Introduction to GPU Programming with CUDA

Orian Louant

LIÈGE université

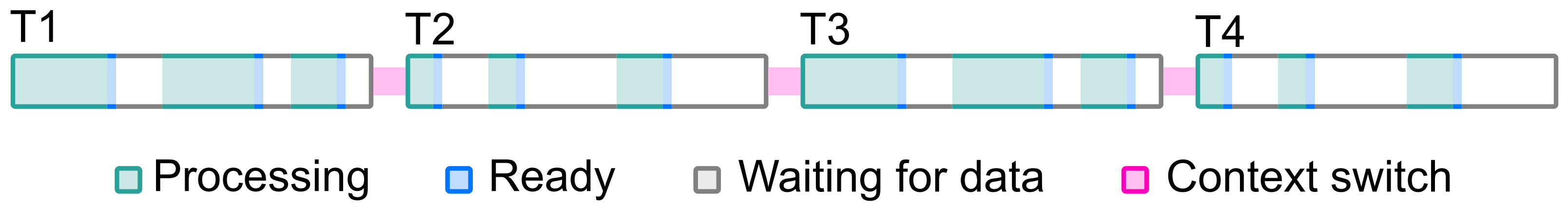
INFO0939: High Performance Scientific Computing

Fall 2025

CPU *vs* GPU execution model

CPUs and GPUs are both powerful processors but they're optimized for very different goals. CPUs use a **latency-optimized** design:

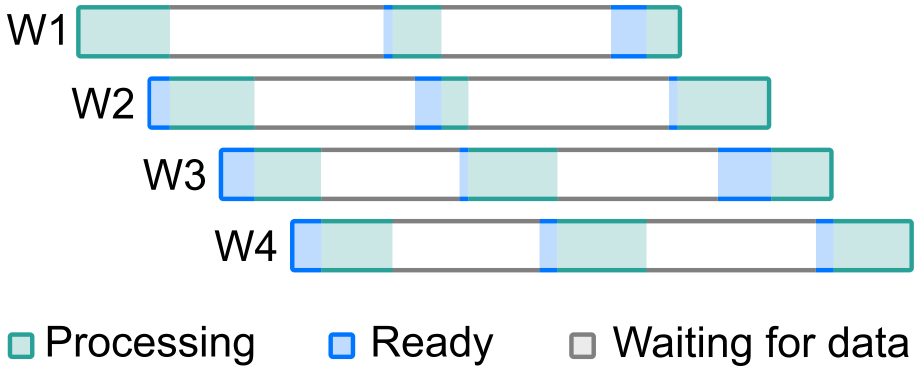
- CPUs are built to minimize the time per task, i.e., latency. They have a few complex cores that run instructions sequentially but very fast.
- Each core uses pipelining, large caches, sophisticated branch predictors and out-of-order execution
- CPUs are general purpose, ideal for control-heavy or diverse workloads where each thread does something different



CPU *vs* GPU execution model

CPUs and GPUs are both powerful processors but they're optimized for very different goals. GPUs use a **throughput-optimized** design:

- GPUs are built to maximize throughput, with thousands of simple cores executing many threads in parallel
- When one group of threads stalls waiting for memory, the GPU quickly switches to another ready group, hiding latency and keeping work flowing
- GPUs are perfect for data-parallel workloads



CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA’s platform for parallel computing.

- By extending the standard C/C++, it lets developers write programs that run directly on the GPU, using its thousands of lightweight cores to perform many operations at once
- CUDA exposes a programming model where you launch large numbers of threads organized into blocks and grids
- CUDA is a proprietary platform, so executables run only on NVIDIA GPUs. AMD offers HIP (Heterogeneous-computing Interface for Portability), a near drop-in replacement that lets most CUDA code run on AMD hardware with minimal changes

A First CUDA Application

CUDA Hello World

A CUDA program is composed of two components that run in distinct execution environments:

- **The host side:** the code that runs on the CPU, including calls to the CUDA runtime and kernel launches
- **The device side:** the code (kernel) that executes on the GPU

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void hello_kernel()
{
  printf("Hello from GPU thread %d out of %d "
         "of block %d out of %d\n",
         threadIdx.x, blockDim.x, blockIdx.x, gridDim.x);
}

int main(int argc, char* argv[])
{
  const int num_threads = 4;
  const int num_blocks = 2;

  hello_kernel<<<num_blocks, num_threads>>>();

  cudaDeviceSynchronize();

  return 0;
}
```

CUDA kernel: the `__global__` function qualifier

A CUDA kernel is a function defined using `__global__` function qualifier. This qualifier defines a function that runs on the GPU but is called from the CPU.

When a CUDA compiler encounters a function marked with `__global__`, it compile the code in two phases: one for the host (CPU) and one for the device (GPU). The result is

- A host-side stub function that sets up the kernel launch parameters
- A device-side kernel function compiled into PTX (or SASS) that the GPU executes

`__global__` functions cannot be called directly like normal C/C++ functions, they must be launched with the special *triple chevrons* syntax :

```
kernel<<<grid, block>>>(args);
```

CUDA kernel: the grid of blocks of threads

In CUDA, when you launch a kernel (a `__global__` function), you don't just call it once but launch many parallel threads organized into a hierarchy:

- **Block:** a group of threads that can cooperate. Threads in the same block can share data through fast shared memory and can synchronize
  - Indexes of the thread: `threadIdx.x`, `threadIdx.y`, `threadIdx.z`
  - Dimensions of the block: `blockDim.x`, `blockDim.y`, `blockDim.z`
- **Grid:** the collection of all blocks launched for one kernel call. Blocks in the same grid cannot directly synchronize or share memory
  - Indexes of the block: `blockIdx.x`, `blockIdx.y`, `blockIdx.z`
  - Dimensions of the grid: `gridDim.x`, `gridDim.y`, `gridDim.z`

Multidimensional grid of blocks of threads

A grid of block of threads can have up to 3 dimensions. A 1-dimensional grid can be created by providing integers values at kernel launch:

```
kernel<<<num_blocks, num_threads>>>(args)
```

For multidimensional grids, CUDA provide the `dim3` type:

2-dimensional grid

```
dim3 grid(num_block_x, num_block_y)
dim3 block(num_threads_x, num_threads_y)
kernel<<<grid, block>>>(args)
```

3-dimensional grid

```
dim3 grid(num_block_x, num_block_y, num_block_y)
dim3 block(num_threads_x, num_threads_y, num_threads_z)
kernel<<<grid, block>>>(args)
```

CUDA device function: the `__device__` function qualifier

In CUDA, the `__device__` qualifier marks a function that executes on the GPU and can only be called from other GPU code. Such functions can be invoked from within a GPU kernel or another `__device__` function, whereas host functions (without this qualifier) cannot be called from device code.

```
__device__ void print_thread_info(int tidx, int blkdim, int blkidx, int gdim) {
  printf("Hello from GPU thread %d out of %d of block %d out of %d\n",
         tidx, blkdim, blkidx, gdim);
}

__global__ void hello_kernel() {
  print_thread_info(threadIdx.x, blockDim.x, blockIdx.x, gridDim.x);
}
```

Compiling a CUDA application

On Lyra, CUDA applications can be built with either the NVIDIA-provided `nvcc` compiler or the LLVM-based `clang` compiler

Using NVDIA compiler

```
module load CUDA
nvcc -o <EXECUTABLE> --gpu-architecture sm_<CC> <SOURCE>
```

Using LLVM Clang

```
module load Clang
clang++ -o <EXECUTABLE> --cuda-gpu-arch=sm_<CC> <SOURCE> \
                 -lcudart -ldl -lrt -pthread
```

where `<SOURCE>` is the CUDA source file, `<EXECUTABLE>` is the name of the output program you want to generate, and `<CC>` specifies the target compute capability

Compiling a CUDA application

CUDA source code, that is, code containing CUDA-specific constructs such as kernels (`__global__` functions) or kernel launch syntax (`<<< >>>`) — must be compiled as C++ source code

- The standard file extension for CUDA source files is `.cu`. When this extension is used, the compiler automatically recognizes the file as CUDA code and compiles it appropriately using both the host C++ compiler and the device compiler
- If a CUDA source file uses a nonstandard extension (such as `.c`, `.cpp`, or `.cc`), the compiler will not automatically treat it as CUDA code. In that case, you can explicitly tell the compiler to interpret it as CUDA source by using the `-x cu` compiler flag.

