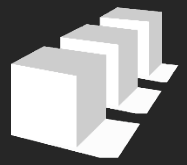
Universidad Zaragoza

1542



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

GPU programming foundations and performance optimization

Mario Morales Hernández (mmorales@unizar.es)

March18 2025

Outline

- Introduction
- Basic concepts in GPU computing
  - Understanding a loop
  - The GPU memory
  - CUDA example
- Performance optimization strategies
- Vectorization
- Memory access optimization
- Thread management and load balancing
- Computation optimization and best practices

What is GPU computing?

1 Definition

GPU computing refers to the use of a **Graphics Processing Unit (GPU)** to perform computation in applications traditionally handled by the Central Processing Unit (CPU). GPUs excel at processing **many tasks simultaneously**, making them ideal for complex computations and data-intensive tasks.

2 Evolution

Originally designed for rendering graphics in video games, GPUs have evolved to become highly efficient parallel processors, capable of a wide range of computational tasks beyond graphics rendering

3 Key Concept

Parallelism – **GPUs contain thousands of cores** designed for performing multiple calculations simultaneously, providing a significant speedup for compatible tasks.

CPU vs GPU: key differences

1 **Architecture**

CPUs consist of a few cores optimized for sequential serial processing, while GPUs have thousands of smaller, more efficient cores designed for parallel processing.

2 **Use Cases**

CPUs are versatile and excel at general-purpose tasks and single-threaded operations. GPUs are specialized for tasks that can be parallelized, such as simulations, data analysis, and machine learning.

3 **Performance**

For tasks suited to their architecture, GPUs can significantly outperform CPUs in terms of computation speed and efficiency.

