,0}$ $t_{0,1}$ $t_{0,2}$ $t_{0,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

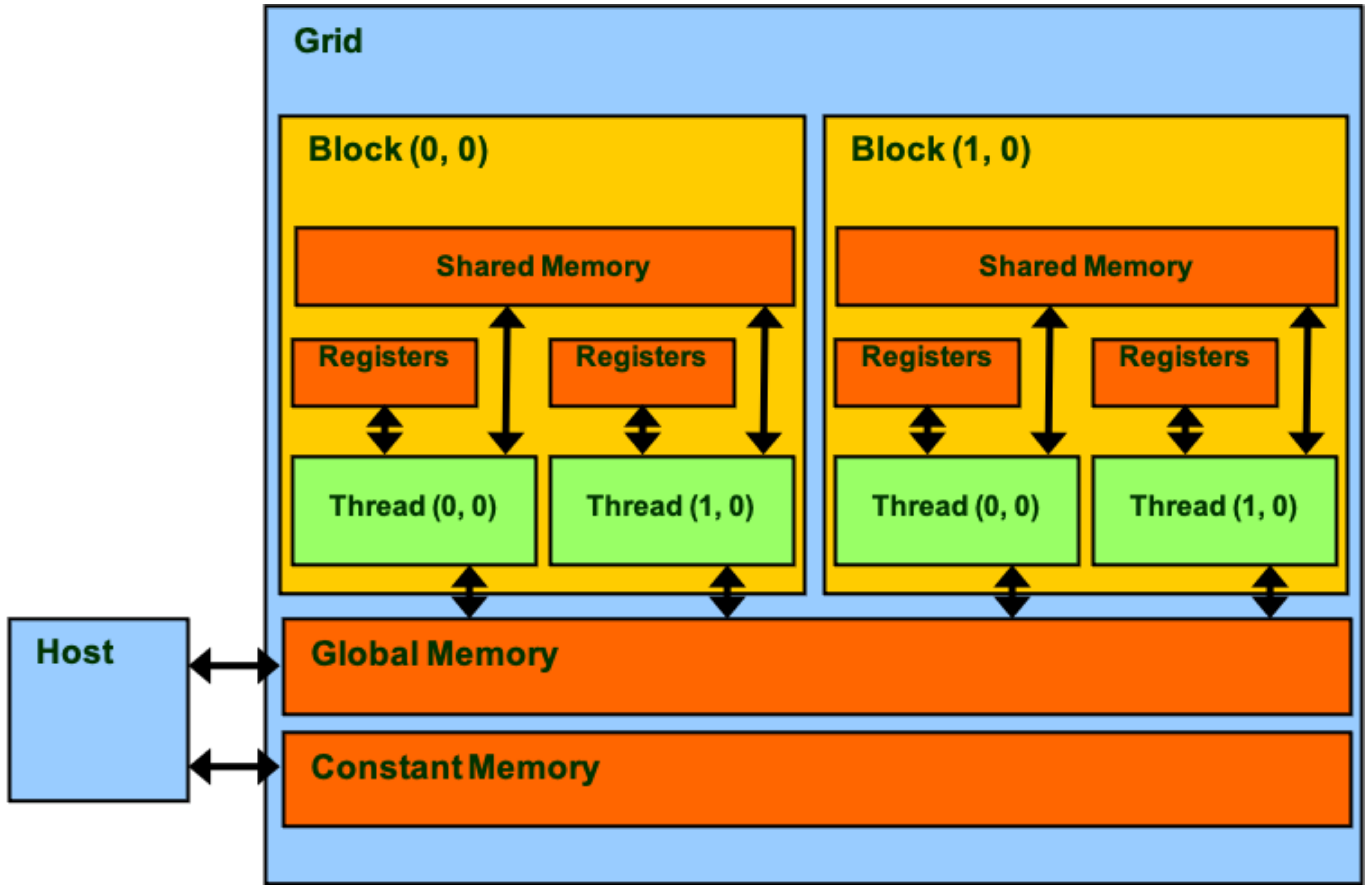
However, accesses to A are **non-coalesced**:



Load iteration 1
$t_{0,0}$ $t_{1,0}$ $t_{2,0}$ $t_{3,0}$
Load iteration 0
$t_{0,0}$ $t_{1,0}$ $t_{2,0}$ $t_{3,0}$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$

A Step Further: Shared Memory

Going back to the types of memories available in a GPU:



Shared memory is a low-latency memory that reides on L1 cache.

How to Use Shared Memory

Shared memory can be defined by using the keyword `__shared__`

Any variable declared like this will be accessible by **all threads in a block**.

```
__global__ void shared_memory_example(float* dev_array) {
  __shared__ float array [256];

  for (int i = threadIdx.x; i < 256; i += blockDim.x) {
    array[i] = dev_array[i];
  }

   __syncthreads();

  // Now all threads can access array, which is initialized with
  // the first 256 elements of dev_array.
}
```

Things to Consider about Shared Memory

Shared memory is a scarce resource that should be used carefully.

- It is limited in size, the maximum varies depending on the architecture.
- It is a limiting resource that is used to determine maximum number of blocks in flight in a Streaming Multiprocessor (SM).

The amount of memory reserved for `L1`-cache / shared memory is configurable (in CUDA, it can be configured with `cudaDeviceSetCacheConfig`).

At the same time, a good use of shared memory can lead to juicy performance gains!

Matrix Multiplication With Shared Memory

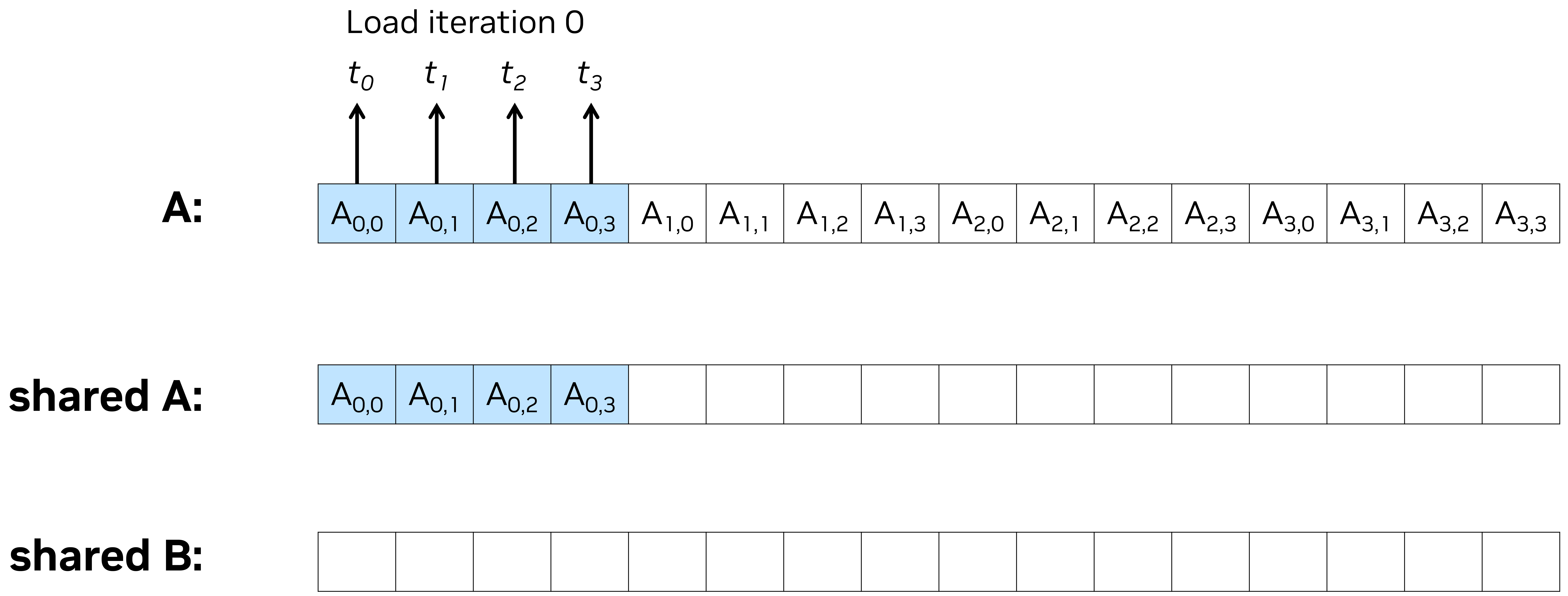
With a small enough matrix, we could use shared memory:

- Preload all elements of **A** and **B** onto shared memory.
- Perform matrix multiplication reading from shared memory and store the result in **C**.

Bonus point: We can use **coalesced accesses** to populate the shared memory buffers!

Small Matrix Multiplication Example

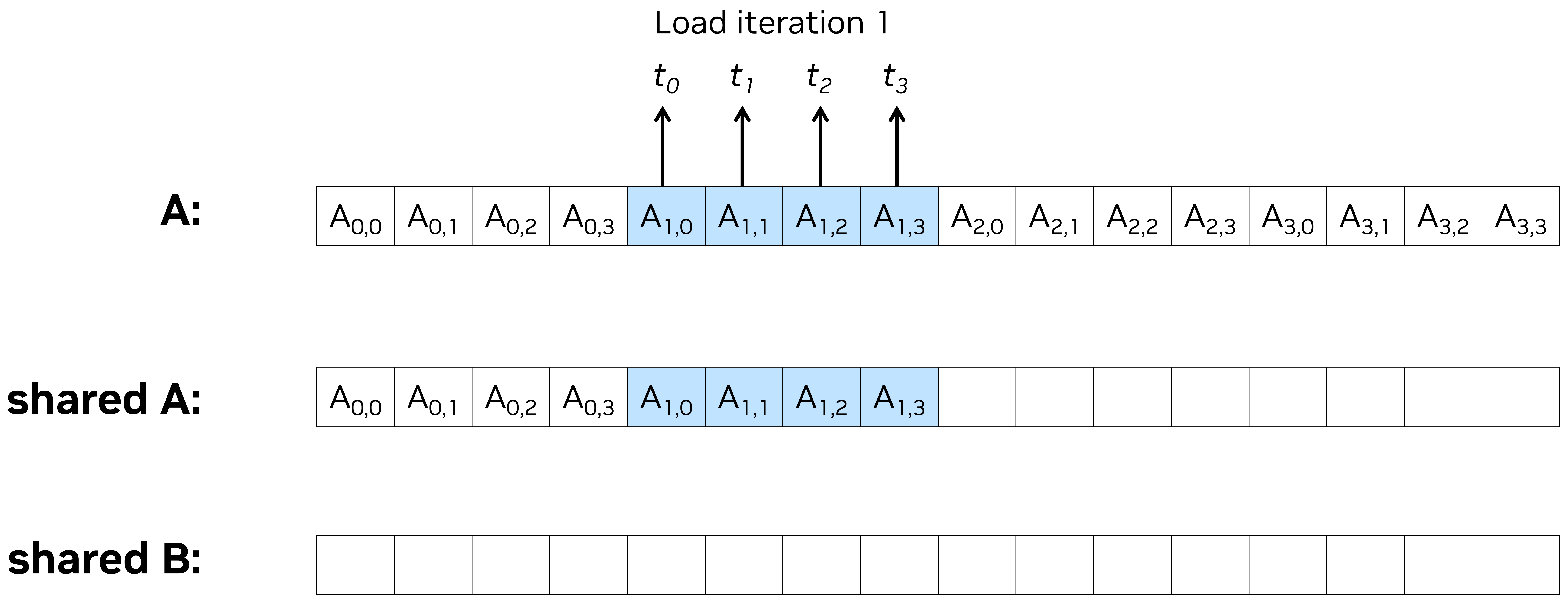
Load **A** and **B** onto shared memory:



Load iteration 0
t_0
t_1
t_2
t_3
A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
shared B:

Small Matrix Multiplication Example

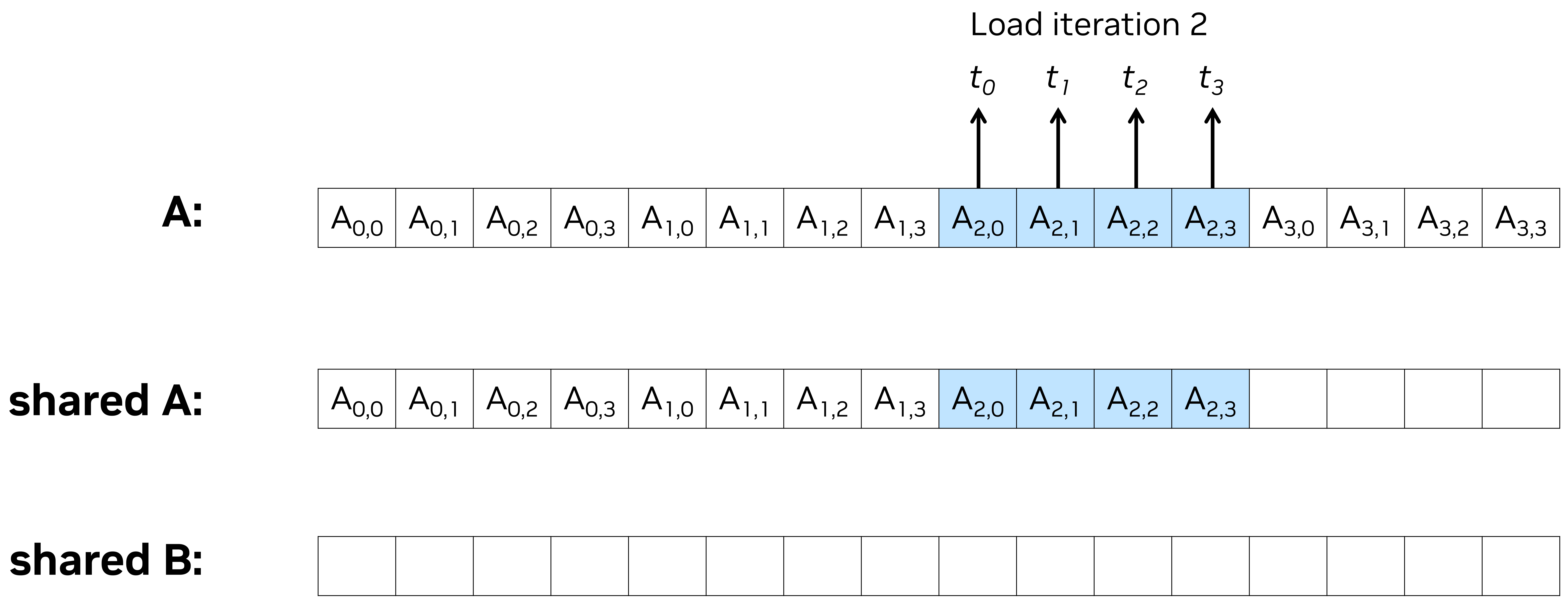
Load **A** and **B** onto shared memory:



Load iteration 1
t_0
t_1
t_2
t_3
A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
shared B:

Small Matrix Multiplication Example

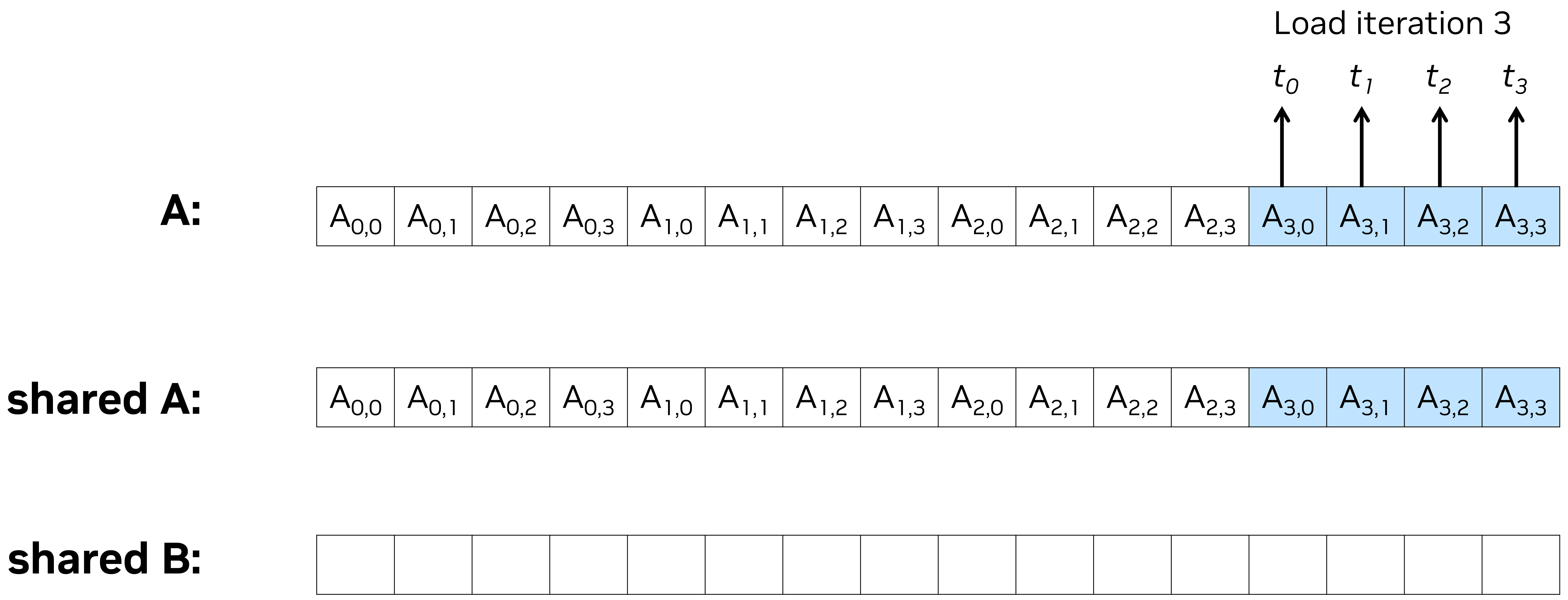
Load **A** and **B** onto shared memory:



Load iteration 2
t_0
t_1
t_2
t_3
A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
shared B:

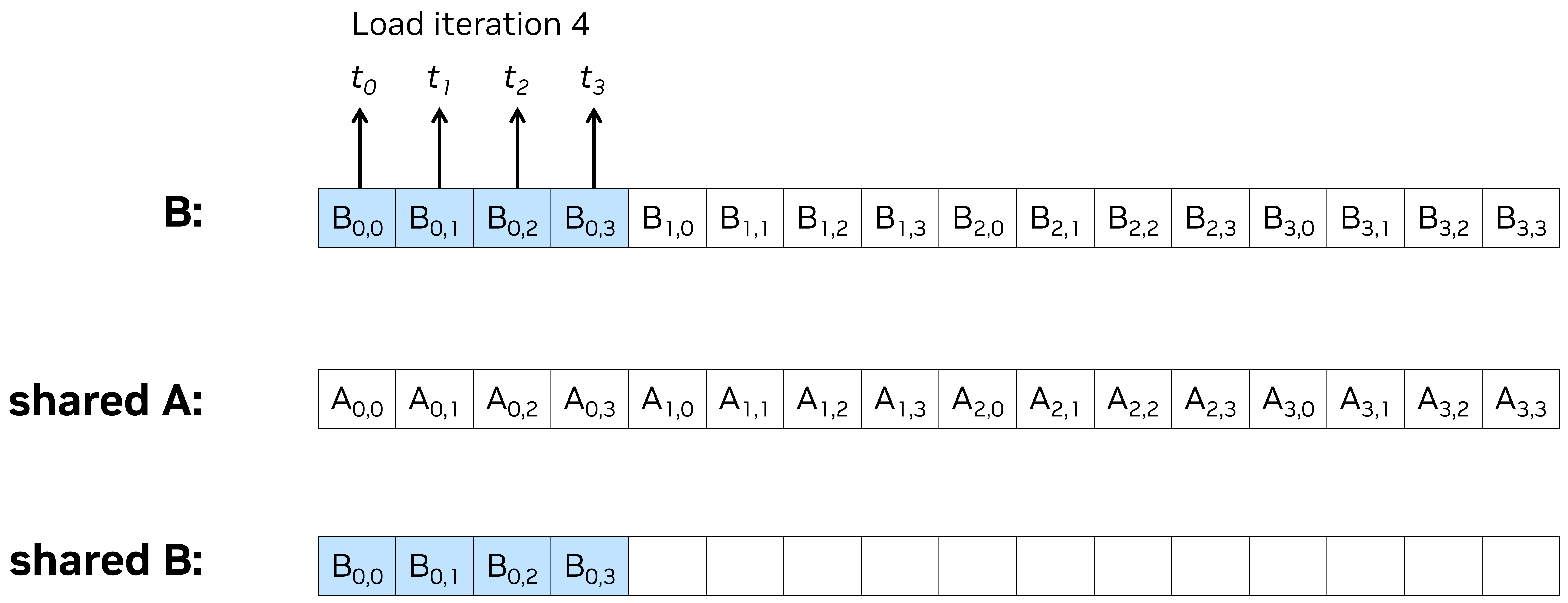
Small Matrix Multiplication Example

Load **A** and **B** onto shared memory:


Load iteration 3
t_0
t_1
t_2
t_3
A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared B:

Small Matrix Multiplication Example

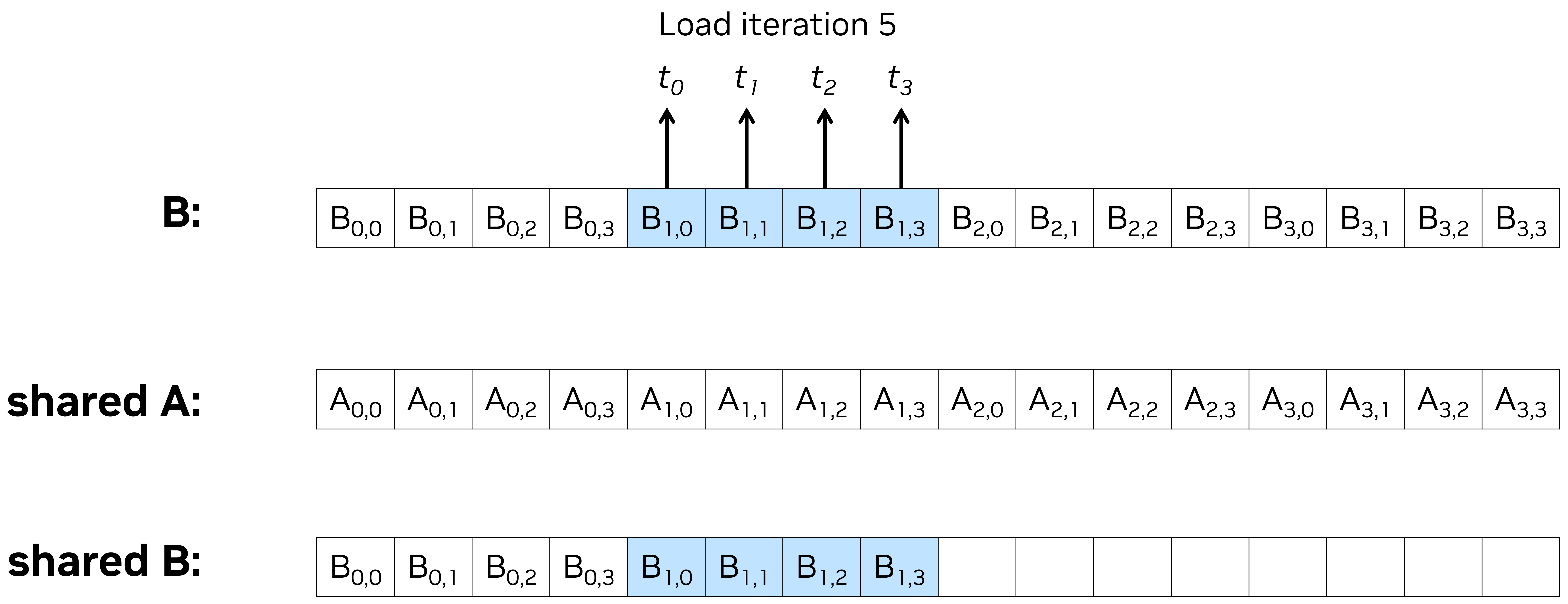
Load **A** and **B** onto shared memory:



Load iteration 4
t_0
t_1
t_2
t_3
B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$

Small Matrix Multiplication Example

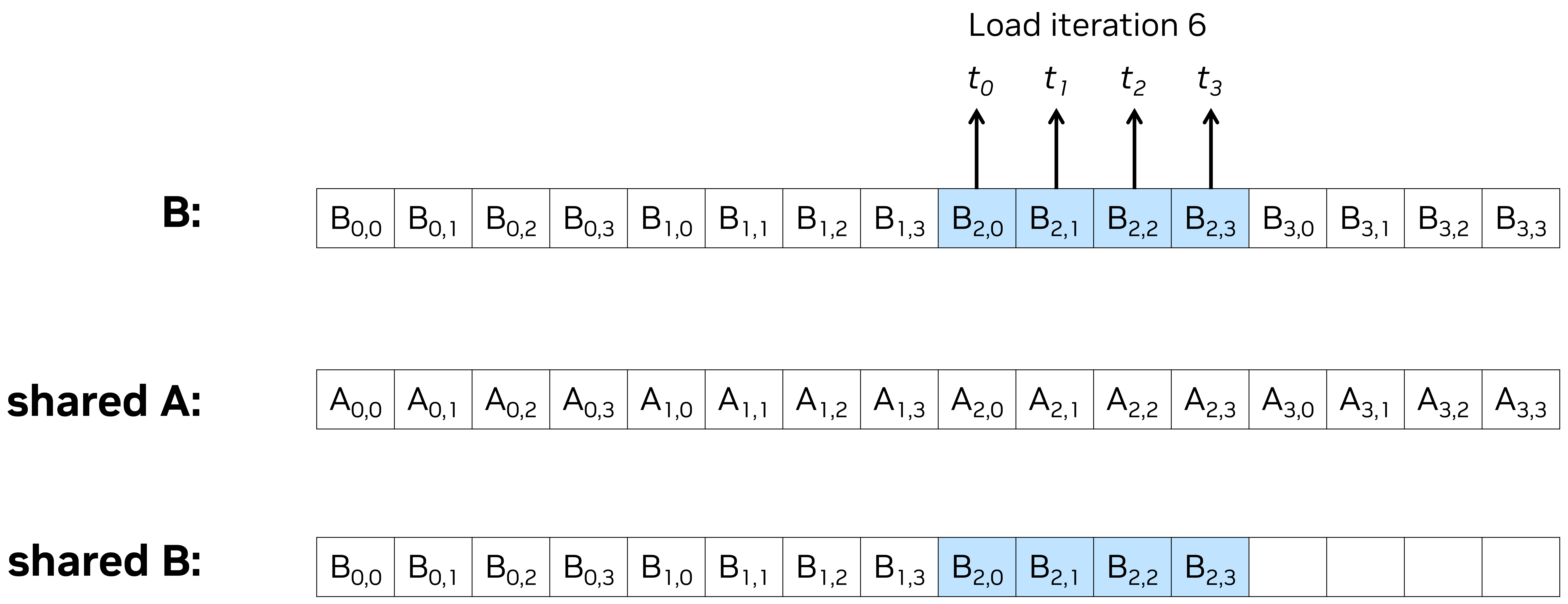
Load **A** and **B** onto shared memory:



Load iteration 5
t_0
t_1
t_2
t_3
B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$