```
nvcc -x cu source_code.cpp -o output_executable
```

Compiling a CUDA application: compute capability

Every NVIDIA GPU has a compute capability, often written as a pair of numbers like 8.0 or 8.9. It's essentially the GPU's version number from the CUDA compiler's perspective, defining what hardware instructions, memory types, and features the device supports. You can use the CUDA `deviceQuery` demo to determine the compute capability of a GPU. For example, on Lyra

```
module load CUDA
$EBROOTCUDA/extras/demo_suite/deviceQuery | grep Capability
```

This produces output such as:

```
CUDA Capability Major/Minor version number:    8.9
```

This value corresponds to a compiler flag `--gpu-architecture=sm_89` for the NVIDIA compiler, or `--cuda-gpu-arch=sm_89` when using LLVM Clang

🔗 Technical Specifications per Compute Capability

Compile and run the Hello World

Now, we can compile the `hello_word.cu` source file with

```
module load CUDA
nvcc --gpu-architecture sm_89 -o hello_world hello_world.cu
```

and then run the executable:

```
./hello_world
Hello from GPU thread 0 out of 4 of block 0 out of 2
Hello from GPU thread 1 out of 4 of block 0 out of 2
Hello from GPU thread 2 out of 4 of block 0 out of 2
Hello from GPU thread 3 out of 4 of block 0 out of 2
Hello from GPU thread 0 out of 4 of block 1 out of 2
Hello from GPU thread 1 out of 4 of block 1 out of 2
Hello from GPU thread 2 out of 4 of block 1 out of 2
Hello from GPU thread 3 out of 4 of block 1 out of 2
```

The GPU kernel `hello_kernel` was launched with a grid of 2 blocks, each containing 4 threads, for a total of 8 threads

Run the Hello World with SLURM

With SLURM, allocating a GPU is done using the `--gpus=<NGPUS>` option. A simple job script to run our *Hello World* CUDA example would look like this:

```bash
#!/bin/bash
#
#SBATCH --job-name="CUDA Hello World"
#SBATCH --ntasks=1
#SBATCH --gpus=1
#SBATCH --time=01:00
#SBATCH --output="cuda_hello_world.out"