CPU vs GPU: key differences

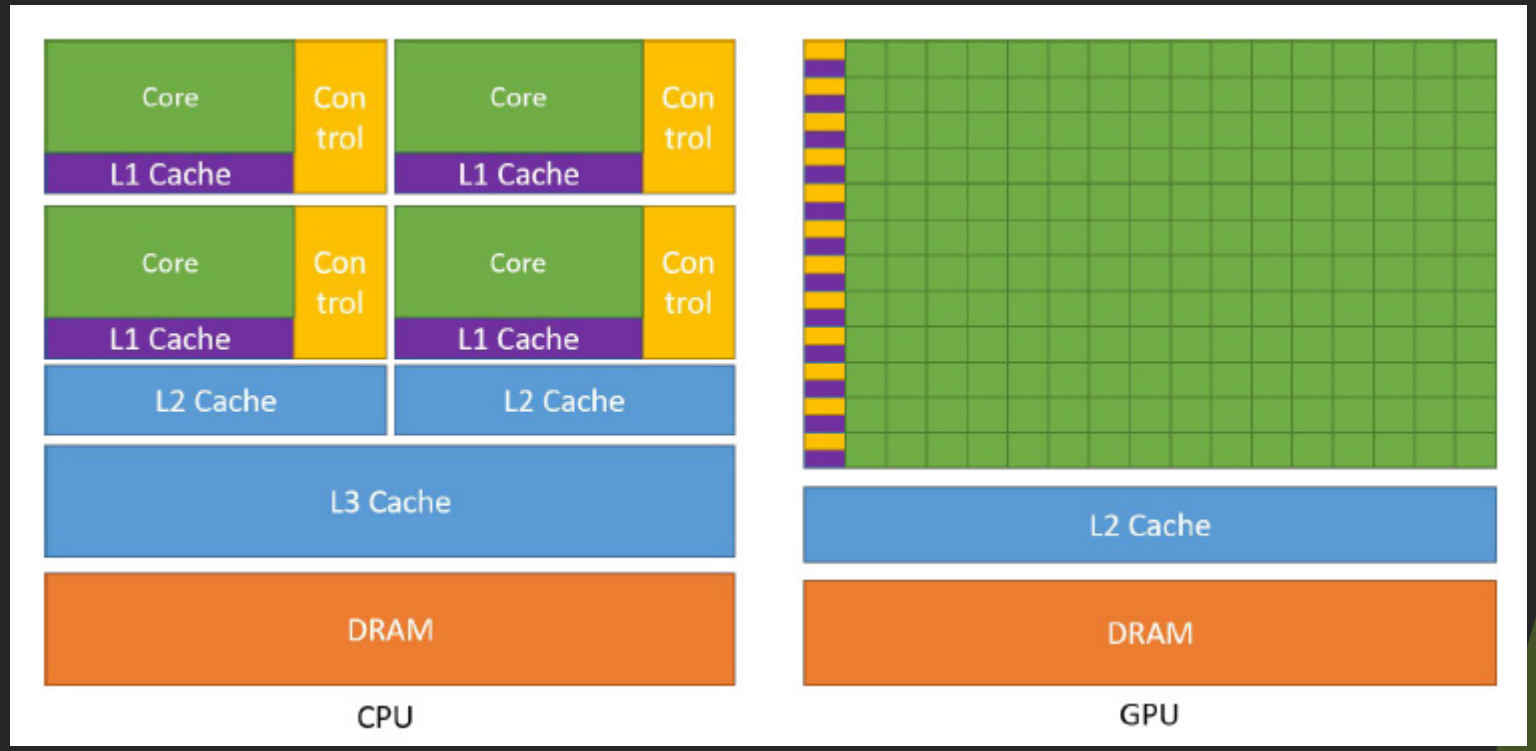
CPU

Optimized to reduce latency

Serial work

Few threads with high frequency

Large amount of memory but slow



GPU

Optimized for throughput

Paralell work

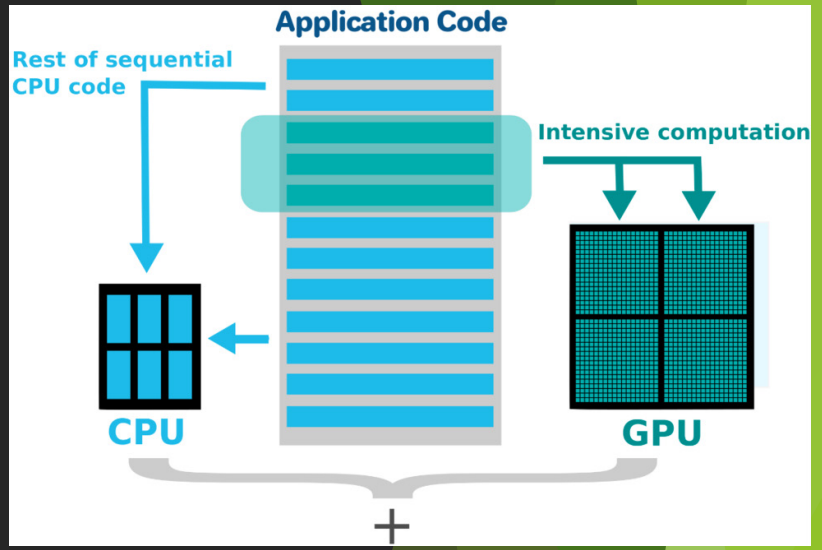
Many threads

“Smaller” memory capacity but faster

CPU + GPU applications: general principles

1 Offload parallel work to GPU

Parallel tasks (e.g., matrix multiplications, image processing) should be offloaded to the GPU for efficient execution.



2 Keep sensitive serial work on CPU

Serial tasks or latency-sensitive operations (e.g., decision-making, control logic) should remain on the CPU

3 Keep data where it used

Minimize memory transfer overhead. Use shared memory on the GPU for frequently accessed data to reduce latency. Use host memory for CPU tasks.

4 Best practices

Identify tasks that are inherently parallel and offload them to the GPU. Use profiling tools to determine bottlenecks and optimize data movement between CPU and GPU.

Basic concepts in GPU

- Massive parallelism through thousands of concurrent threads
- Hierarchical organization for efficient execution
- SIMD architecture for parallel data processing