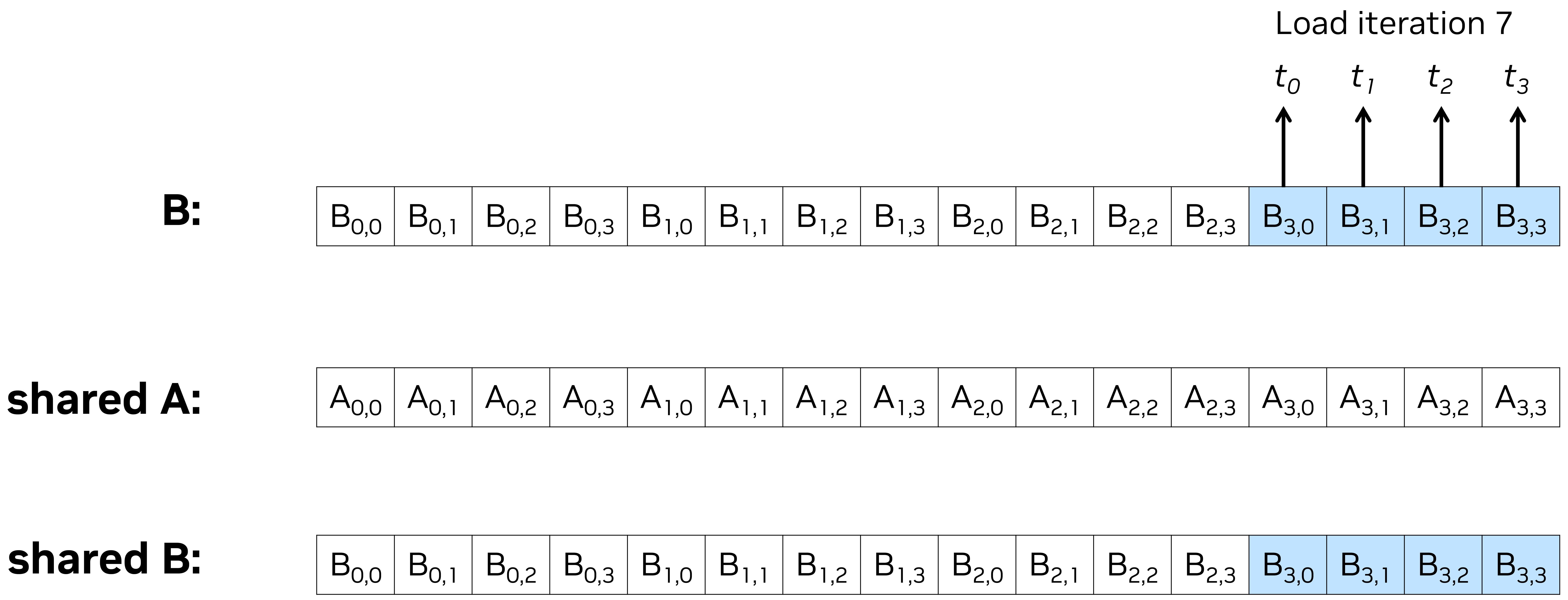
Small Matrix Multiplication Example

Load **A** and **B** onto shared memory:


Load iteration 6
t_0
t_1
t_2
t_3
B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$

Small Matrix Multiplication Example

Load **A** and **B** onto shared memory:



Load iteration 7
t_0 t_1 t_2 t_3
B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
shared A:
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
shared B:
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$

Small Matrix Multiplication Example (2)

shared A:	$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$
shared B:	$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$	$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$	$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$	$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

And finally do the matrix-matrix multiplication from shared memory buffers **shared A** and **shared B**, storing it in **C**.

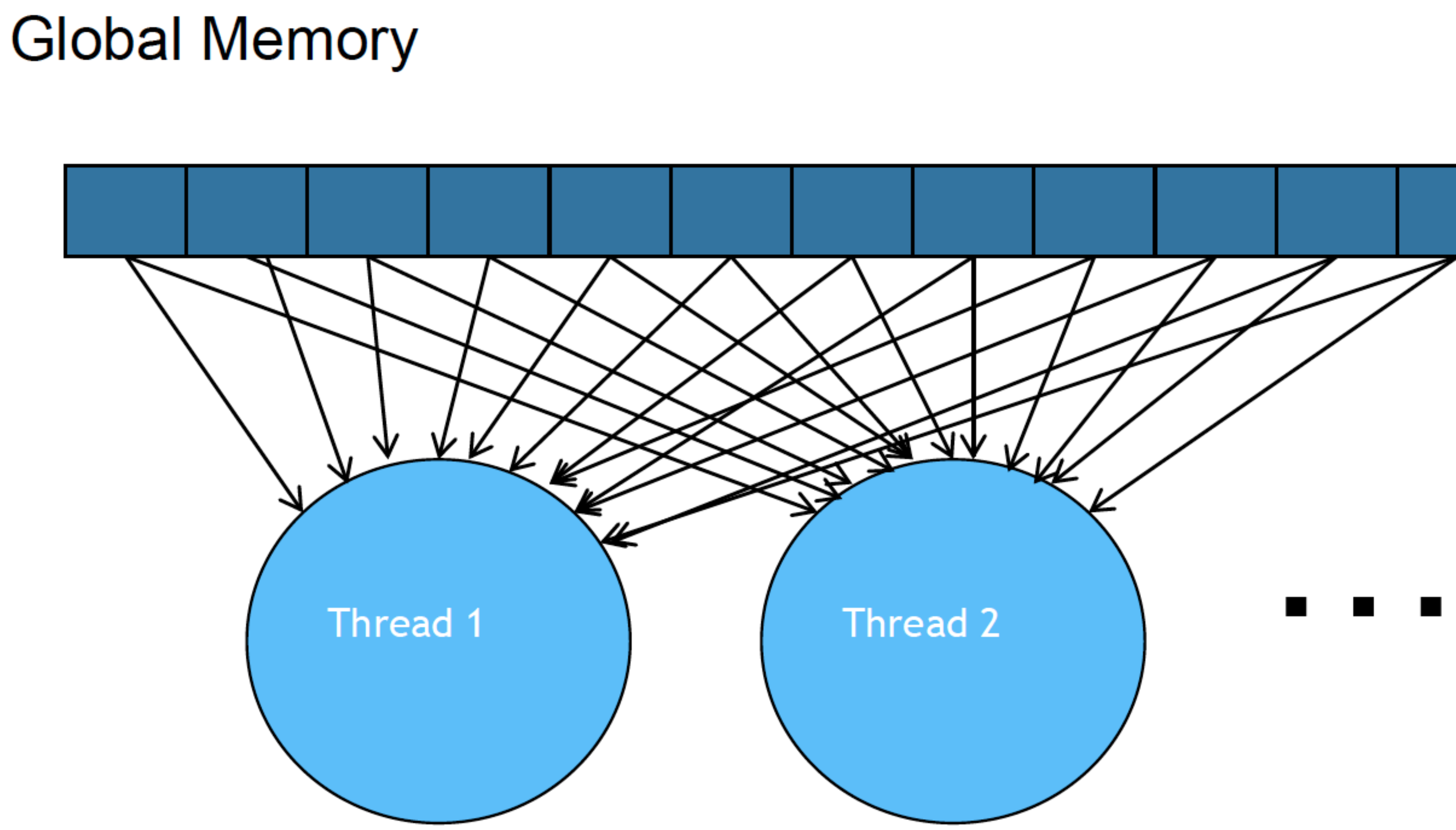
```
__global__ void shared_matrix_multiply_16_16(float* A, float* B, float* C) {
  __shared__ float shared_A [256];
  __shared__ float shared_B [256];

  // Coalesced loads
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
    shared_A[i] = A[i];
  for (int i = threadIdx.x; i < 256; i += blockDim.x)
    shared_B[i] = B[i];
  __syncthreads();

  // Now shared_A and shared_B are populated and can be used
  // instead of the original arrays to perform the multiplication
  multiply_arrays(shared_A , shared_B , C, 16, 16, 16);
}
```

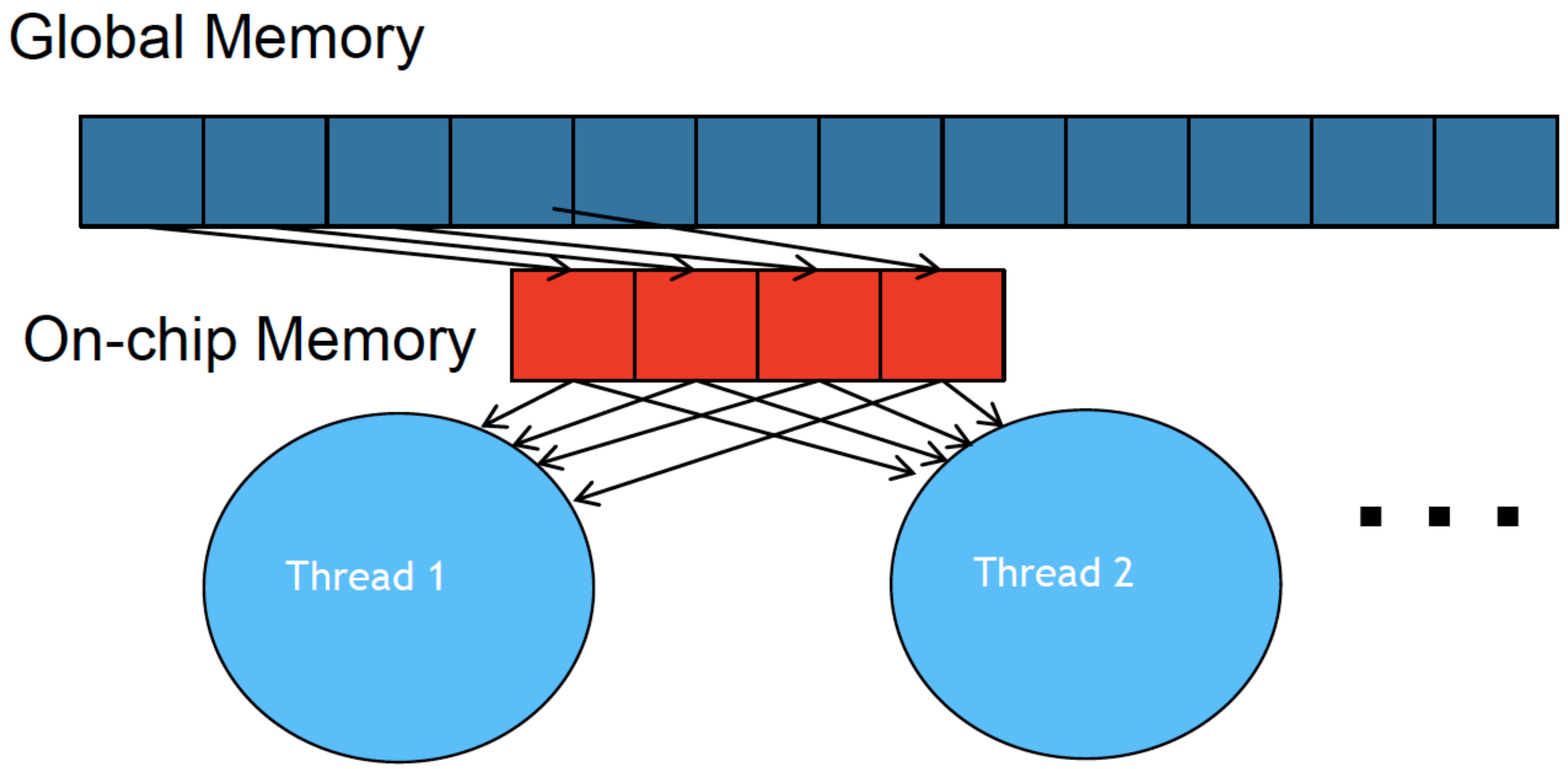
Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



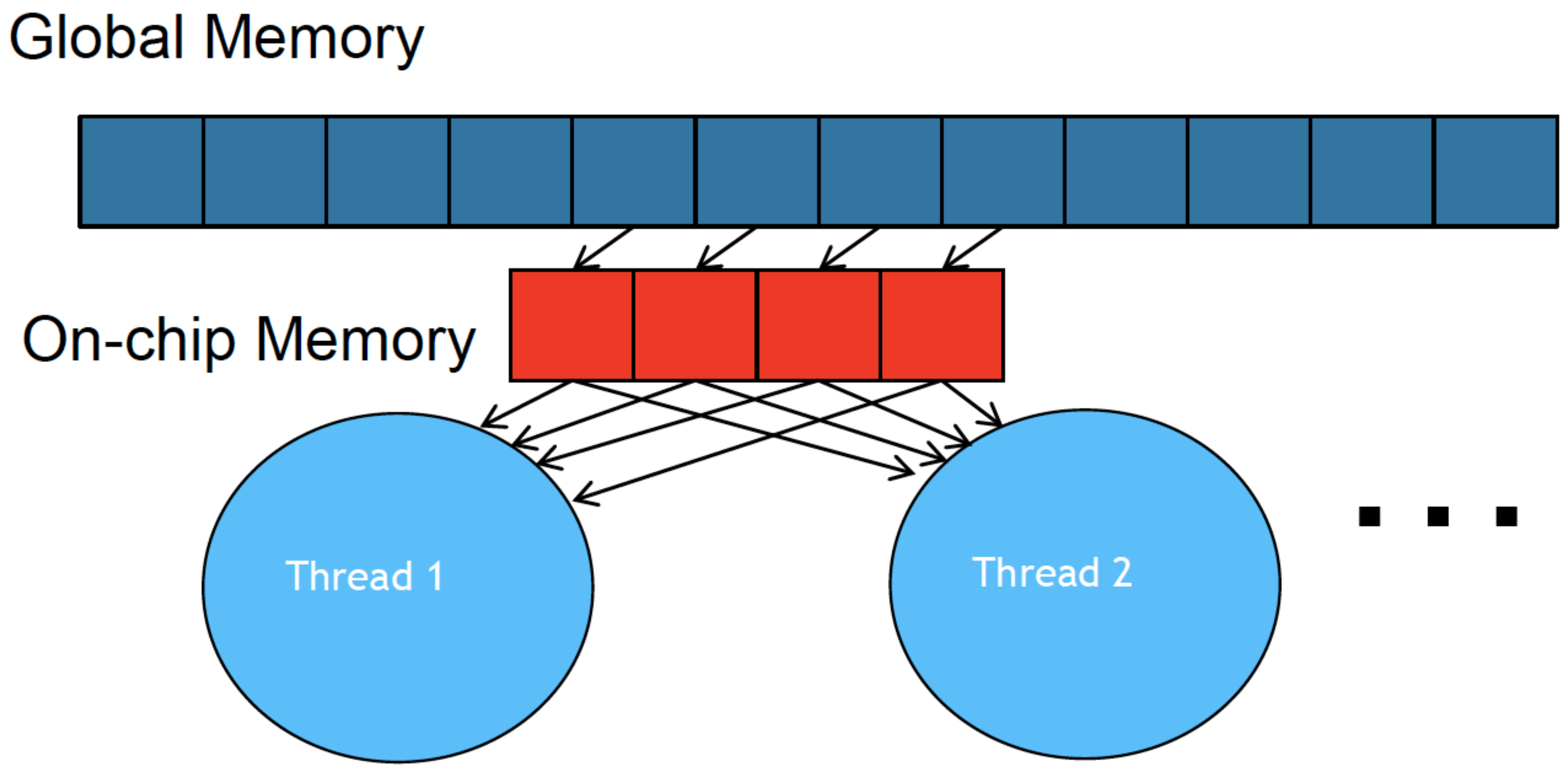
Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



Tiling

Tiled data processing or just **tiling** consists in dividing a big chunk of data into many tiles which are processed one at a time.