module load CUDA

./hello_world
```

A note about kernel execution

CUDA kernel launches are asynchronous with respect to the host: the call returns immediately and the host continues execution without waiting for the device to finish

Synchronization may be forced using `cudaDeviceSynchronize()` which blocks until the device has completed all preceding requested tasks

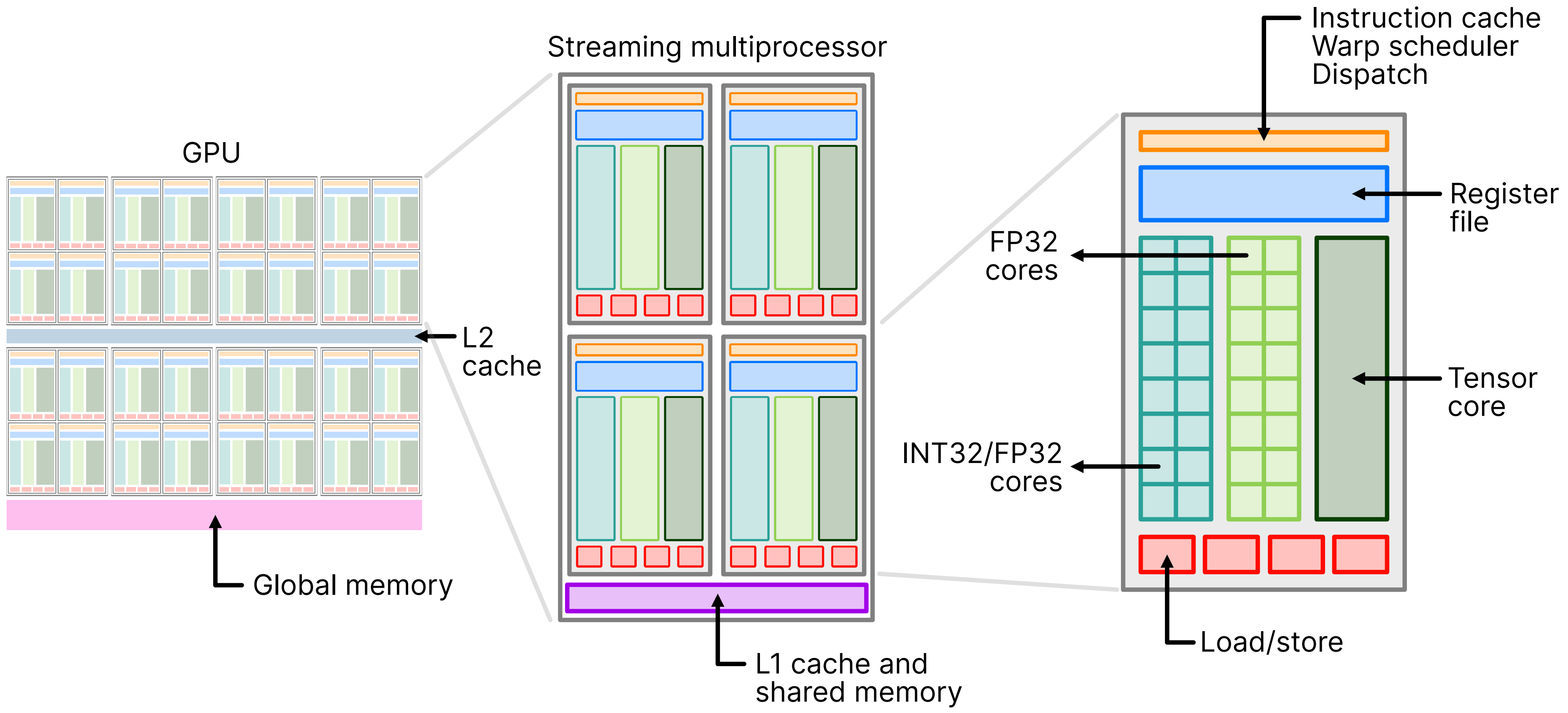
```
__host__ __device__ cudaError_t cudaDeviceSynchronize(void)
```

If the host code depends on GPU results, explicit synchronization may be required. We will see later that some operations, like memory copies may be synchronous

When a CUDA program ends the CUDA runtime automatically performs an implicit device synchronization before shutting down the context

CUDA Execution Model

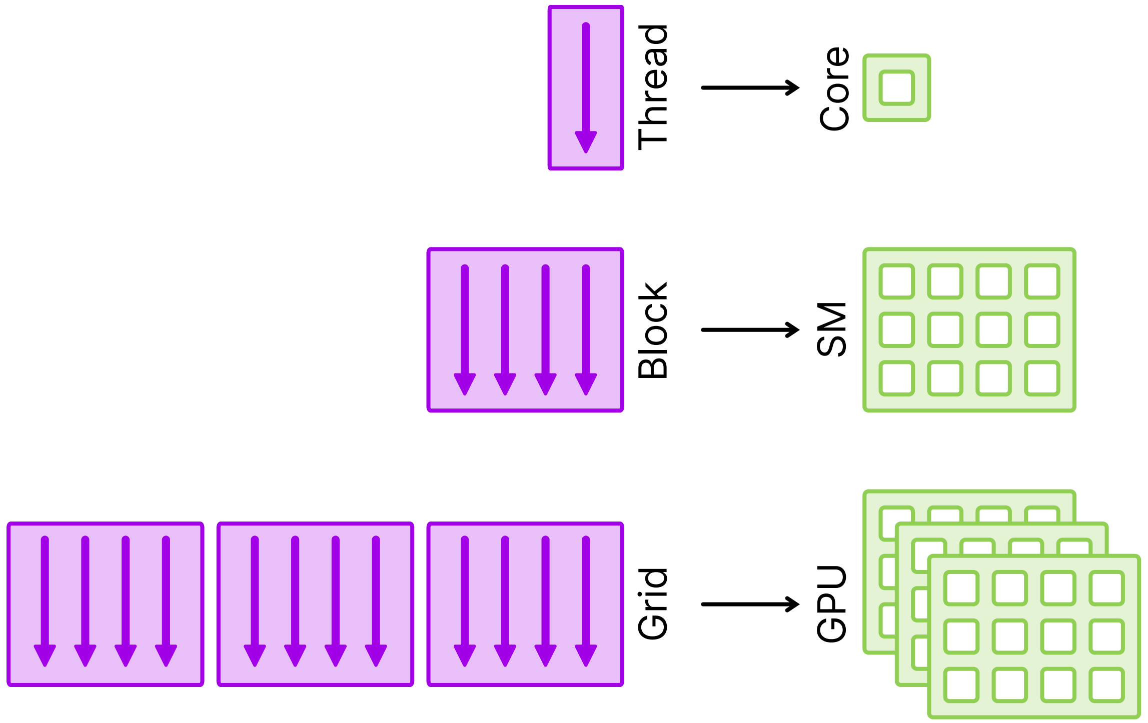
NVIDIA GPUs: hardware overview



Threads, blocks, grid: hardware mapping

The concepts of threads, blocks, and grids in CUDA are directly mapped to the underlying GPU hardware components:

- Each thread executes instructions on a CUDA core
- Threads blocks are scheduled to run on one Streaming Multiprocessor (SM)
- A grid consists of multiple blocks, which are distributed across all SMs in the GPU



Lyra GPUs: NVIDIA RTX 6000 Ada

Lyra features 40 nodes, each with one NVIDIA RTX 6000 Ada GPU with

- 48 GB of GDDR6 memory (960 GB/s)
- 96 MB of L2 cache
- 142 streaming multiprocessors (SMs)
    - 64K 32-bit registers register file
    - 128 KB of L1 cache from which up to 100 KB can be used for shared memory
    - 128 FP32 cores and 2 FP64 cores

Theoretical FP32 peak performance:

$$128 \text{ cores} \cdot 2 \text{ FMA FLOPs} \cdot 142 \text{ SMs} \cdot 2505 \text{ MHz clock} = 91.06 \text{ TFLOPS}$$

Theoretical FP64 peak performance:

$$2 \text{ cores} \cdot 2 \text{ FMA FLOPs} \cdot 142 \text{ SMs} \cdot 2505 \text{ MHz clock} = 1.42 \text{ TFLOPS}$$

Lucia GPUs: NVIDIA A100

Lucia features 50 GPU nodes, each with 4 NVIDIA A100 GPUs with

- 40 GB of HBM2e memory (1.460 TB/s)
- 40 MB of L2 cache
- 108 streaming multiprocessors (SMs)
    - 64K 32-bit registers register file
    - 192 KB of L1 cache from which up to 164 KB can be used for shared memory
    - 32 FP64 cores and 64 FP32 cores

Theoretical FP32 peak performance:

$$64 \text{ cores} \cdot 2 \text{ FMA FLOPs} \cdot 108 \text{ SMs} \cdot 1410 \text{ MHz clock} = 19.49 \text{ TFLOPS}$$

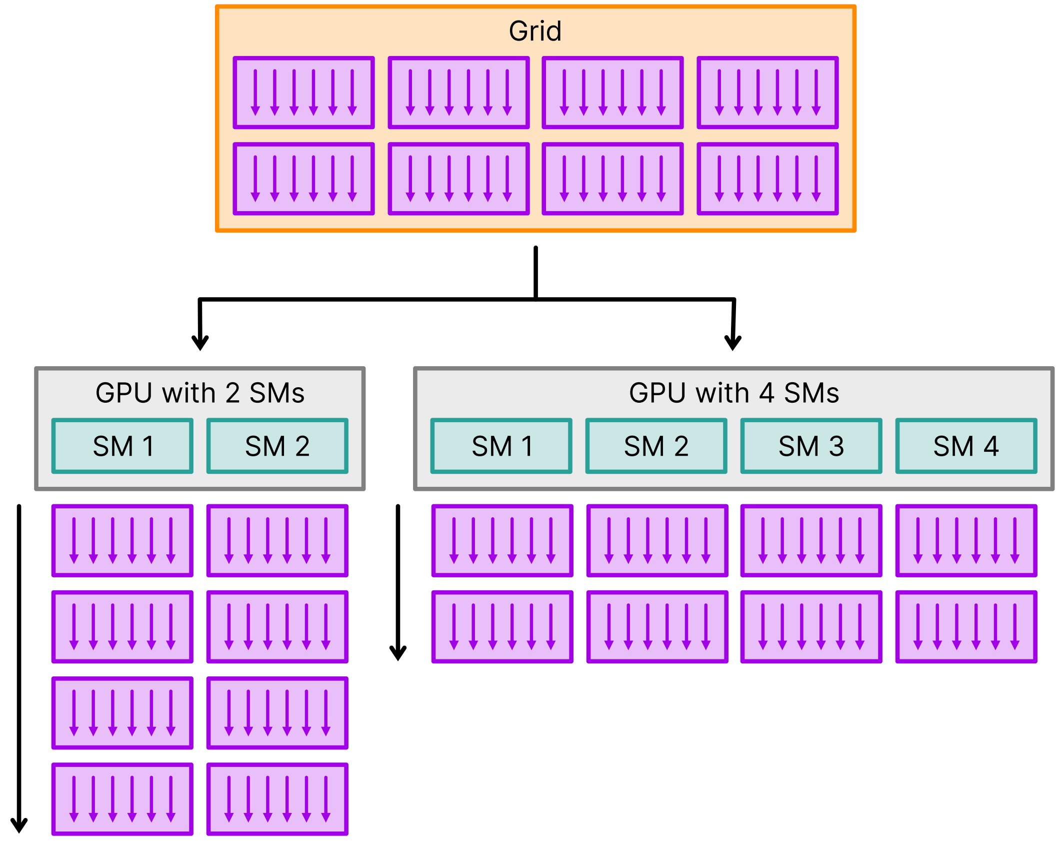
Theoretical FP64 peak performance:

$$32 \text{ cores} \cdot 2 \text{ FMA FLOPs} \cdot 108 \text{ SMs} \cdot 1410 \text{ MHz clock} = 9.75 \text{ TFLOPS}$$

Threads, blocks, grid: flexibility

The CUDA programming model allows the GPU architecture to span a wide market range by scaling the number of multiprocessors:

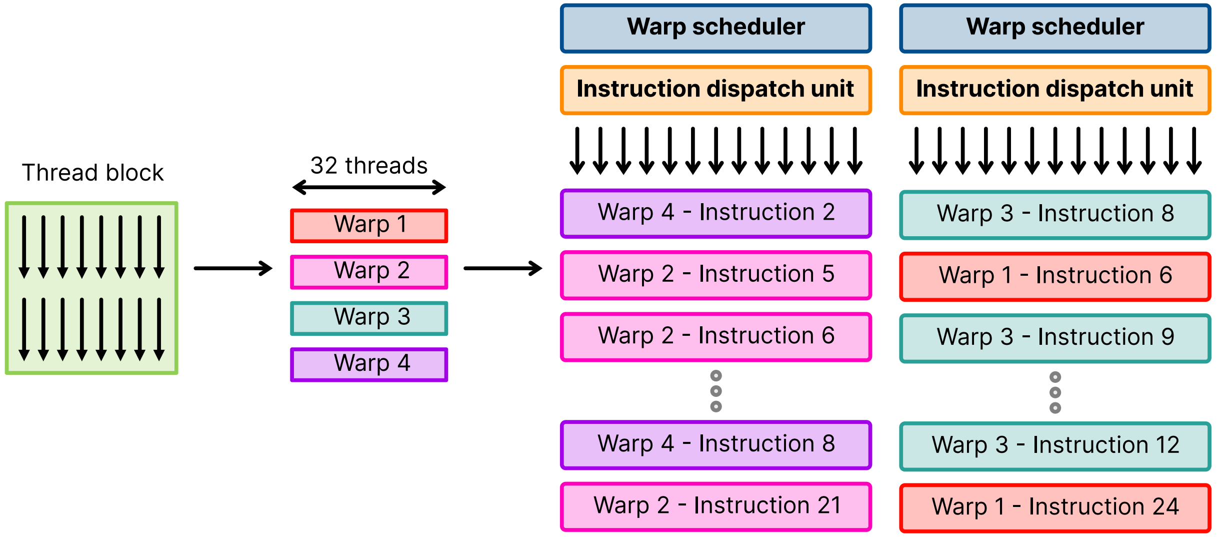
- Thread blocks can run on any available GPU multiprocessor and the runtime system alone manages the actual number of multiprocessors
- The same CUDA code can scales across a wide range of GPUs, from high-end HPC or workstation GPUs to mainstream gaming GPUs



Single instruction multiple thread

The basic scheduling unit on NVIDIA GPUs is a **warp**, a group of threads (typically 32 threads) that execute the same instruction simultaneously on a GPU

- All active threads in a warp execute the same instruction
- Each thread has its own registers and can access different data

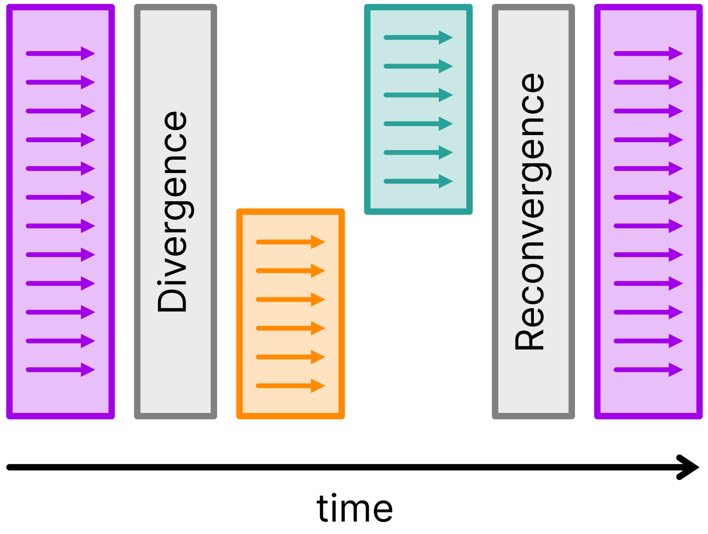


The CUDA execution model is commonly referred to as Single Instruction, Multiple Threads (SIMT).

Branching and thread divergence

On GPU it's recommended to avoid branching. When threads in a program take different code paths (e.g., through `if` or `else`), threads within the same warp may diverge.

- If threads diverge, the GPU must serialize the different paths, executing one branch while masking off threads not taking it
- After all paths complete, threads reconverge to continue execution together



The consequence is a reduced parallel efficiency whith some threads staying idle while others run.

Memory and Data Management

Host and device memory

Let's consider this simple vector addition kernel:

```
__global__ void vector_add_kernel(float* a, float* b, float* c) {
  const size_t idx = blockIdx.x * blockDim.x + threadIdx.x;

  c[idx] = a[idx] + b[idx];
}
```

The arrays `a`, `b`, and `c` need to be accessible to the GPU kernel, which means they must reside in device memory. Since regular host memory (allocated with `malloc` or `new`) is not directly visible to the GPU, we must:

- Allocate memory on the GPU
- Copy the input data (`a` and `b`) from host memory to device memory
- Launch the kernel, passing the device pointers as arguments
- Copy the result (`c`) back from device to host memory after the kernel finishes.
- Free the device memory.

Allocating memory on the GPU

The cudaMalloc() allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The memory is not cleared. Returns cudaErrorMemoryAllocation in case of failure. The cudaFree() function frees memory on the device

```
__host__ __device__ cudaError_t cudaMalloc(void** devPtr, size_t size)
```

devPtr	Pointer to allocated device memory
size	Requested allocation size in bytes

```
__host__ __device__ cudaError_t cudaFree(void* devPtr)
```

devPtr	Device pointer to memory to free

Copy data to and from the GPU

The `cudaMemcpy()` function copies count bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy. `cudaMemcpy()` is **blocking (synchronous)** with respect to the host.