Thread

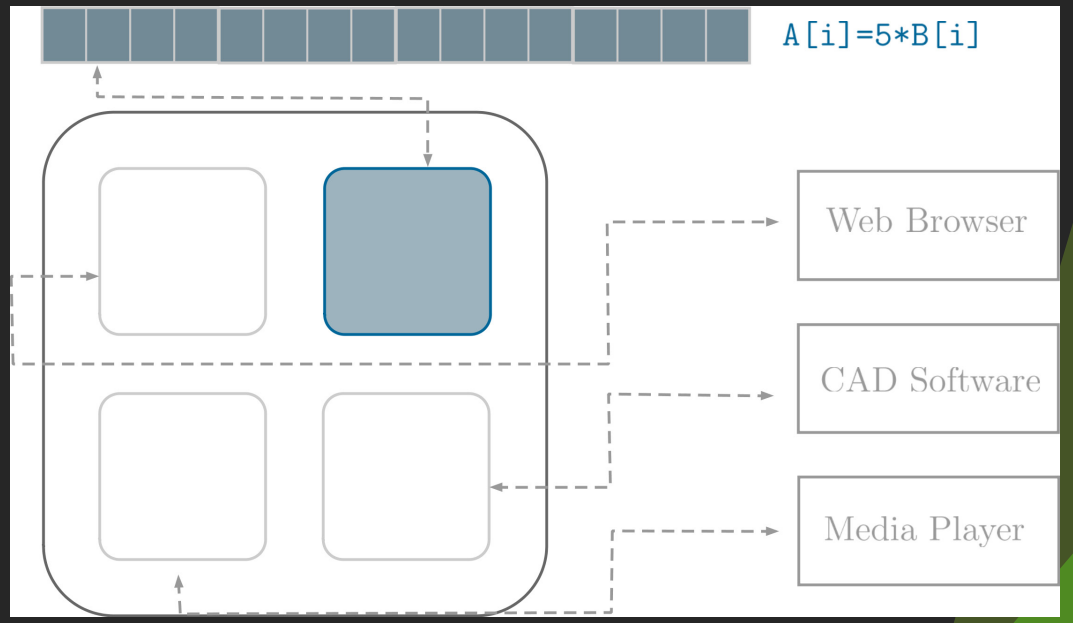
Warp/wavefront

Block

Grid

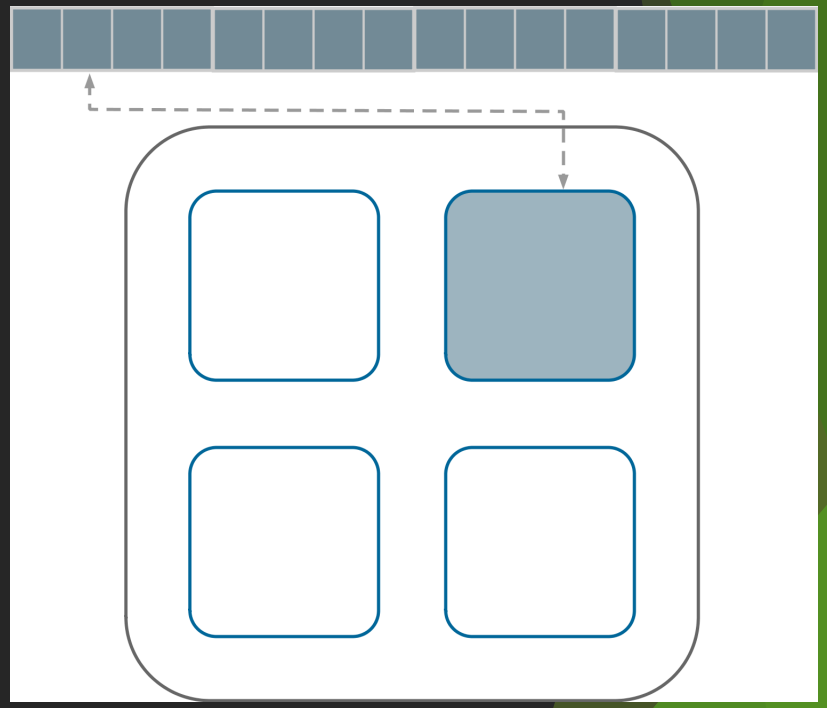
Basic concepts in GPU

- Traditionally, processors are composed by more than one core (multi-core)