Analogy

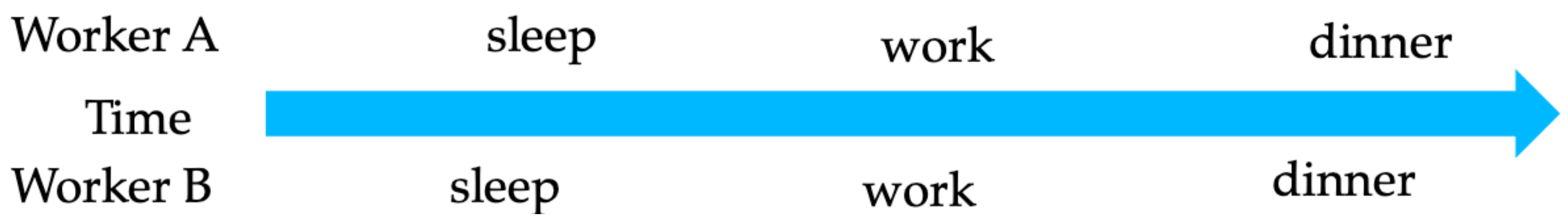
The basic concept is similar to carpooling:

- Drivers / Passengers – threads accessing memory.
- Cars – memory access requests.

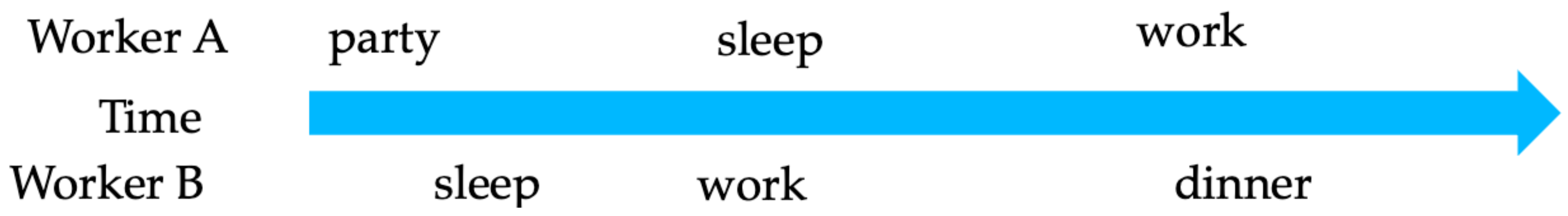
Image from *NVIDIA DLI course "Fundamentals of Accelerated Computing with CUDA C/C++"*

Schedule Is Important!

It works well when people have similar schedules



But it goes really wrong otherwise!



A Generic Tiling Algorithm

Follow the next steps:

1. Identify an access pattern where threads access global memory in a tiled manner.
2. Load the tile from global into shared memory in a coalesced manner.
3. Synchronize.
4. Have multiple threads access the data from the shared buffer.
5. Synchronize.
6. Move on to the next tile.

Tiled Matrix Multiplication

Using this technique we can multiply two arrays of *any given size* by dividing it into tiles.

At every step, we will load the data into shared memory and perform the multiplication.

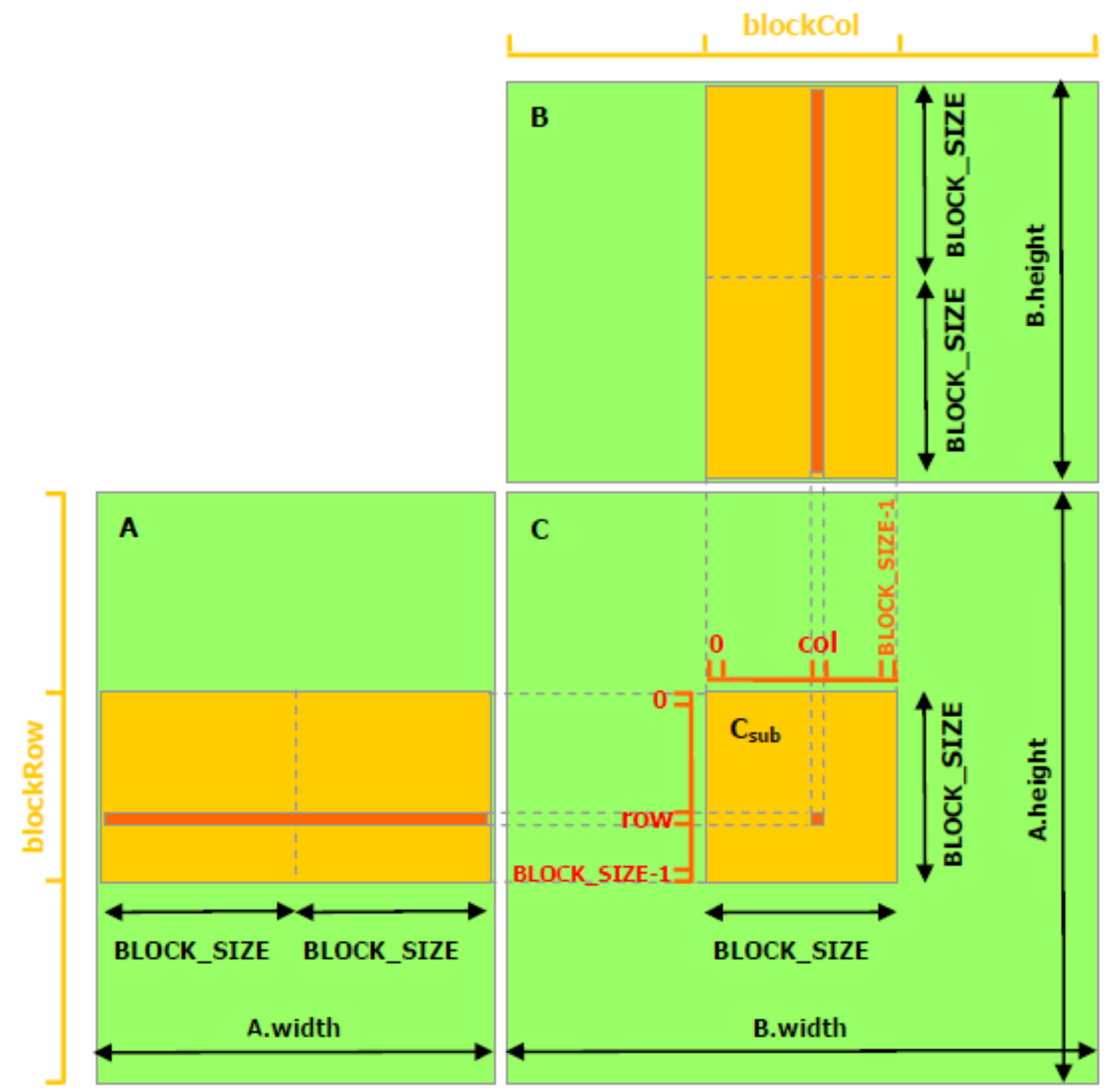


Table of Contents

Streams

A stream is a sequence of commands that execute in order.

A stream can execute various types of commands. For instance:

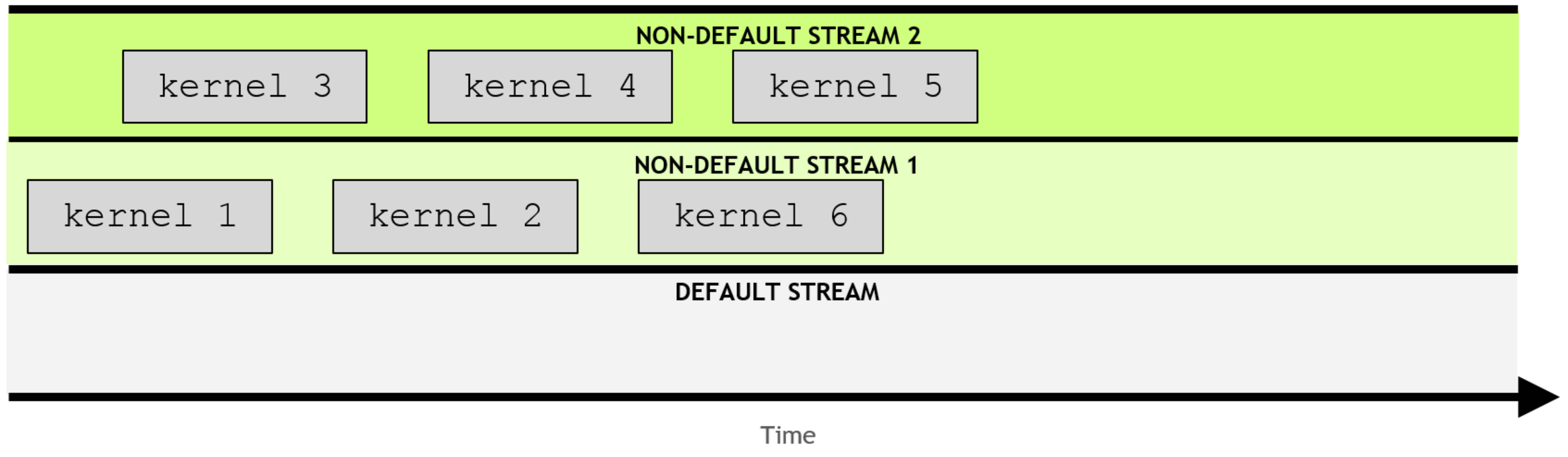
- Kernel invocations
- Memory transmissions.
- Memory (de)allocations.
- Memsets.
- Synchronizations.

The Default Stream

DEFAULT STREAM
kernel 1
kernel 2
kernel 3
kernel 4
kernel 5
Time

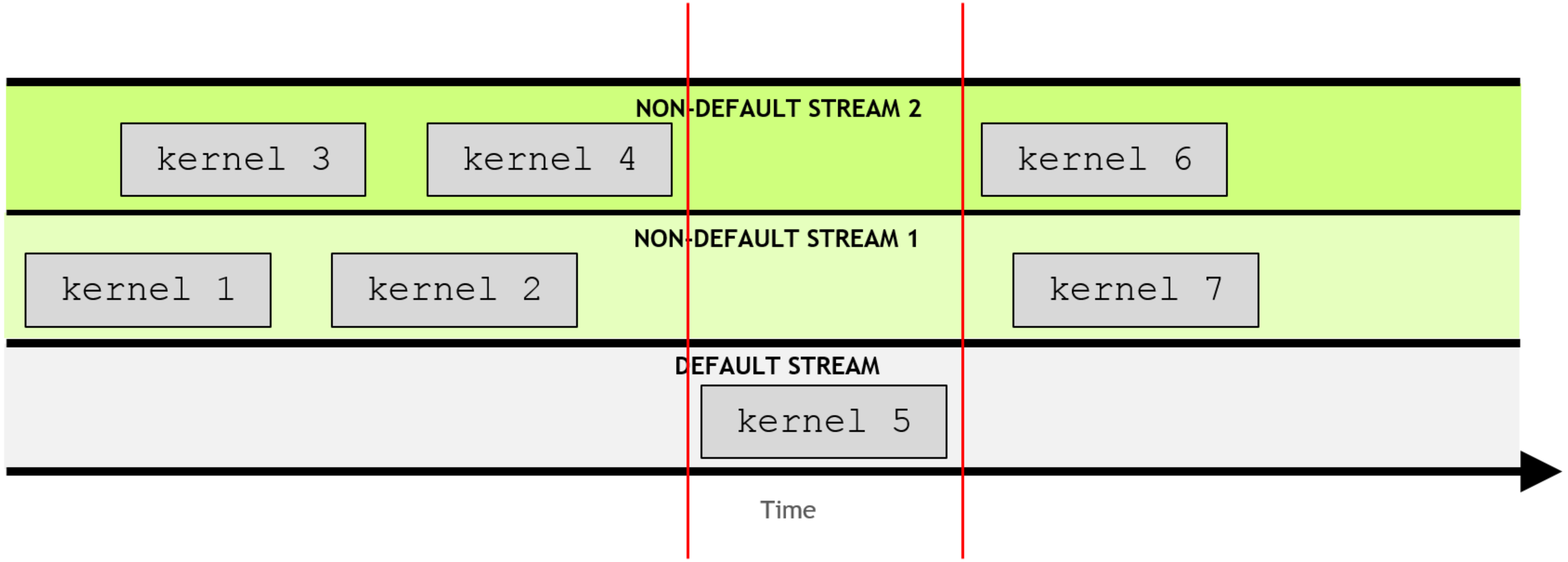
- A CUDA stream is a *sequence of operations executed in issue order*.
- CUDA has a default stream.
- By default, CUDA kernels run in the default stream.
- Any instruction run in a stream must complete before the next instruction can be issued.

Non-default streams



- Non-default streams can also be created in a CUDA application.
- Commands running on a non-default stream must still complete before the next can be issued.
- However, commands in different, non-default streams can run concurrently.

The Default Stream is Blocking



- The default stream is special: it acquires exclusive access preventing other streams from running.