```
__host__ cudaError_t cudaMemcpy(void* dst, const void* src,
                                size_t count, cudaMemcpyKind kind)
```

`dst`	Destination memory address
`src`	Source memory address
`count`	Size in bytes to copy
`kind`	Type of transfer. must be one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, or `cudaMemcpyDefault`

GPU vector addition (1/3)

The first step is to allocate arrays that live in CPU memory and fill these with test data

```
float* host_a = (float*)malloc(VECTOR_SIZE * sizeof(float));
float* host_b = (float*)malloc(VECTOR_SIZE * sizeof(float));

for (size_t i = 0; i < VECTOR_SIZE; i++) {
    host_a[i] = (float)i;
    host_b[i] = (float)i;
}
```

Then, allocate the corresponding arrays in GPU memory and transfer the data from CPU to GPU memory using `cudaMemcpy()`

```
float *device_a, *device_b, *device_c;
cudaMalloc(&device_a, VECTOR_SIZE * sizeof(float));
cudaMalloc(&device_b, VECTOR_SIZE * sizeof(float));
cudaMalloc(&device_c, VECTOR_SIZE * sizeof(float));

cudaMemcpy(device_a, host_a, VECTOR_SIZE * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(device_b, host_b, VECTOR_SIZE * sizeof(float), cudaMemcpyHostToDevice);
```

GPU vector addition (2/3)

The kernel calculates each thread's global index, performs a bounds check, and computes the element-wise sum of the input vectors

```
__global__ void vector_add_kernel(float* a, float* b, float* c) {
  const size_t idx = blockIdx.x * blockDim.x + threadIdx.x;

  if (idx ≥ VECTOR_SIZE)
    return;

  c[idx] = a[idx] + b[idx];
}
```

This kernel can be launched as a grid of blocks with 256 threads per block, where the grid size is computed based on the vector length

```
const int num_threads = 256;
const int num_blocks = (VECTOR_SIZE + (num_threads - 1))  / num_threads;

vector_add_kernel<<<num_blocks, num_threads>>>(device_a, device_b, device_c);
```

GPU vector addition (3/3)

The kernel calculates each thread's global index, performs a bounds check, and computes the element-wise sum of the input vectors

```
float* host_c = (float*)malloc(VECTOR_SIZE * sizeof(float));
cudaMemcpy(host_c, device_c, VECTOR_SIZE * sizeof(float), cudaMemcpyDeviceToHost);
```

This kernel can be launched as a grid of blocks with 256 threads per block, where the grid size is computed based on the vector length

```
cudaFree(device_a);
cudaFree(device_b);
cudaFree(device_c);

free(host_a);
free(host_b);
free(host_c);
```

Dealing with structures

When passing structures as kernel arguments in CUDA, special care must be taken to ensure that all members of the structure are valid in device memory

- This is particularly important for structures that contain pointers or references to dynamically allocated data. When such a structure is passed by value to a kernel, CUDA copies it from host to device memory as a raw byte sequence. Any pointer members will still reference host memory addresses, which are invalid on the GPU
- To avoid illegal memory accesses, all pointer members must be allocated in device memory, and the structure itself should either reside in device memory or be updated so that its internal pointers refer to device addresses
- In practice, this often means performing a deep copy of the structure before launching the kernel and passing a pointer to the device-side copy rather than passing the structure by value

Deep copy of a structure

The deep copy of a structure from the host to the device involve:

- Allocating the structure on the device
- Allocating memory for all internal pointers on the device
- Copying the data referenced by these pointers to device memory
- Updating the host structure to use the device pointers
- Copying the updated structure to the device

```
typedef struct matrix_ {
  size_t num_rows, num_cols;
  float* data;
} matrix_t;

void matrix_copytodevice(matrix_t** devptr,
                         const matrix_t* hostptr) {
  matrix_t* devmat;
  cudaMalloc(&devmat, sizeof(matrix_t));

  float* devdata;
  const size_t num_elems =
    hostptr->num_rows * hostptr->num_cols;

  cudaMalloc((void**)&devdata, num_elems * sizeof(float));
  cudaMemcpy(devdata, hostptr->data, num_elems * sizeof(float),
    cudaMemcpyHostToDevice);

  matrix_t temp = *hostptr;
  temp.data = devdata;

  cudaMemcpy(devmat, &temp, sizeof(matrix_t),
    cudaMemcpyHostToDevice);

  *devptr = devmat;
}
```

Shared Memory

The 1D diffusion equation

The diffusion equation, also known as the heat equation, reads

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, L), t \in (0, T]$$

where $u(x, t)$ is the unknown function to be solved for, x is a coordinate in space, t is time and α is the diffusion coefficient

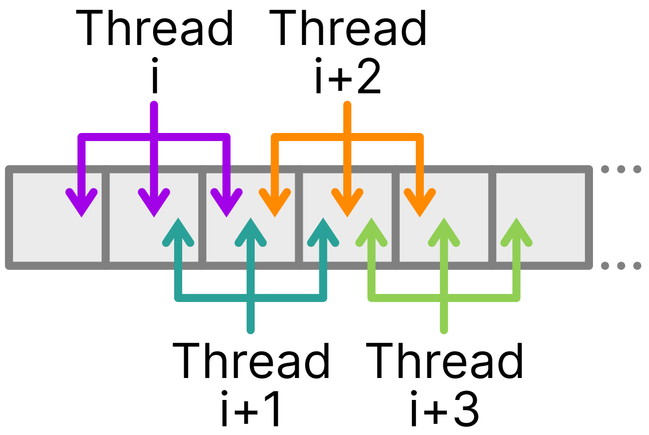
After discretization and using a forward difference in time and a central difference in space, we get

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}$$

so that, at time step $n + 1$, we can update the value of u using

$$u_i^{n+1} = u_i^n + \frac{\alpha \Delta t}{\Delta x^2} (u_{i+1}^n - 2u_i^n + u_{i-1}^n)$$

The 1D diffusion equation: first kernel



```
__global__ void update_kernel(const float* uold, float* unew,
                              const float alpha_dt_dx2, const int ncells)
{
  const int idx = blockIdx.x * blockDim.x + threadIdx.x + 1;

  if (idx > ncells)
    return;

  unew[idx] = uold[idx] + alpha_dt_dx2
    * (uold[idx + 1] - 2.0f * uold[idx] + uold[idx - 1]);
}
```

For each thread, we access 3 locations in the `uold` array. Most of these accesses are shared with neighboring threads

Shared memory overview

Shared Memory is a small, fast on-chip memory space that is shared among threads within the same thread block on a GPU. It provides much lower latency than global memory (hundreds of times faster if used efficiently)

Starting with the Volta architecture, NVIDIA unified the physical hardware for L1 cache and shared memory into a single pool of on-chip memory:

- **L1 cache** managed by the hardware and the access pattern can be non-deterministic as it depends on cache replacement policy. The data exists as long as cache lines remain valid
- **Shared memory:** managed by the programmer who explicitly allocates and accesses it. Data remains valid within a kernel execution and specific thread block

Allocating shared memory

Shared memory variables are declared in the kernel function using the `__shared__` qualifier

```
__shared__ type shmem_array[SIZE]
```

where `SIZE` is a compile time constant which means the size of fixed at compile time. The value of `SIZE` specifies the amount of shared memory allocated **per thread block.**

To allocate shared memory dynamically at runtime, use the `extern` keyword:

```
extern __shared__ type shmem_array[]
```

Then specify the total shared memory size (in bytes) when launching the kernel:

```
kernel<<<grid, block, shmem_in_bytes>>>(args)
```

Shared memory and synchronization

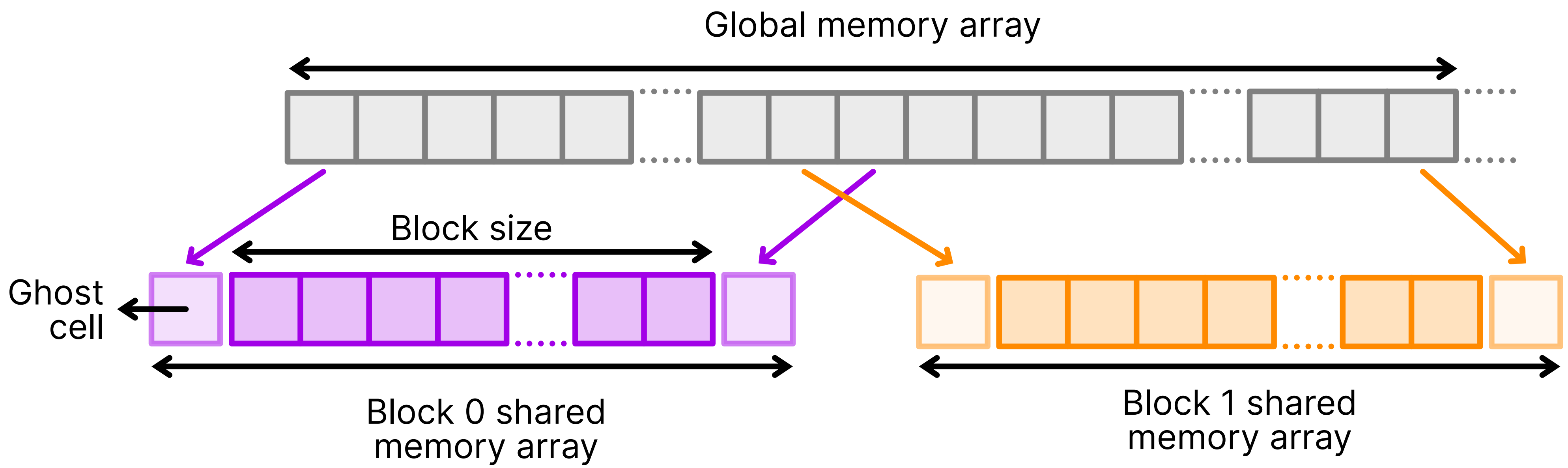
After filling a shared memory array, it's essential to ensure that all threads in the block have finished writing their data before any thread starts reading or modifying it.

This synchronization is achieved by calling the CUDA built-in barrier function `__syncthreads()` which acts as a barrier at which all threads in the block must wait before any is allowed to proceed