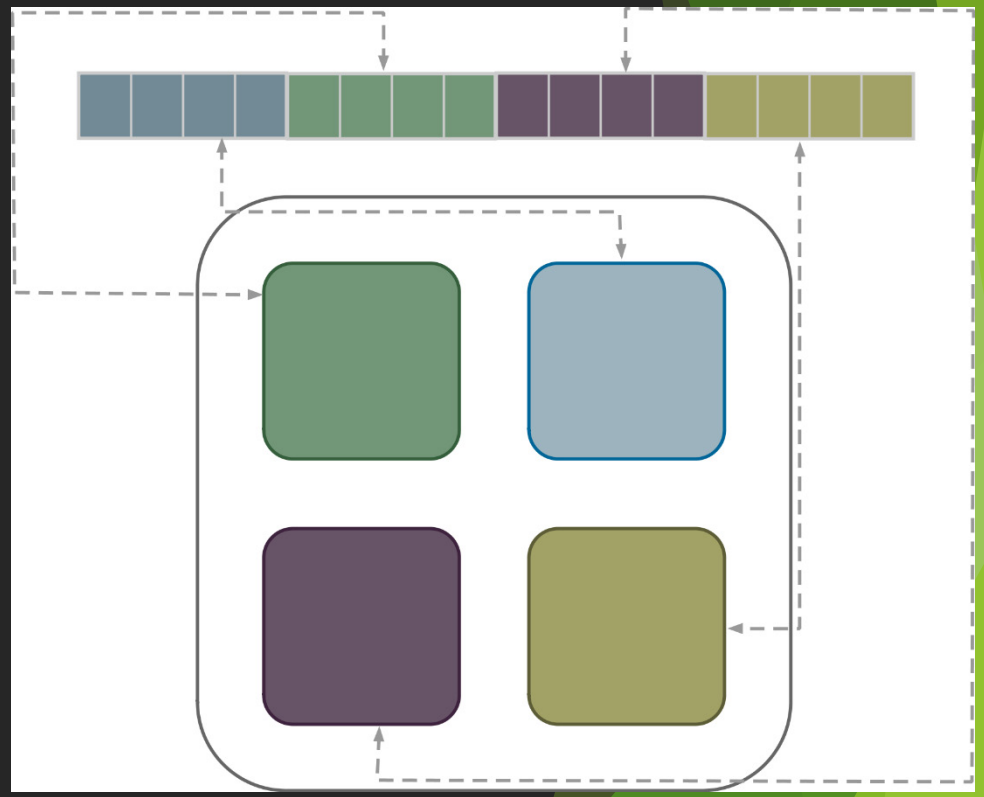
Basic concepts in GPU

- **SISD**: Single Instruction Single Data ("classic" iterative and sequential handling)
- Supported by almost every conventional processors



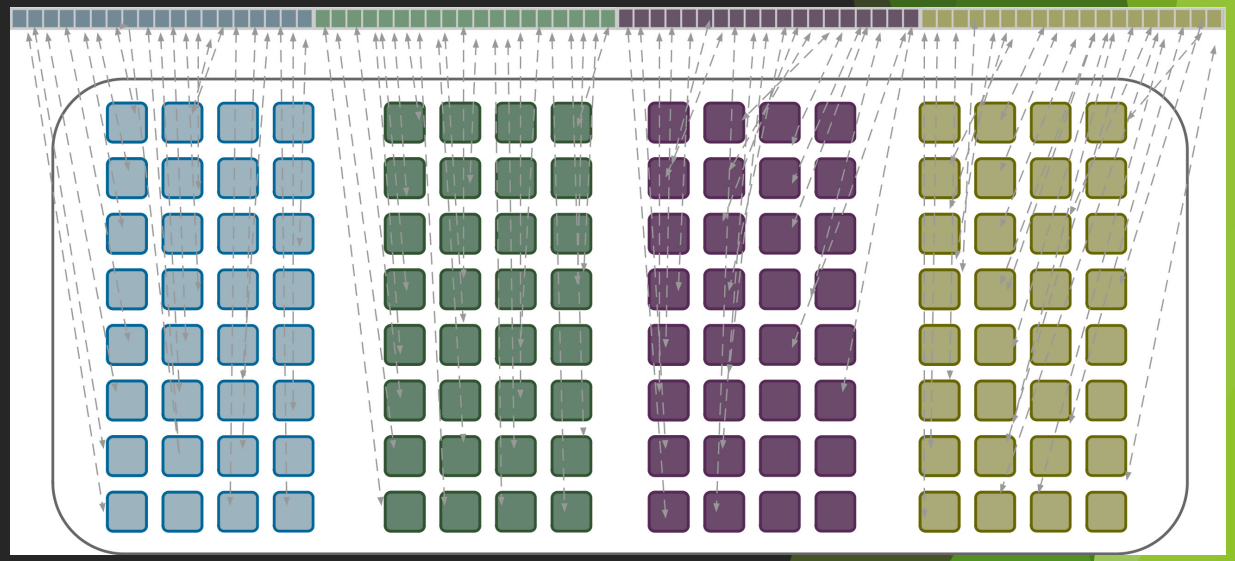
Basic concepts in GPU

- **SIMD**: Single Instruction Multiple Data (shared memory parallel model).
- Supported by almost every modern CPU too, harder to write code.
- Algorithms usually require to be redesigned.