Pipelines

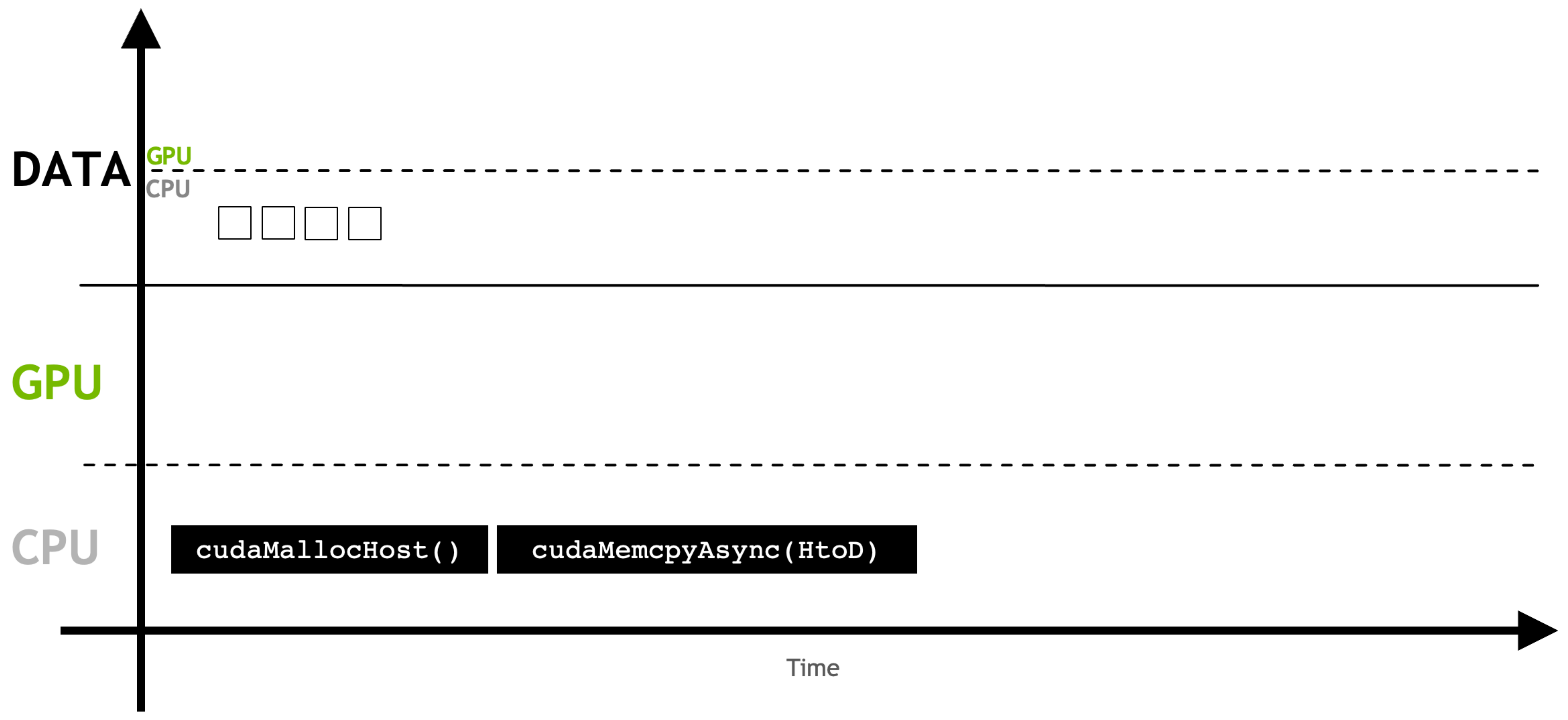
If we were to use a single stream to perform all calculations and transfer all data, GPUs would be *hopelessly slow*.

Thankfully, GPUs can perform data transmissions while executing kernels.

Given that a GPU is sitting on a PCI-express slot, we can even exploit the full-duplex capability of the link if we so desire. Typically at least three streams are needed to achieve a full pipeline:

- Use SMs to perform some computation.
- Transfer data host-to-device.
- Transfer data device-to-host.

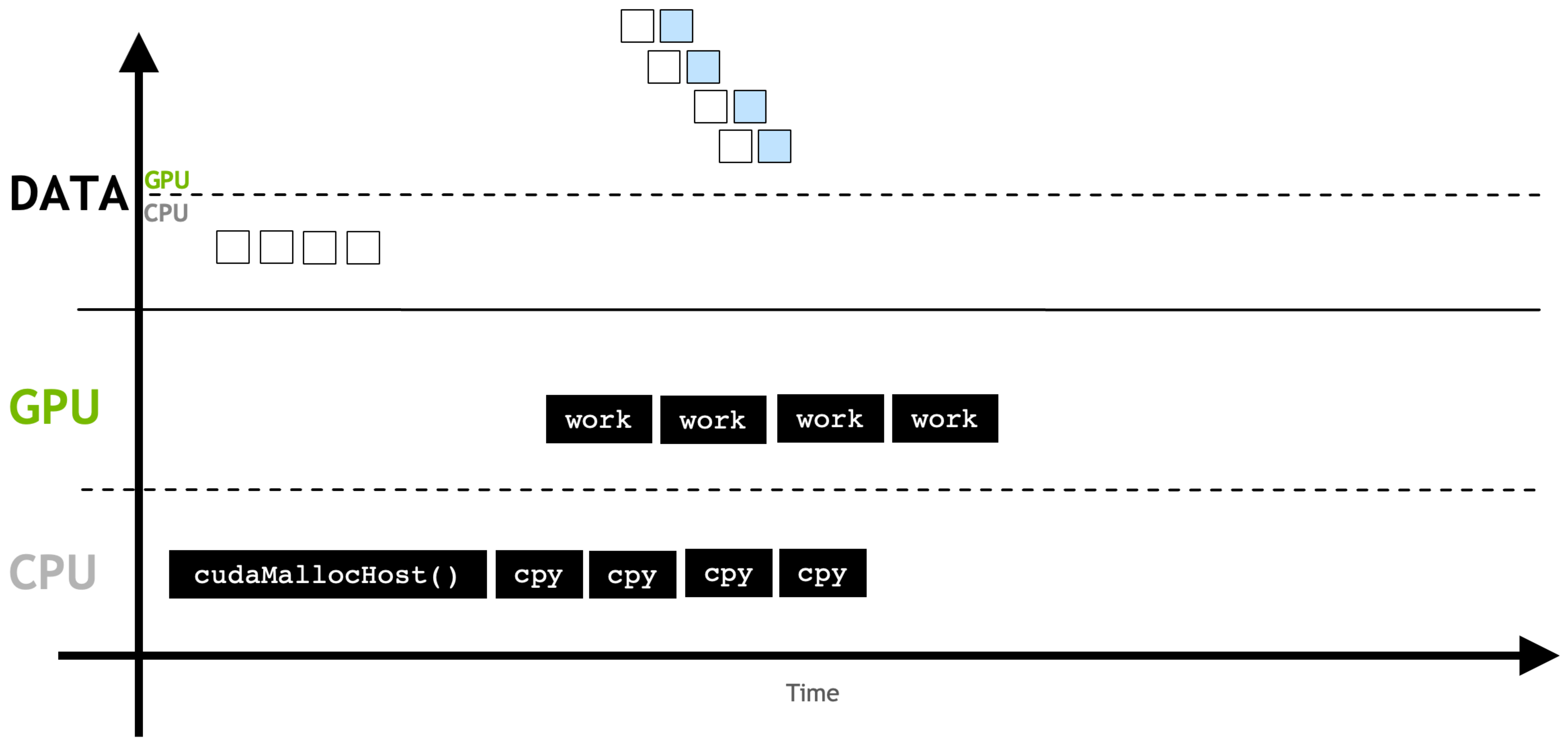
Pipeline Example



Main memory (host) must be **pinned** in order for asynchronicity to work.

`cudaMemcpyAsync` can transfer data **asynchronously** in a **non-default stream**.

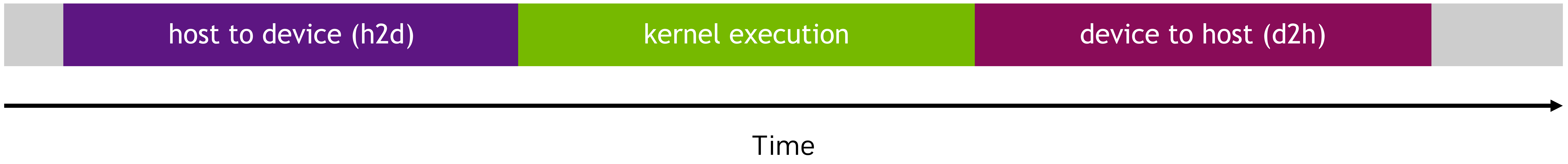
Pipeline Example (2)



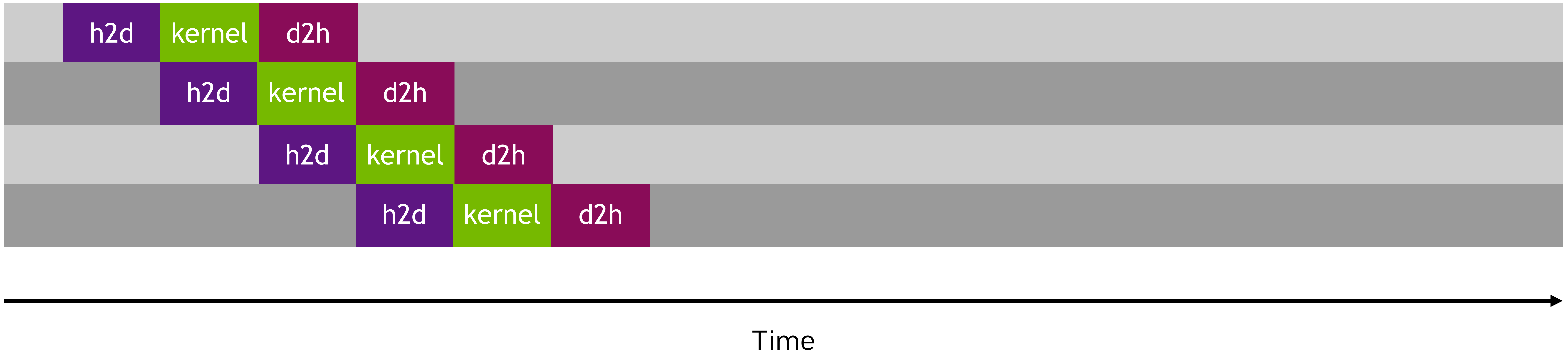
This allows overlapping memory copies and computation.

Full Pipeline

Serial:



Pipelined:



Full Pipeline (2)

Pipelined:

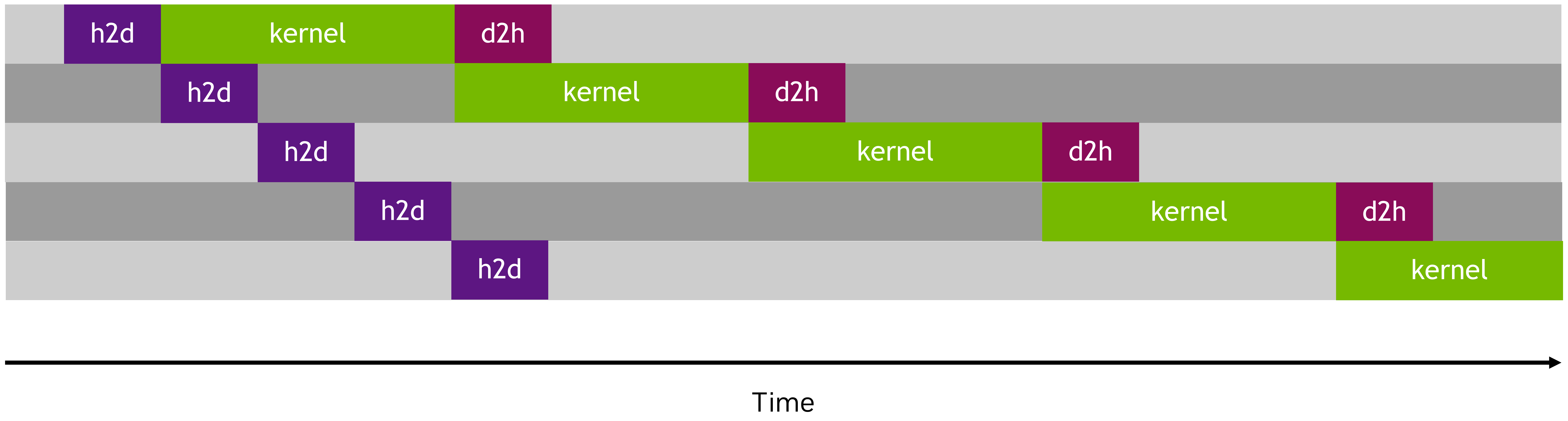


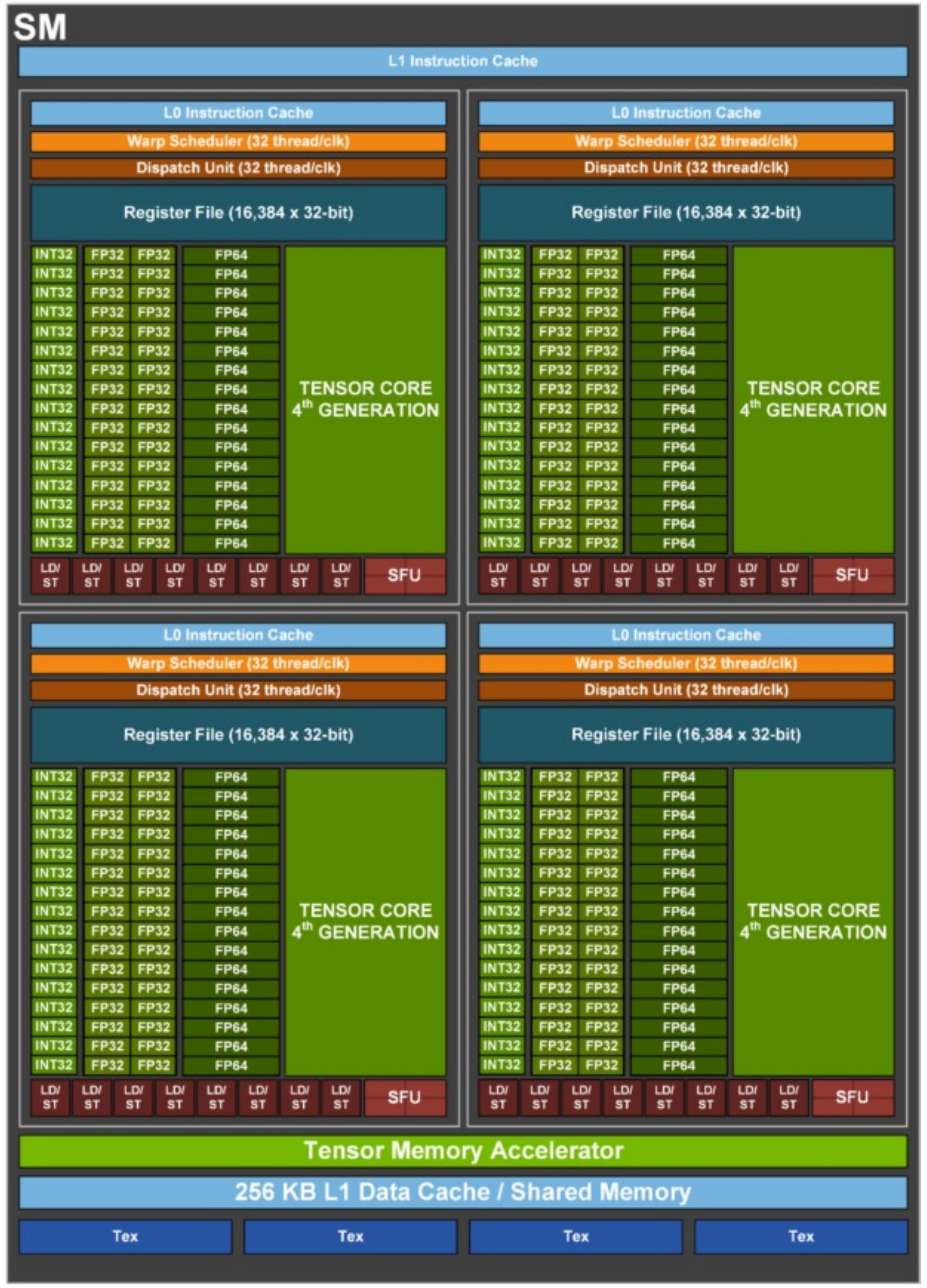
Table of Contents

Streaming Multiprocessor

The processor that performs computations in NVIDIA architectures is the Streaming Multiprocessor. It consists of:

- Arithmetic (green)
- Load / store (red)
- Memory (blue)
- Control units (orange)

There are many SMs on a GPU, current models have up to 132 SMs and thousands of *CUDA cores*.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Recap

Programming model

- Kernel: `__global__` functions.
- Blocks: Subdivision of work into groups.
- Thread: Unit of work.