```
__shared__ float shmem_array[SIZE]

// Fill the array from global memory

__syncthreads();

// Read shmem_array
```

The 1D diffusion equation: shared memory

Shared memory can be used to reduce the number of redundant transfers from global memory by storing the data required by a block into shared memory



The 1D diffusion equation: shared memory kernel

For the 1D diffusion equation implementation using shared memory:

- Each thread loads its own cell value into shared memory
- The first thread in each block loads the left ghost cell
- The last thread in each block (or the thread whose global index equals the number of cells) loads the right ghost cell

```
__global__ void update_kernel(const float* uold, float* unew,
    const float alpha_dt_dx2, const int ncells)
{
  __shared__ float u_shared[BLOCK_SIZE + 2];

  const int g_idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
  const int s_idx = threadIdx.x + 1;

  if (g_idx > ncells)
    return;

  u_shared[s_idx] = uold[g_idx];

  if (threadIdx.x == 0)
    u_shared[0] = uold[g_idx - 1];

  if (threadIdx.x == blockDim.x - 1 || g_idx == ncells)
    u_shared[s_idx + 1] = uold[g_idx + 1];

  __syncthreads();

  unew[g_idx] = u_shared[s_idx] + alpha_dt_dx2
    * (u_shared[s_idx + 1] - 2.0f * u_shared[s_idx]
                            + u_shared[s_idx - 1]);
}
```

Memory coalescence and alignment

Memory coalescing

Memory coalescing is a technique to optimize data access by grouping multiple logical memory requests into a single, wider physical one. It improves memory bandwidth by ensuring that threads in a warp or group access consecutive memory locations. On NVIDIA GPUs

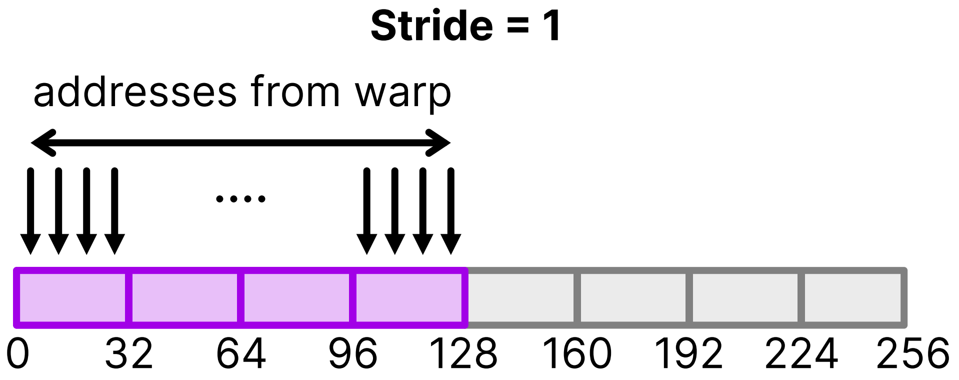
- Global memory is accessed via 32-byte memory transactions
- When a thread requests data from global memory, memory accesses from all threads in that warp are coalesced into a minimum number of memory transactions
- The number of memory transactions required depends on the size of the word accessed by each thread and the distribution of the memory addresses across the threads

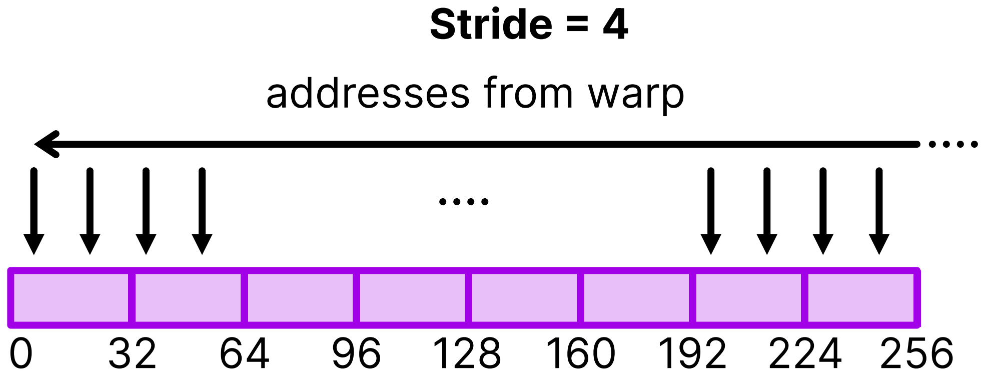
Memory coalescing

If a kernel accesses global memory with a stride, it can lead to additional 32 bytes sectors read/write. For example, if a kernel reads an an array of `float`:

- **stride 1:** 4 sectors
- **stride 4:** 16 sectors

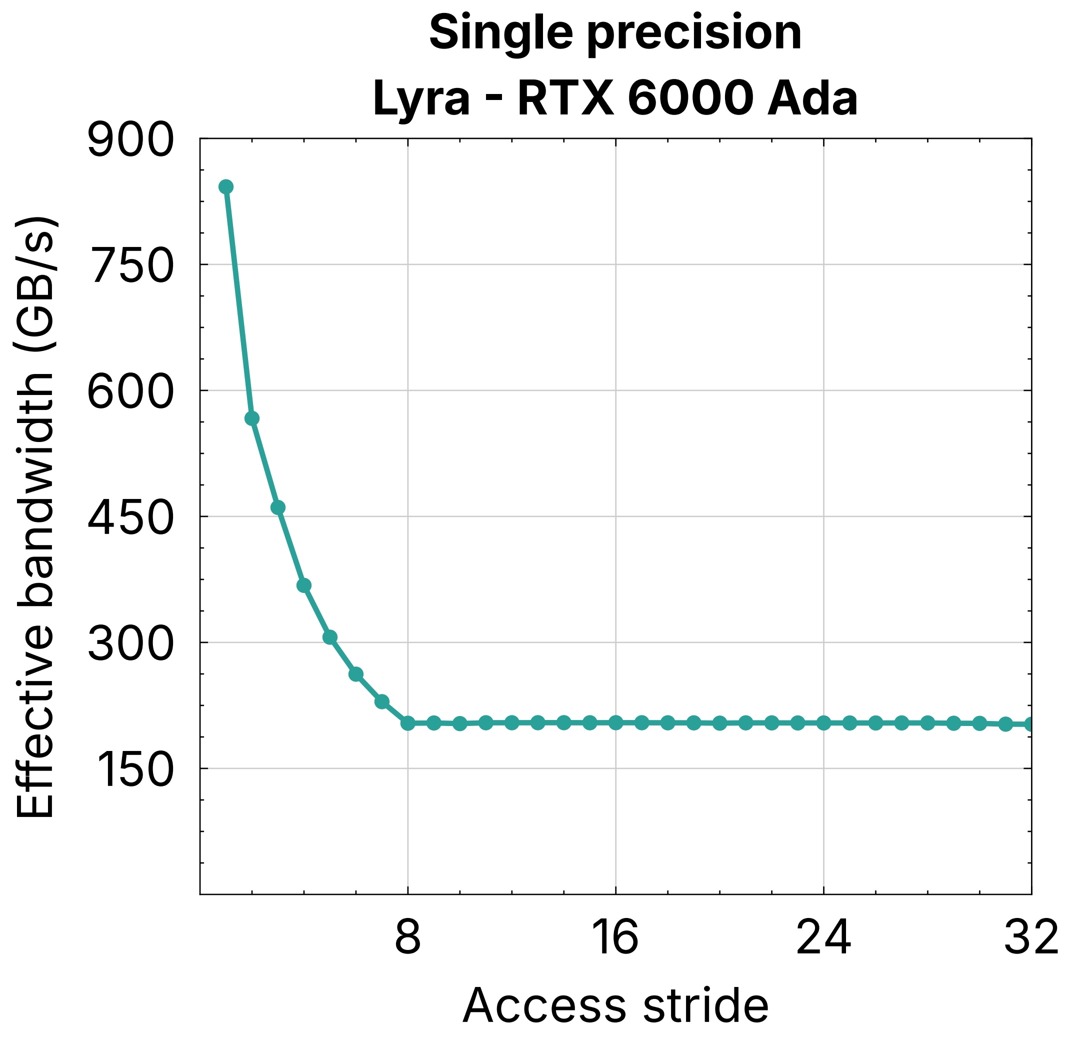
```
int idx = blockIdx.x * blockDim.x + threadIdx.x;