Basic concepts in GPU

- **SIMT**: Single Instruction Multiple Threads
- Large amount of quite simple arithmetical processing units
- Specific processors that require specific programming languages
- Algorithms need to be rethought

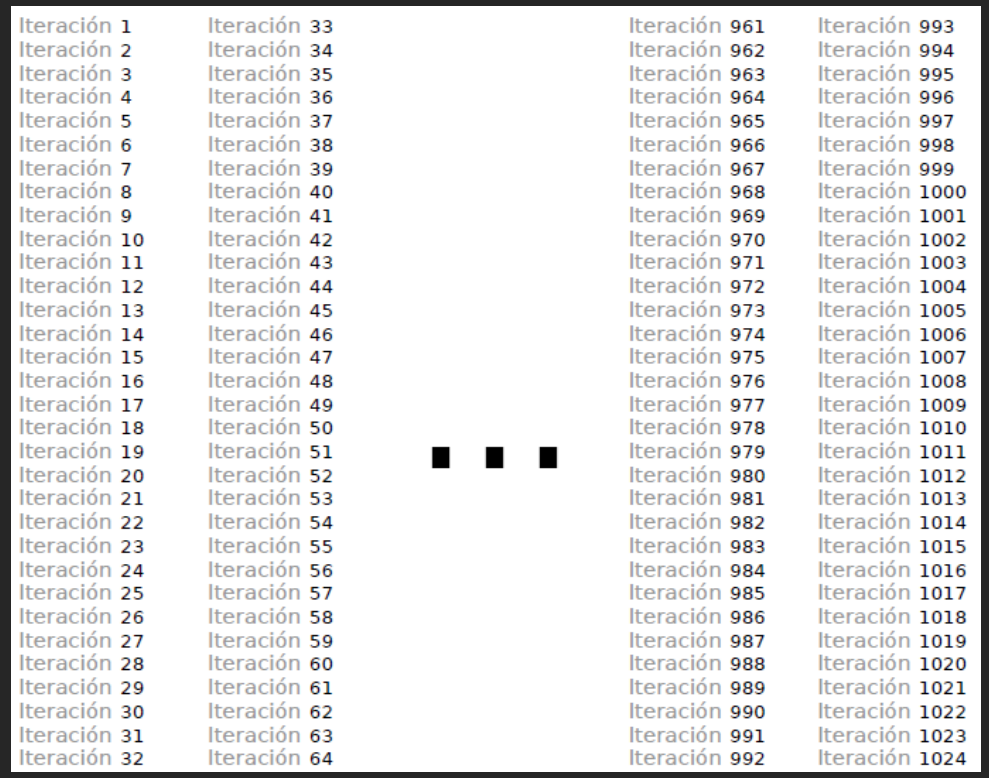


Understanding a loop

- `for` **`i=1..1024`**
- `A[i]=...`