- Local variables.
- Shared memory: Small and fast memory.
- Global memory: Large and slow memory.

Underlying hardware

- The CUDA scheduler.
- Streaming multiprocessors.
- Warp of 32 threads.

- Registers.
- Cache (L1, L2).
- DRAM memory.

The Warp

The SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.

Threads inside a warp:

- start at the same program address.
- have their own program counter (instruction address counter).
- have their own register state.

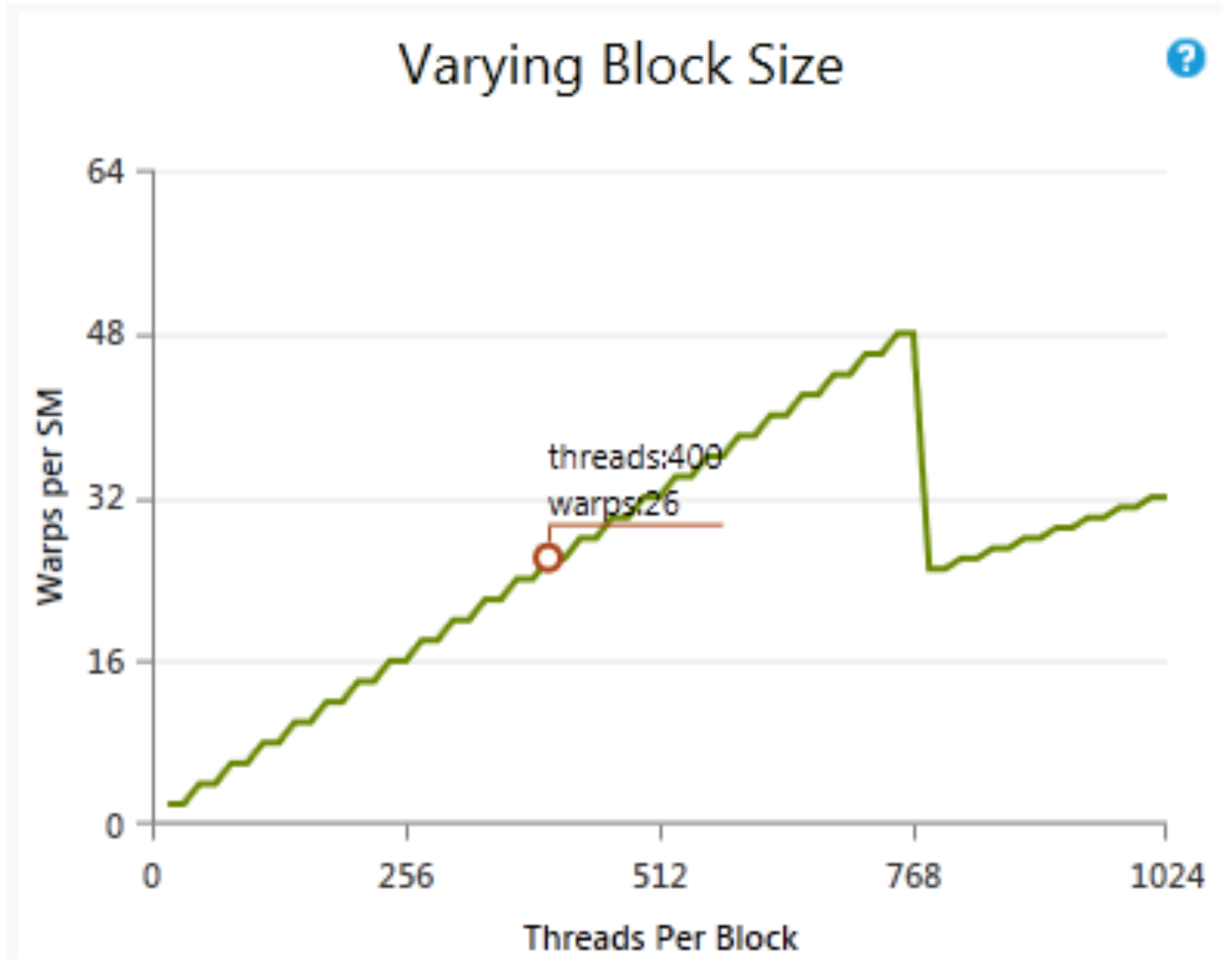
Does Warp Size Matter?

Warp size affects what block size configurations fully occupy SMs. It affects *occupancy*:

Occupancy is the ratio of active warps to maximum supported active warps in a SM.

Example: On a GPU that supports **64** active warps per SM, full occupancy on a SM can be achieved with:

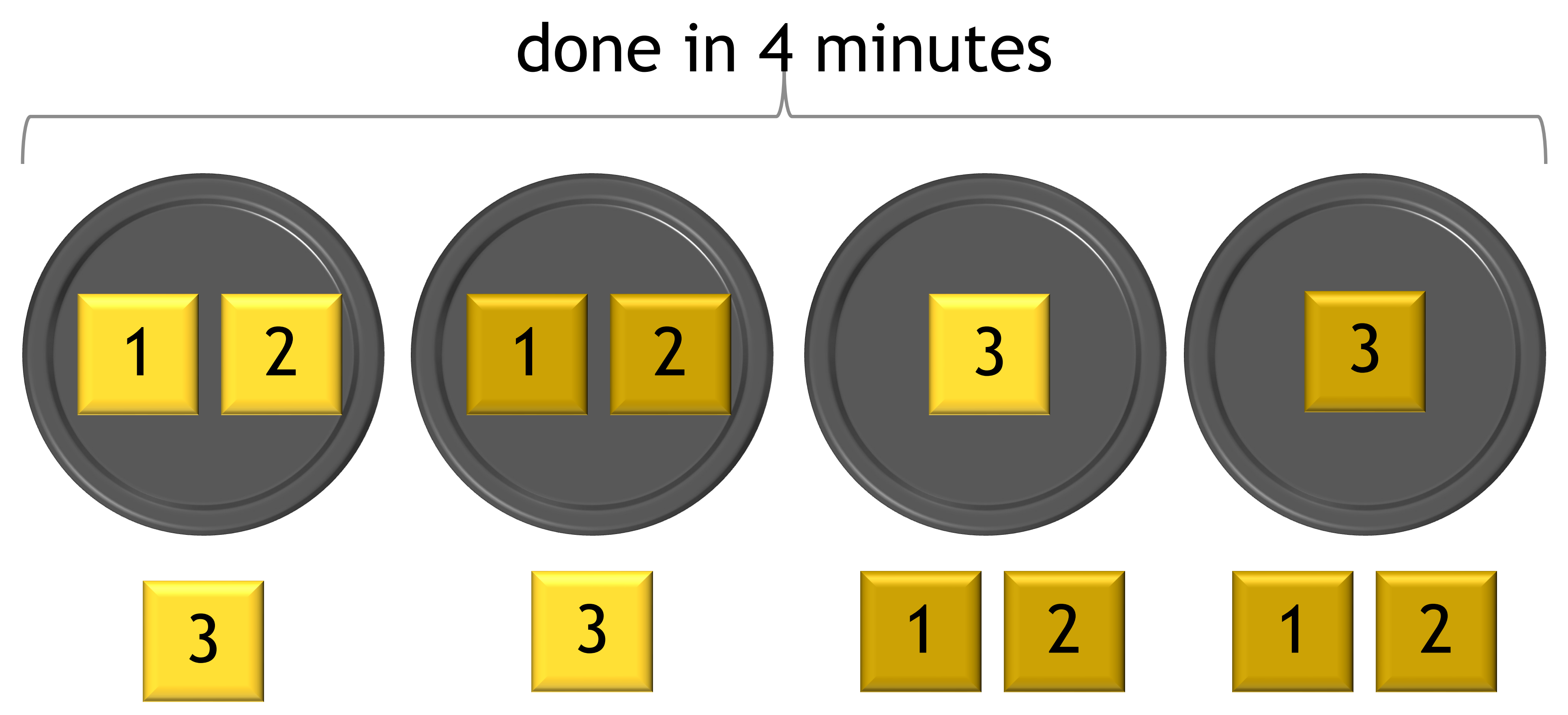
- 8 active blocks with 256 threads per block.
- 16 active blocks with 128 threads per block.
- 32 active blocks with 64 threads per block.



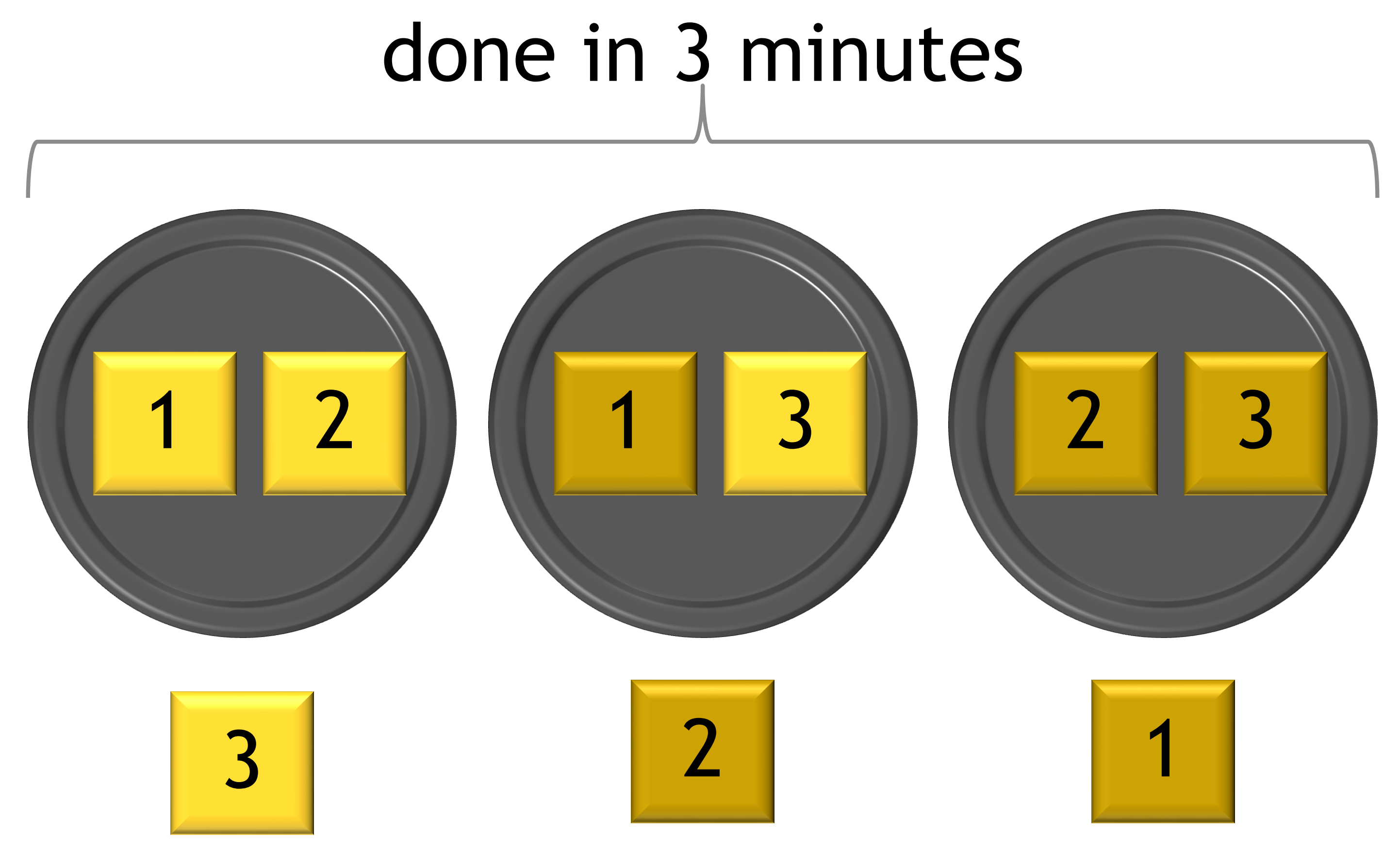
Is 100% occupancy the best?

Latency vs Throughput

- Full occupancy may not be desirable in scenarios where latency is the relevant metric.
- Efficiently using the GPU resources sometimes means going for lower configurations.
  - Profilers in particular will always suggest methods to improve **latency**, not throughput.
- See this presentation for more on this topic.
- Toasting – do you want the best **time-to-toast** or the best **toasts-per-second**?



Optimized for individual toast latency.