if (idx ≥ SIZE)
  return;

int strided_idx = (idx * stride) % SIZE;
out[idx] = 2.0f * in[strided_idx];
```





Memory coalescing: effect on effective bandwidth

- Non-coalescent memory accesses have a huge impact on effective memory bandwidth
- For large strides, when threads of a warp access memory addresses that are far apart in physical memory, the hardware can't combine these accesses efficiently, the effective bandwidth is poor



Memory coalescing: multi-dimensional arrays

When using 2 or 3-dimensional thread blocks in a CUDA kernel, the threads are laid out linearly with the X index, or `threadIdx.x`, moving the fastest, then Y (`threadIdx.y`) and then Z (`threadIdx.z`)

When using a 2D thread blocks to access 2D data such as a matrix stored as a 1D memory array in row-major storage row accesses are contiguous.

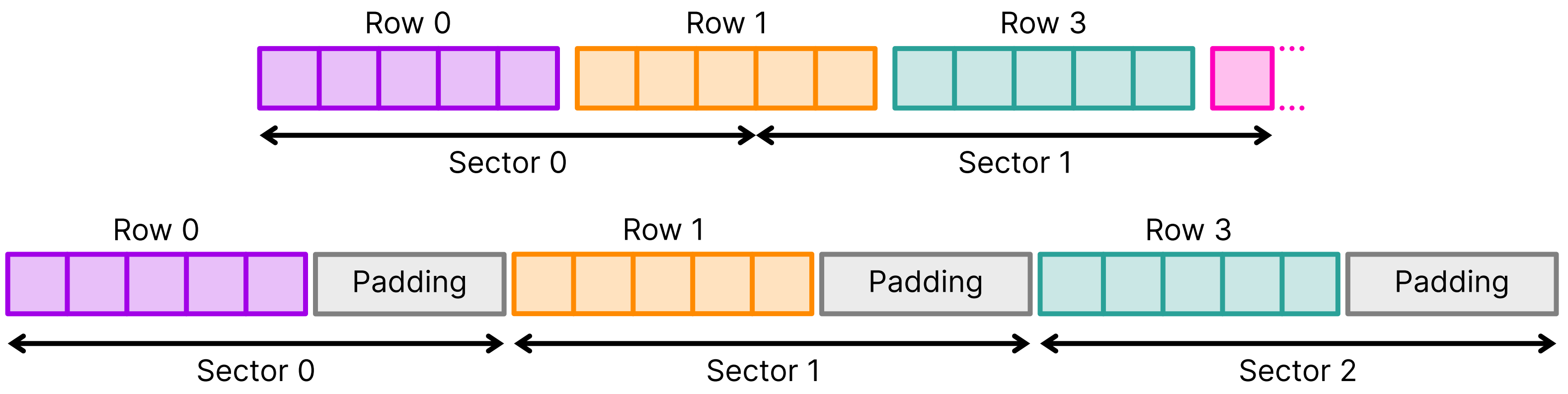
- If consecutive threads access consecutive memory locations across a row, those accesses will be efficient (coalesced)
- Column access is inefficient (strided, non-coalesced).

Memory coalescing: multi-dimensional arrays

Because of how memory requests are handled, performance improves when each row starts at an address aligned to the GPU natural transaction size (typically 128 bytes per warp)

If threads in a warp access misaligned addresses, such as rows not starting on a 128-byte boundary,the hardware must issue multiple overlapping transactions to fetch the same data, wasting bandwidth

To prevents this we can add padding bytes between rows so each begins on a properly aligned boundary



Allocation of a 2D array with memory alignment

The `cudaMallocPitch()` function allocates “pitched” (2D) memory on the device. Pitched memory means that each row of the 2D array is padded to meet specific alignment requirements imposed by the GPU hardware. This ensures that data accesses by consecutive threads are properly aligned and coalesced

```
__host__ cudaError_t cudaMallocPitch(void** devPtr, size_t* pitch,
                                     size_t width, size_t height)
```

`devPtr`	Pointer to allocated pitched device memory
`pitch`	Pitch for allocation
`width`	Requested pitched allocation width (in bytes)
`height`	Requested pitched allocation height

Copy of a 2D array with memory alignment

The `cudaMemcpy2D()` function copies a matrix (`height` rows of `width` bytes each) from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy. `dpitch` and `spitch` are the widths in memory in bytes of the 2D arrays pointed to by `dst` and `src`

```
__host__ cudaError_t cudaMemcpy2D (void* dst, size_t dpitch,
                                   const void* src, size_t spitch, size_t width,
                                   size_t height, cudaMemcpyKind kind)
```

`dst`	Destination memory address	`dpitch`	Pitch of destination memory
`src`	Source memory address	`spitch`	Pitch of source memory
`width`	Width of the matrix (in bytes)	`height`	Height of the matrix
`kind`	Type of transfer		

Pitched array example: matrix addition (1/3)

As an example, consider a matrix addition kernel where each thread computes the sum of a single matrix element. The pitch parameter accounts for the padding bytes added between rows by `cudaMallocPitch()`, ensuring correct indexing across properly aligned rows

```
__global__ void matrix_add_kernel(float* a, float* b, float* c, size_t mat_sz, size_t
pitch) {
  const size_t col_idx = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t row_idx = blockIdx.y * blockDim.y + threadIdx.y;

  if (col_idx >= mat_sz || row_idx >= mat_sz)
    return;

  const size_t idx = row_idx * pitch + col_idx;

  c[idx] = a[idx] + b[idx];
}
```

Pitched array example: matrix addition (2/3)

To launch the kernel presented on the previous slide, we use `cudaMallocPitch()` to allocate the memory and `cudaMemcpy2D()` to copy memory to and from the device

```
cudaMallocPitch(&dev_a, &pitch, row_bytes, mat_sz);
// Same for dev_b and dev_c
cudaMemcpy2D(dev_a, pitch, host_a, row_bytes, row_bytes, mat_sz, cudaMemcpyHostToDevice);
// Same for dev_b

const size_t pitched_size = pitch / sizeof(float);

dim3 block(32, 8);
dim3 grid((mat_sz + block.x - 1)  / block.x, (mat_sz + block.y - 1)  / block.y);

matrix_add_kernel<<<grid, block>>>(dev_a, dev_b, dev_c, mat_sz, pitched_size);

cudaMemcpy2D(host_c, row_bytes, dev_c, pitch, row_bytes, mat_sz, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
```

Pitched array example: matrix addition (3/3)

For a matrix of size 1024, the kernel should execute with $(1024 \cdot 1024)/32 = 32768$ warps. The number of sectors transferred for each matrices should be

$$1024 \cdot 1024 \cdot \frac{4 \text{ bytes}}{32 \text{ bytes per sector}} = 131072 \text{ sectors} \cdot 2 \text{ matrices} = 262144 \text{ sectors}$$

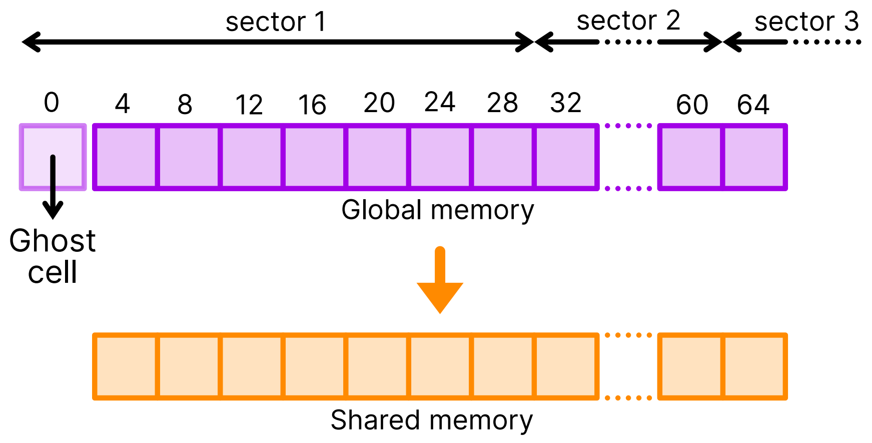
In the non-pitched case, a matrix width of 1023 causes additional memory sectors to be fetched because the rows are not properly aligned, resulting in misaligned and less efficient memory accesses.

	Non-pitched (size = 1023)	**Non-pitched (size = 1024)**	**Pitched (size = 1023)**	**Pitched (size = 1024)**
Number of read requests	65 472	65 536	65 472	65 536
Number of read sectors	313 839	262 144	261 888	262 144

The 1D diffusion equation: memory alignment

A one-dimensional array can also suffer from unaligned memory accesses. In the 1D diffusion example, the presence of the left ghost cell causes the global-to-shared memory loads to become misaligned with the sector boundaries.

The stores of the final results are affected by the same issue

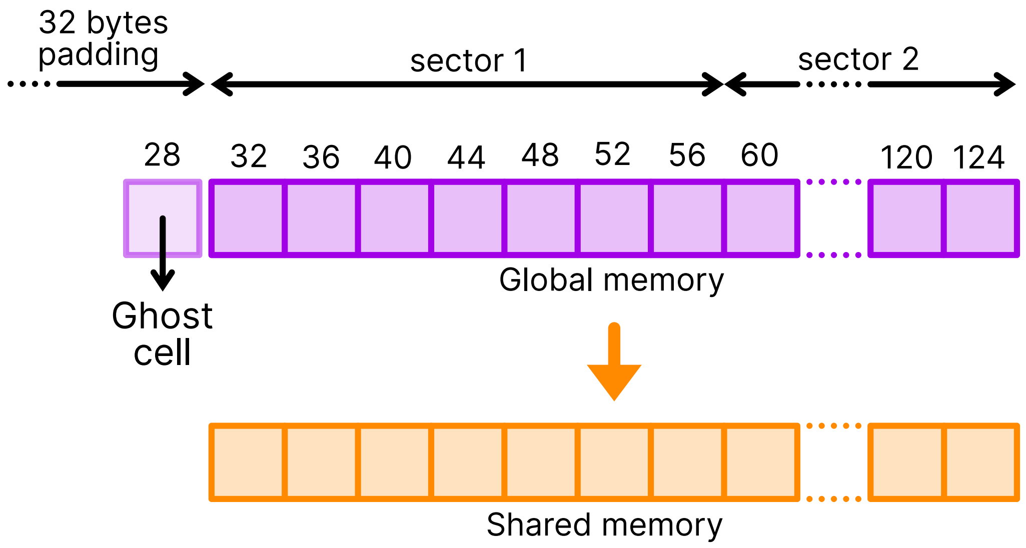


```
__shared__ float u_shared[BLOCK_SIZE + 2];

const int g_idx = blockIdx.x * blockDim.x + threadIdx.x + 1;
const int s_idx = threadIdx.x + 1;

u_shared[s_idx] = uold[g_idx];

// ...

unew[g_idx] = u_shared[s_idx] + ...
```

The 1D diffusion: with left padding

The solution is to add padding to align global-to-shared memory loads with sector boundaries, including the left ghost cell in the padding.

This approach also reduces the number of sectors accessed when writing the results back to global memory



	Load without padding	Load with padding	Store without padding	Store with padding
Number of requests	80 000	80 000	64 000	64 000
Number of sectors	318 913	272 001	320 000	256 000

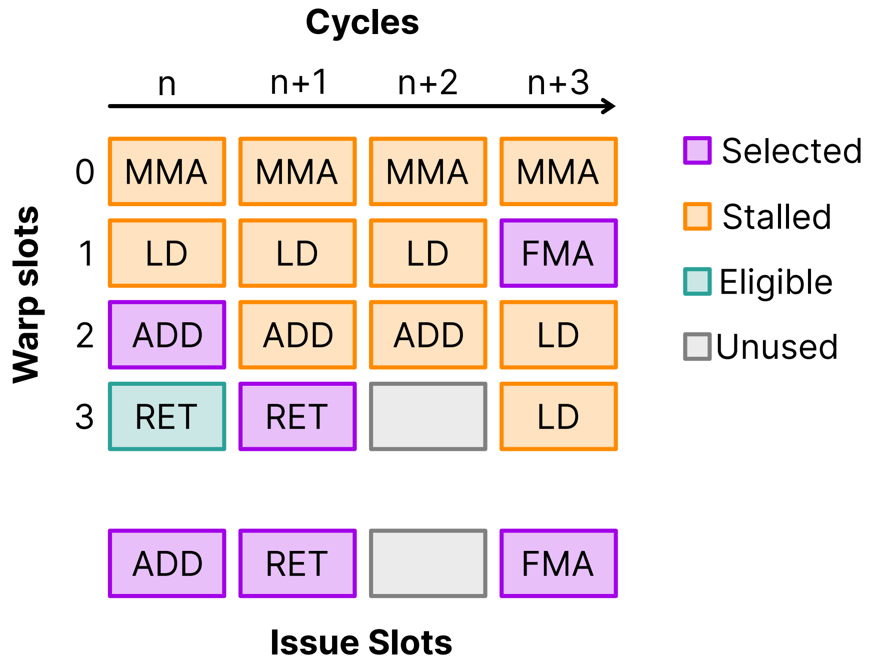
Occupancy and latency hiding

Occupancy: definition

Occupancy is a performance metric that measures how effectively the GPU computational resources are being utilized. It refers to the ratio of active warps to the maximum number of warps that can theoretically reside on a streaming multiprocessor at any given time

For example, if a SM has four warp slots and we consider an execution over 4 clock cycles

- the total number of available slots is $4 \cdot 4 =$ 16 slots.
- there are active warps in 15 of them, therefore the occupancy is $14/15 \approx 94\%$.



Occupancy: latency hiding

Having a lot of wraps executing concurrently is important for latency hiding. Latency hiding is a strategy to mask long-latency operations by running many of them concurrently.

For example, consider the vector addition kernel:

```
# c[idx] = a[idx] + b[idx];

LDG.E R4, [R4.64] # Load from a (400 cycles)
LDG.E R3, [R2.64] # Load from b (400 cycles)
FADD  R9, R4, R3  # Add a and b (4 cycles)
STG.E [R6.64], R9 # Store to c  (100 cycles)
```

Executed sequentially, this would take 904 cycles to complete but by operating concurrently, latency can be hidden by other operations

Occupancy: limiting factor

The maximum number of warps active at one time on a Streaming Multiprocessor (SM) is limited by several factors:

- **Hardware limits:** each GPU architecture has a fixed maximum number of active warps per SM. For example, Lyra GPUs support up to 1 536 threads per SM (48 warps)
- **Register Usage:** every thread consumes a certain number of registers. Higher register usage per thread reduces the total number of threads (and thus warps) that can be active simultaneously, since registers are a shared resource
- **Shared Memory:** shared memory is also limited per SM. If a kernel uses a large amount of shared memory, fewer thread blocks can reside on the SM at once, lowering overall occupancy

Occupancy: limiting factor examples

Register usage

- **Hardware limit:** 64K (65536) registers per SM
- **Kernel configuration:** Each thread uses 64 registers and a block size of 256 threads ($256 \cdot 64 = 16384$ registers)
- **Occupancy:** $65536/16384 = 4$ blocks can fit on an SM (32 warps)

Shared memory usage

- **Hardware limit:** 100 KB shared memory per SM
- **Kernel configuration:** Each block of 256 threads uses 40 KB of shared memory
- **Occupancy:** Only $\lfloor 100/40 \rfloor = 2$ blocks can fit on an SM (16 warps)

Getting registers usage

You can obtain register usage at compile time by enabling verbose output from the PTX optimizing assembler (`--ptxas-options=-v`)