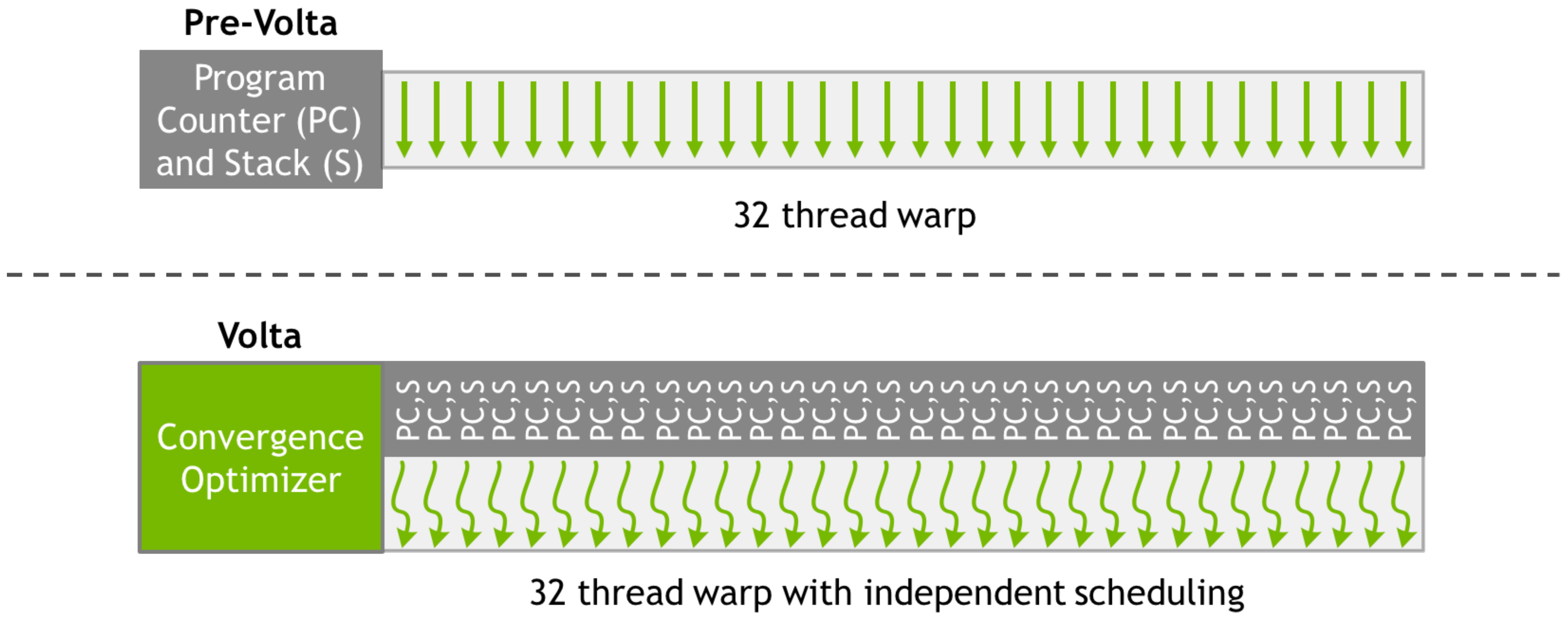
Optimized for overall throughput.

Lockstep

Older architectures executed in *lockstep*, they shared program counter and register state.

However, this *warp-synchronous assumption* is not valid anymore.

Threads can now branch and execute independently.

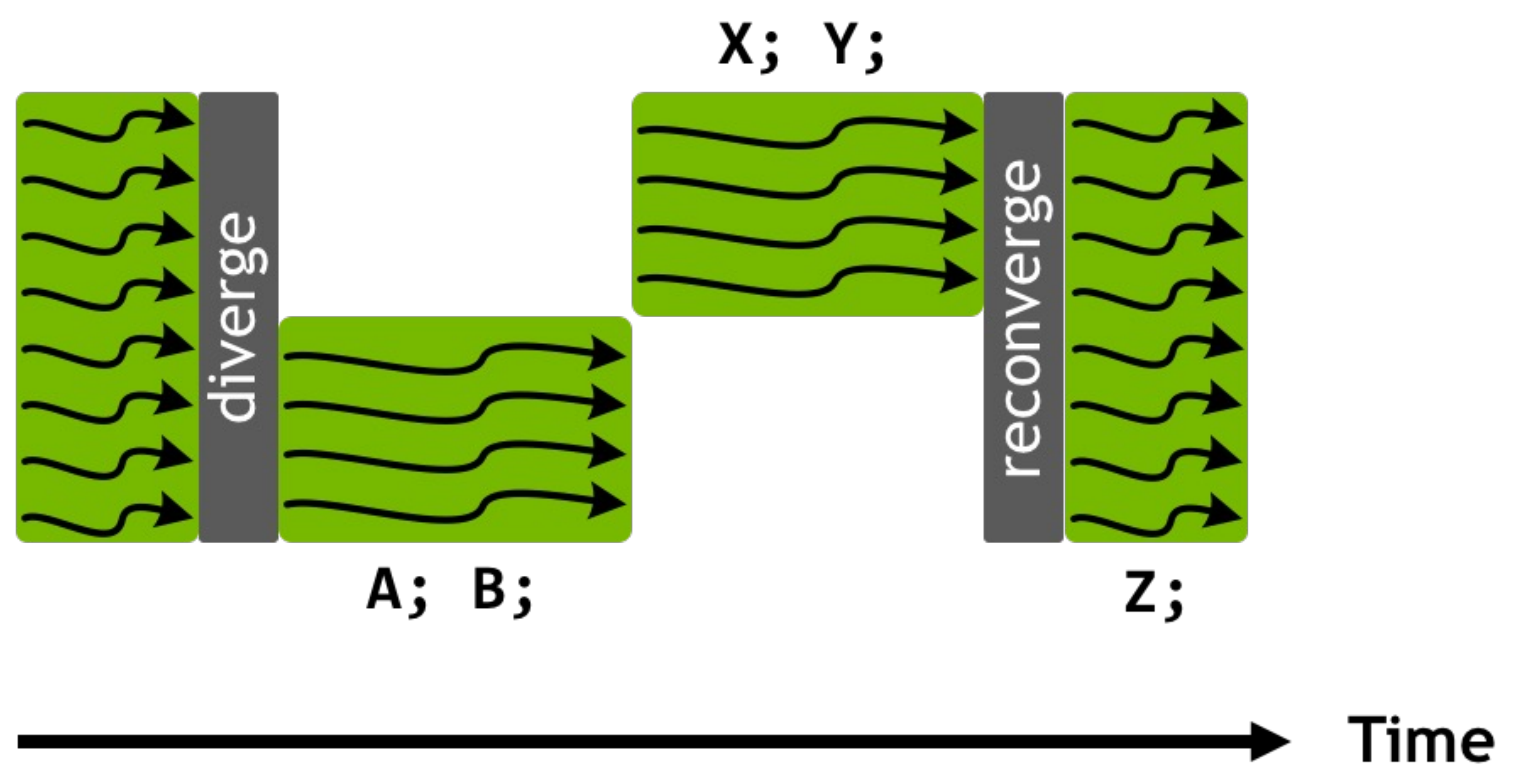

Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

What Happens With Branches?

If you are working on a GPU that runs in lockstep, if there is at least one thread running the branch then the whole warp will go through the branch.

For this reason, it is commonly said that one should avoid branches when writing GPU code:

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



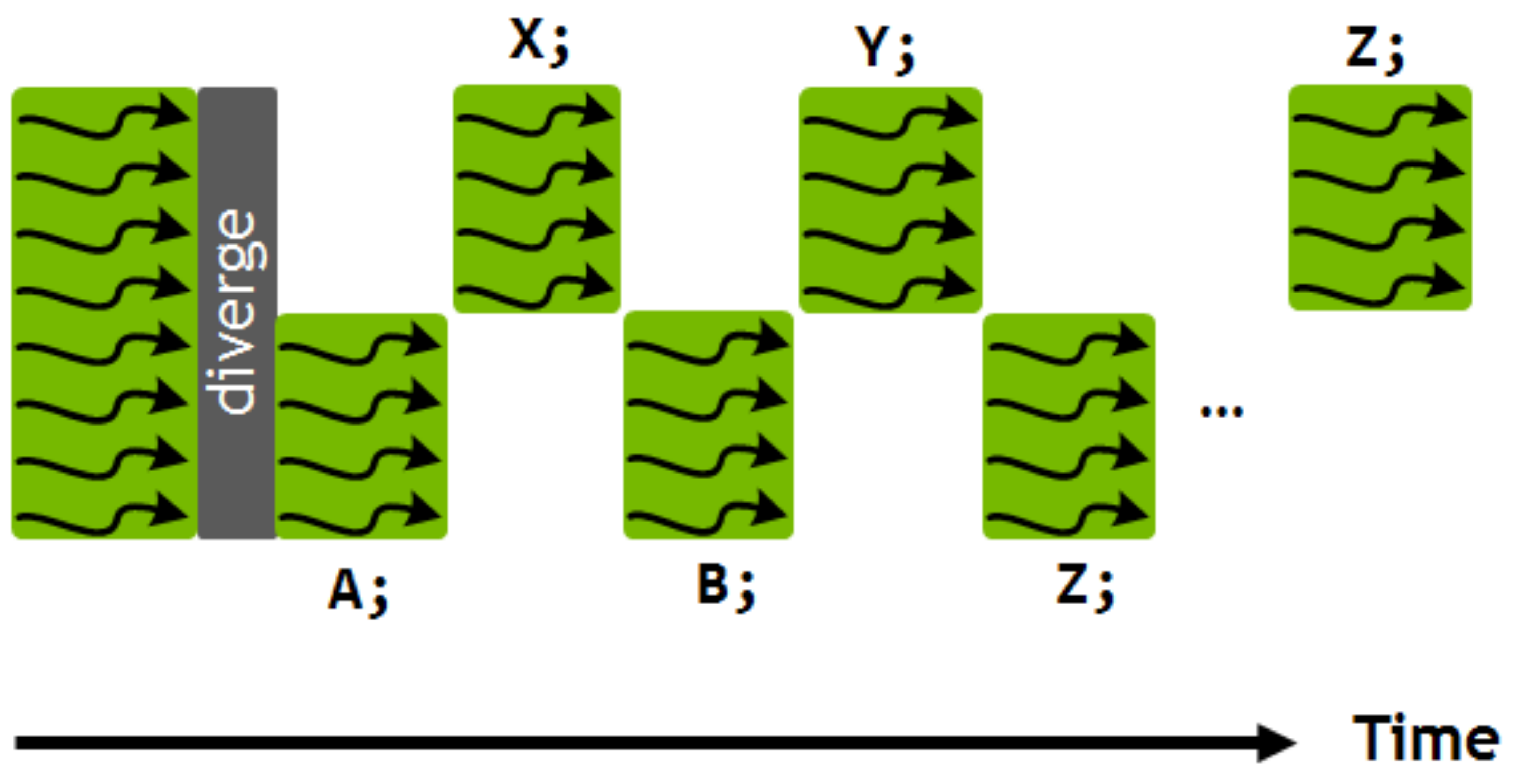
Branchless Code

As compilers get smarter and GPUs do not execute in lockstep, it is not the case anymore that one should avoid branches at any cost. *In most cases, branches are ok*.

In recent models, threads within a warp are **scheduled independently**:

- Execution of statements can be interleaved.
- At one clock cycle, one single same instruction is executed for all threads in a warp (SIMT).

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



Are Branches Relevant?

When branches lead to homogeneous code then it is worth removing the branch. Especially if the code behind the branch is a hot section and complex for the compiler. In essence:

Avoid long sequences of diverged execution by threads within the same warp.

For instance, given a seed of a particle trajectory find compatible hits in other sensors:

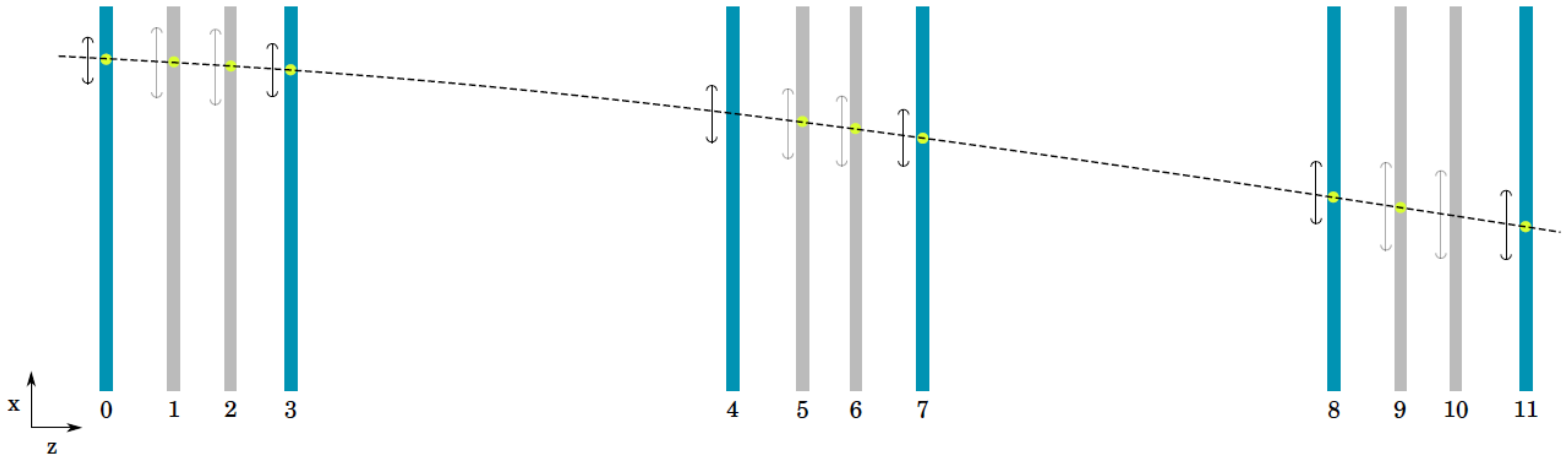


Table of Contents

Debugging

GPU code can be debugged in a similar way to how CPU code is debugged.

There are several tools that can be used for debugging GPU program's execution:

- `cuda-gdb` – Command line debugger that is based off the popular gdb. It can be used to debug CUDA applications, set watchpoints, step into execution of any thread and so on.
- **NVIDIA Nsight** – Nsight is both an extension to Visual Studio and an extension to the Eclipse environment that adds CUDA support. The Visual Studio version is the better of the two, and it contains a built-in debugger and profiler. It is fully integrated with the IDE, so breakpoints can be set, values can be expanded, just like with the CPU debugger.

Profiling

Similarly, for profiling there are various tools to check out:

- `nsys` – Command line profiler that replaces the previous nvprof. It is highly configurable and can produce analytics that can be analyzed with the visual profiler.
- **Nsight Systems, Nsight Compute** – These tools provide complementary analytics and functionalities to optimize your application. It is also possible to connect remotely to a server where the application is run, results are collected and presented in the local profiler instance.