```
 $ nvcc --ptxas-options=-v -O3 -o vector_add vector_add.cu 2>&1 | c++filt
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function 'vector_add_kernel(float*, float*, float*)' for 'sm_89'
ptxas info    : Function properties for vector_add_kernel(float*, float*, float*)
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, used 0 barriers, 376 bytes cmem[0]
```

Register allocations are rounded up to the nearest 256 registers per warp which means that the real number of registers used will be

$$\left\lceil \frac{32 \cdot 12}{256} \right\rceil \cdot \frac{256}{32} = 16 \text{ registers/threads}$$

High occupancy ≠ performance: instruction level parallelism

A common guideline for achieving good performance is to increase the number of threads per SM and the size of thread blocks. Although high occupancy can help hide latencies and improve efficiency, it represents only one of the many factors influencing performance

For instance, a high degree of instruction-level parallelism can greatly improve performance. By providing sufficient independent instructions, it is possible to minimize warp stalls and mask execution latencies.

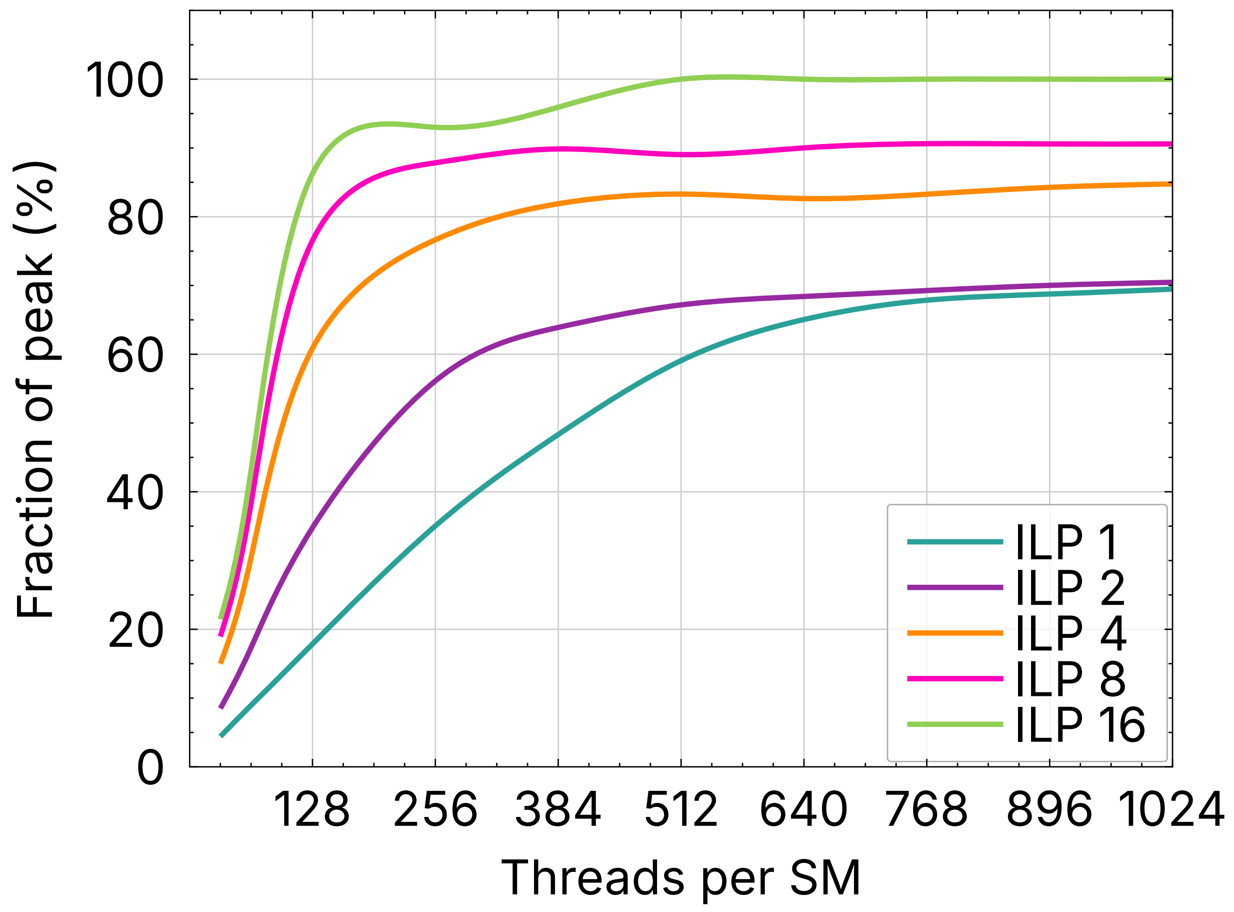
```
x = a + b; // takes ~4 cycles to execute
y = a + c; // independent, can start anytime
// stall
z = x + d; // dependent, must wait for completion
```

Understanding Latency Hiding on GPUs, PhD thesis of V. Volkov

Instruction level parallelism in action

For kernels with sufficient instruction-level parallelism (ILP), fewer threads are needed to achieve a significant fraction of the GPU peak performance

- **ILP 4:** more than 80% of peak can can be achieved at 25% occupancy
- **ILP 8:** more than 80% of peak can can be achieved at 16% occupancy



High occupancy ≠ performance: data parallelism

A common guideline for achieving good performance is to increase the number of threads per SM and the size of thread blocks. Although high occupancy can help hide latencies and improve efficiency, it represents only one of the many factors influencing performance

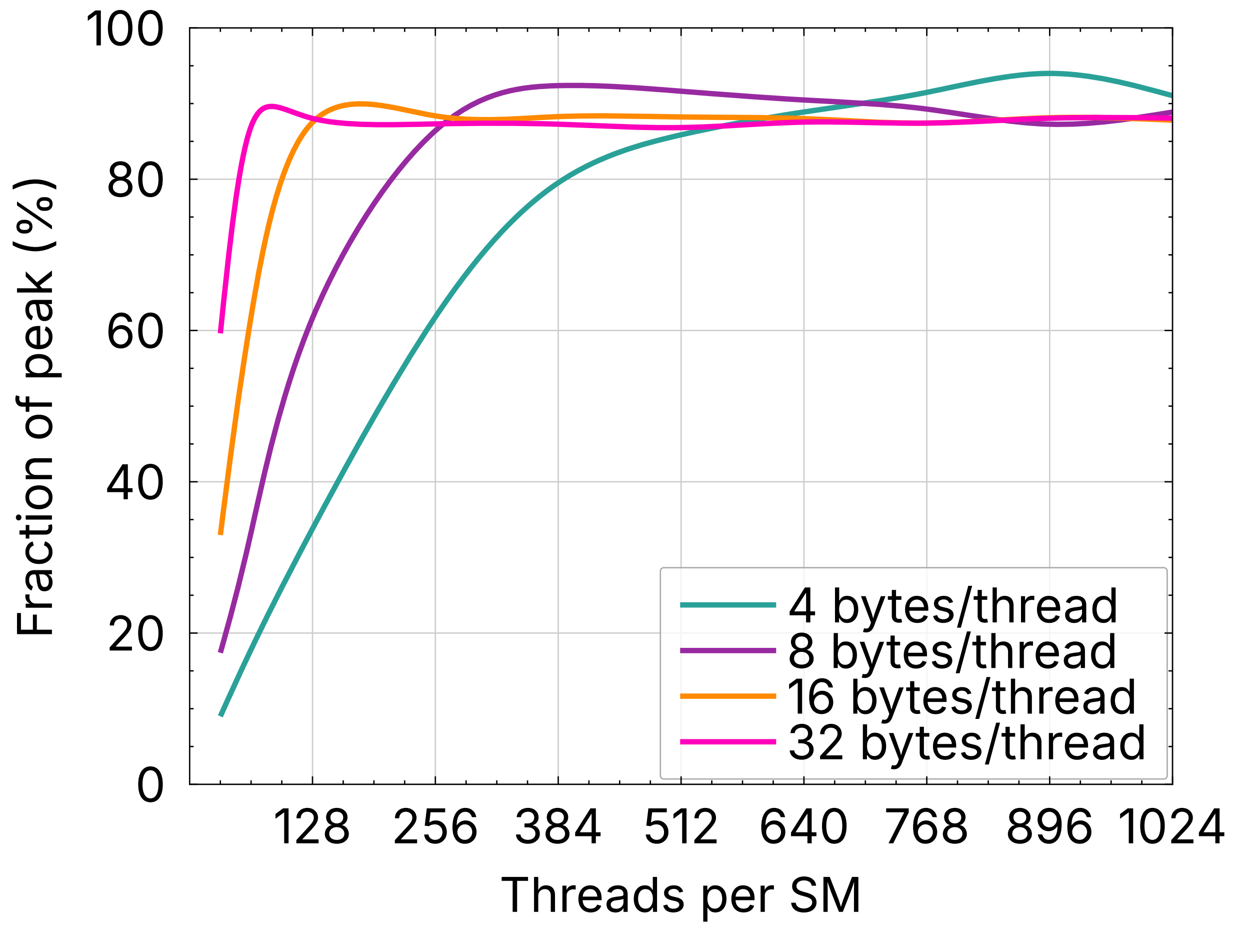
Threads stall when encountering data dependencies rather than during memory accesses per se. By issuing several independent memory operations the hardware can overlap these accesses and hide data access latency, improving overall throughput

```
float a0 = src[index];
// no stall
float a1 = src[index + blockDim.x];
// stall
dst[index] = a0;
dst[index + blockDim.x] = a1;
```

Data parallelism in action

For kernels with sufficient independent memory accesses, fewer threads are needed to achieve a significant fraction of the GPU peak memory bandwidth

- **16 bytes/thread:** more than 80% of peak can can be achieved at 8% occupancy
- **32 bytes/thread:** more than 80% of peak can can be achieved at 4% occupancy



Latency hiding: take home message

Hiding latency is critical to achieving good performance, which requires maintaining enough active warps. While having a high number of active warps per SM (high occupancy) is important, it is not the only factor influencing GPU kernel performance:

- For low arithmetic intensity, it is possible to achieve a large fraction of the available memory bandwidth even at low occupancy, as long as enough independent memory requests are in flight
- For high arithmetic intensity, latency can be effectively hidden even with low occupancy by exploiting instruction-level parallelism

Using more registers or shared memory per thread or block can reduce occupancy but may still improve overall performance

Some extras

CUDA runtime call error checking

Manually verifying the return code for each CUDA runtime API call can be cumbersome. Therefore, developers often encapsulate these checks within a macro to simplify error handling and improve code readability

The macro serves to encapsulate CUDA runtime calls, providing optional error-checking functionality that can be enabled at compile time by defining the `CUDA_DEBUG` flag (using the `-DCUDA_DEBUG` compiler option)

```c
// Compile with -DCUDA_DEBUG to enable checks
#ifndef CUDA_DEBUG
  #define CUDART_CHECK(call) call
#else
  #define CUDART_CHECK(call) cuda_error_check( \
    (call), __FILE__, __LINE__)

  static inline void cuda_error_check(cudaError_t err,
                                      const char *file, int line) {
    if (err != cudaSuccess) {
      fprintf(stderr, "CUDA check failed at %s:%d: %s\n",
              file, line, cudaGetErrorString(err));
      exit(1);
    }
  }
#endif

// Runtime call
CUDART_CHECK( cudaMalloc (&devptr, N * sizeof (float)) );

// Kernel launches are asynchronous, check cudaGetLastError()
kernel<<<grid, block>>>(args);
CUDART_CHECK( cudaGetLastError() );
```

Measuring kernel execution time

To measure how long a kernel takes to execute on the GPU, not on the CPU, i.e., measuring the elapsed time between two points in the GPU command stream, we can use CUDA events

CUDA events are lightweight synchronization and timing primitives provided by the CUDA runtime:

- `cudaEventRecord()` inserts an event and when the GPU reaches this point, it timestamps the event
- `cudaEventSynchronize()` makes the CPU wait until the GPU reaches a given event
- `cudaEventElapsedTime()` computes the time difference between two recorded events, in milliseconds

```
float elapsed;

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);

kernel<<<grid, block>>>(args);

cudaEventRecord(stop);
cudaEventSynchronize(stop);
// Elapsed time in milliseconds
cudaEventElapsedTime(
  &elapsed, start, stop);
```