Nsight Compute Profile

Profile with CLI example

```
$ ncu -o kernel_prof \ # Output filename
      -f \             # Overwrite file if exists
      -c 1 \           # Kernel name to profile
      -s 100 \         # Launch count
      --set-full \     # Launches to skip with match kernel filter (name)
      ./executable     # Execution command
```

https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

File
Connection
Debug
Profile
Tools
Window
Help
Connect
Disconnect
Terminate
Profile Kernel
Baselines
Interactive stepping
Connect to remote host
Occupancy Calculator
Add baseline to compare
Profiler Sections (expand for charts)
API Stream
Untitled 1 *
q72_noinline.ncu-rep
Untitled 2 *
Add Baseline
Apply Rules
Occupancy Calculator
Copy as Image
Sections/Rules Info
API Statistics
NVTX
Resources

API Stream

62854 > unaligned

Next Trigger: Enter trigger

ID	API Name	Details
598	transposeNaive	transposeNaive
599	cudaDeviceSynchronize	
600	cuCtxSynchronize	
601	cudaLaunchKernel	
602	cuLaunchKernel	
603	transposeCoalesced	transposeCoalesced
604	cudaDeviceSynchronize	
605	cuCtxSynchronize	
606	cudaLaunchKernel	
607	cuLaunchKernel	
608	transposeCoalescedN...	transposeCoalescedNoCon
609	cudaDeviceSynchronize	
610	cuCtxSynchronize	
611	cudaLaunchKernel	
612	cuLaunchKernel	
613	strided	strided
614	cudaDeviceSynchronize	
615	cuCtxSynchronize	
616	cudaLaunchKernel	
617	cuLaunchKernel	
618	unaligned	unaligned

Sections/Rules Info

Section Sets

Enter filter

	Name	Sections
	<custom>	See Sections/Rules \
	default	LaunchStats,Occupa
	roofline	SpeedOfLight,Speed
	source	SourceCounters
	detailed	ComputeWorkloadAr
✓	full	ComputeWorkloadAr

Page: Details Result: 3 - 608 - transposeCoalescedNoConflicts

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	608 - transposeCoalescedNoConflicts (100000, 1, 1)x(32, 32, ...	1.42 msecond	1,693,633	16	0 - NVIDIA TITAN V	1.19 cycle/nsecond	7.0	[62854] demo

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	18.90	Duration [msecond]	1.42
Memory Throughput [%]	88.51	Elapsed Cycles [cycle]	1693633
L1/TEX Cache Throughput [%]	23.62	SM Active Cycles [cycle]	1682002.62
L2 Cache Throughput [%]	33.15	SM Frequency [cycle/nsecond]	1.19
DRAM Throughput [%]	88.51	DRAM Frequency [cycle/usecond]	846.78

High Throughput — The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the Memory Workload Analysis section.

Roofline Analysis — The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for mode details on roofline analysis.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	0.59	SM Busy [%]	14.87
Executed Ipc Active [inst/cycle]	0.59	Issue Slots Busy [%]	14.87
Issued Ipc Active [inst/cycle]	0.59		

Low Utilization — All compute pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	575.62	Mem Busy [%]	33.15
L1/TEX Hit Rate [%]	0	Max Bandwidth [%]	88.51
L2 Hit Rate [%]	50.00	Mem Pipes Busy [%]	18.90

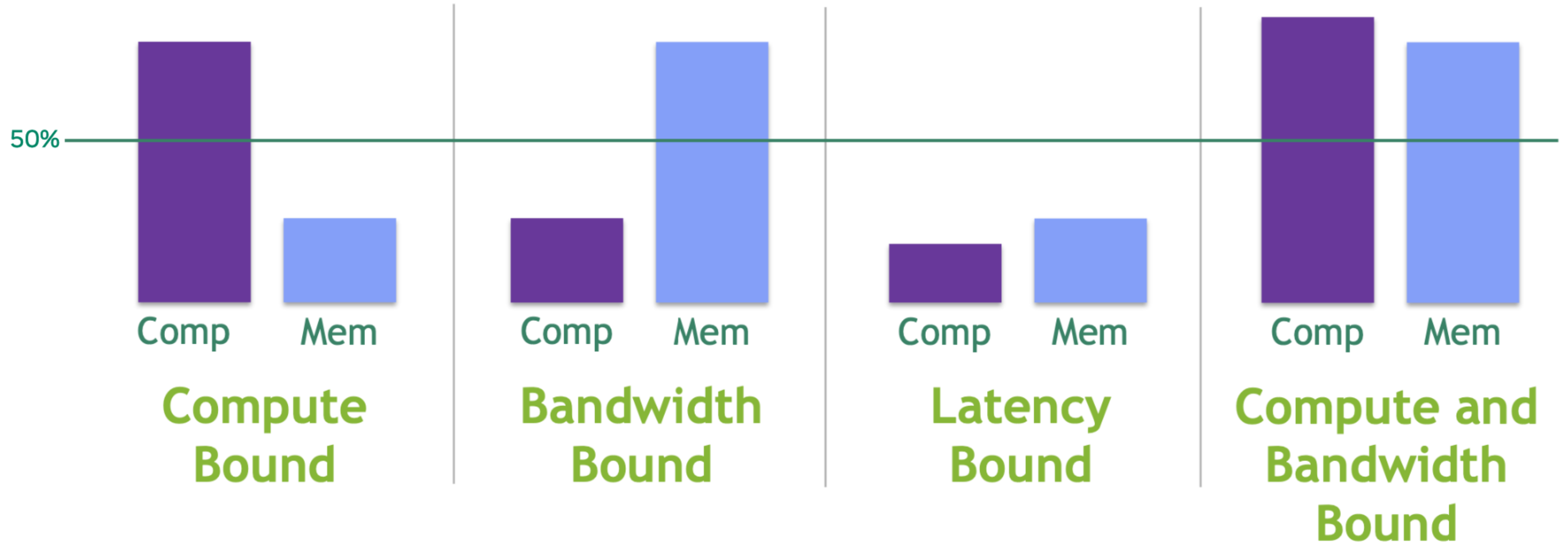
Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	14.52	No Eligible [%]	85.07

Applied rule "Thread Divergence" (0 new results)

GPU *Speed Of Light* Throughput



Nsight Compute

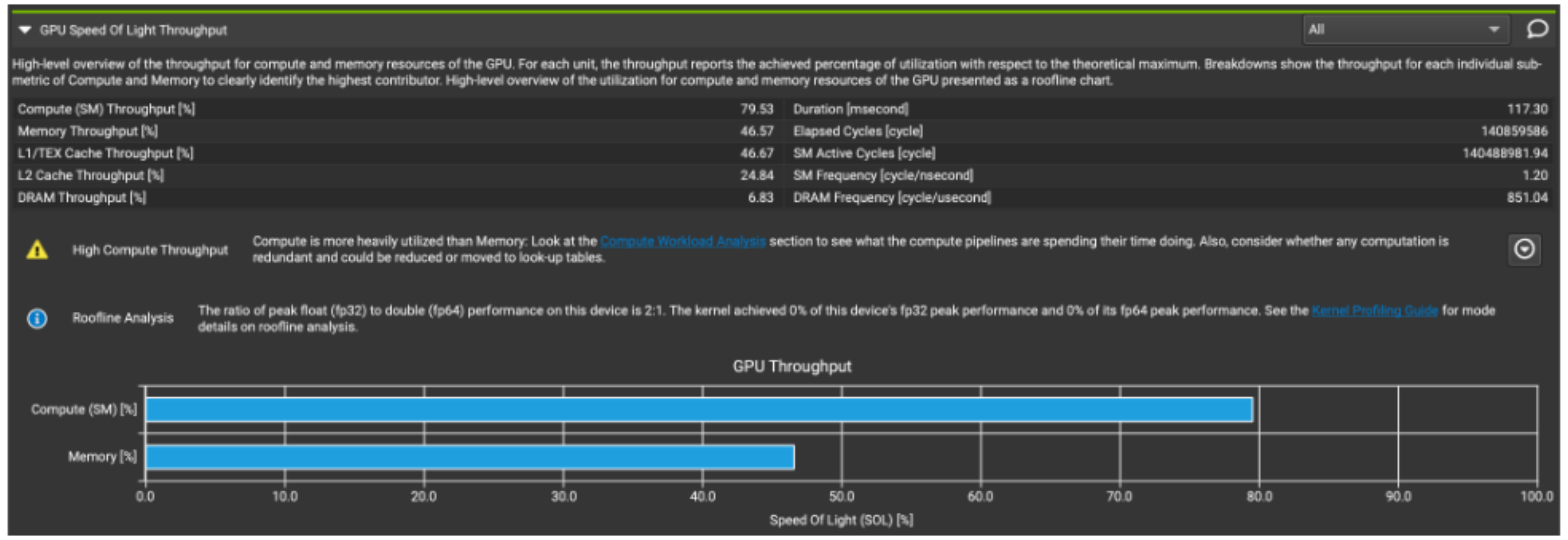
Analysis workflow

Start Here
GPU Speed of Light
Compute bound
Compute Workload Analysis
Understand Limiter
Instruction Statistics
Memory bound
Memory Workload Analysis
Latency bound
Scheduler & Warp State Statistics
Understand Limiter
Occupancy & Launch Statistics

SOL Section

Start here

- **Speed Of Light (SOL)** chart presents a high-level overview of compute and memory utilization of the GPU. For each unit, it reports the achieved % of utilization with respect to its theoretical maximum.



GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

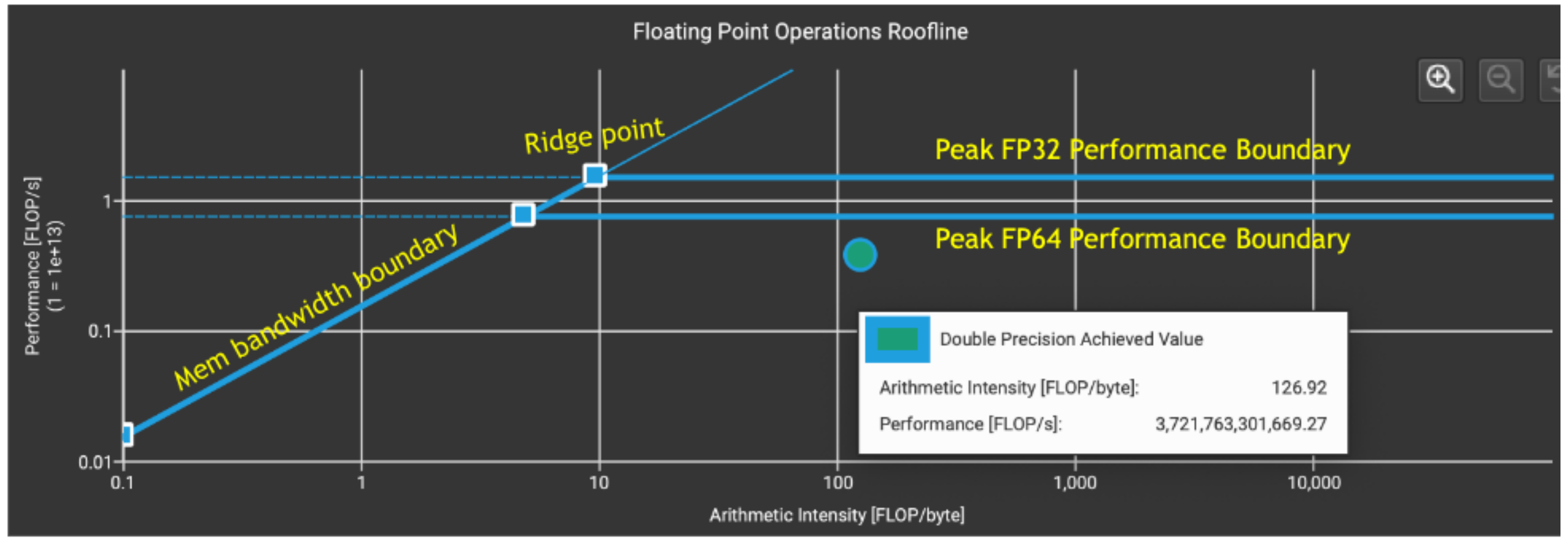
Compute (SM) Throughput [%]	79.53	Duration [msecond]	117.30
Memory Throughput [%]	46.57	Elapsed Cycles [cycle]	140859586
L1/TEX Cache Throughput [%]	46.67	SM Active Cycles [cycle]	140488981.94
L2 Cache Throughput [%]	24.84	SM Frequency [cycle/nsecond]	1.20
DRAM Throughput [%]	6.83	DRAM Frequency [cycle/usecond]	851.04

High Compute Throughput — Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

Roofline Analysis — The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for mode details on roofline analysis.

- **Roofline** chart shows your achieved FLOPS with respect to the theoretical maximum single-precision or double-precision FLOPS.

  Arithmetic Intensity = Compute / Memory
  = FLOPS / Byte



🟩 Double Precision Achieved Value	
Arithmetic Intensity [FLOP/byte]:	126.92
Performance [FLOP/s]:	3,721,763,301,669.27

Memory Workload Analysis

Logical Units vs Physical Units

L1TEX Global Store Access Pattern — The memory access pattern for global stores in L1TEX might not be optimal. On average, this kernel accesses 4.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 32.0 sectors per request, or 32.0*32 = 1024.0 bytes of cache data transfers per request. The optimal thread address pattern for 4.0 byte accesses would result in 4.0*32 = 128.0 bytes of cache data transfers per request, to maximize L1TEX cache performance.Check the Source Counters section for uncoalesced global stores.

L2 Store Access Pattern — The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

Show As: Transfer Size

Table of Contents

Summary

- Prefer coalesced memory accesses.
- Shared memory is a valuable resource. Use it, don’t abuse it.
- Tiling is a technique that helps optimize memory performance.

- Use streams to optimize GPU usage.
- Pipelines with three or more strems yield best results.

- Warp size is a hardware detail that affects efficient block sizes.
- Avoid branches but don’t go paranoid.
- Profile, profile, profile.

Resources Used in the Talk

- GPU Teaching Kit on Accelerated Computing.
- NVIDIA Deep Learning Institute materials.
- Talk on NVIDIA Profiling Tools by Jeff Larkin.
- GPU Profiling for Inference by William Raveane.