- `end for`

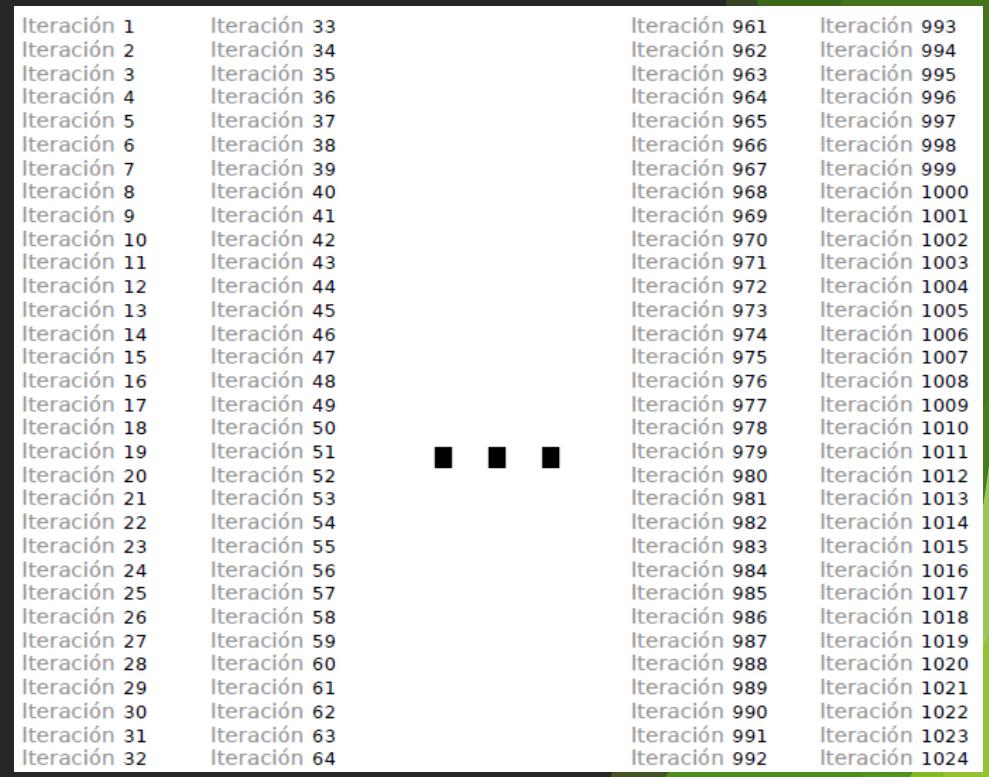
Understanding a loop

- The whole 1024 iterations must be executed...



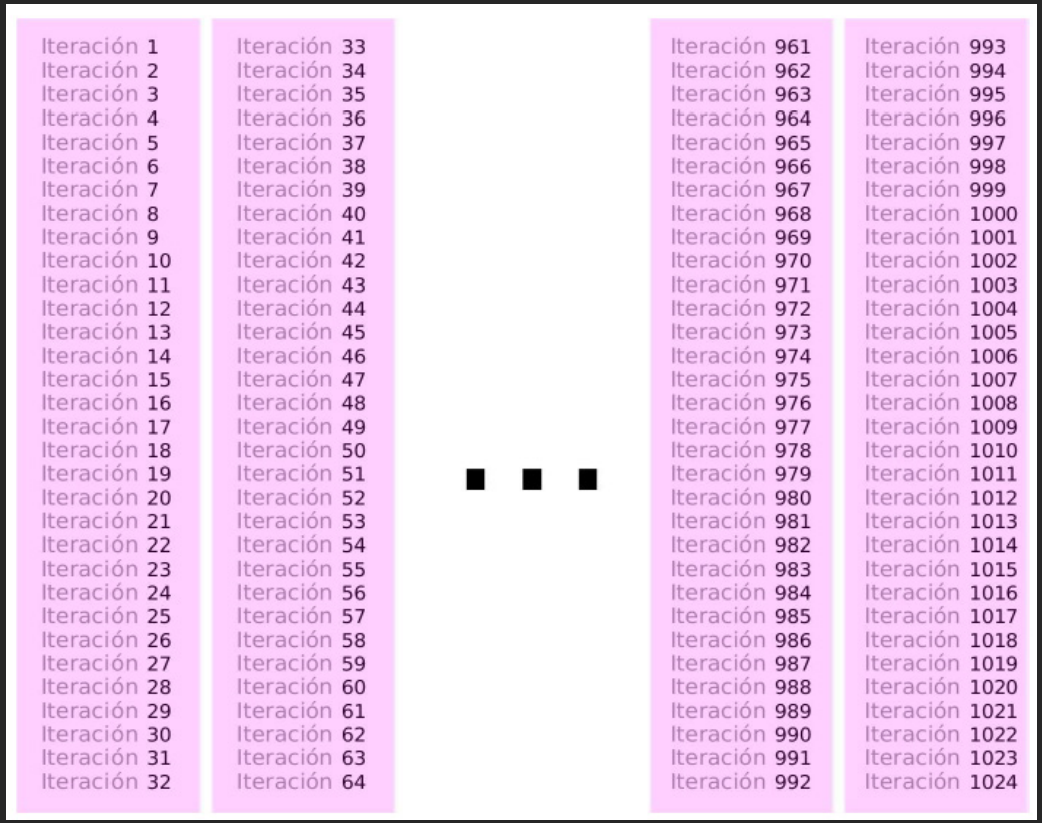
Understanding a loop

- The whole 1024 ~~iterations~~ must be executed...
- ...but they are required to be executed at the same time



Understanding a loop

- Conceptually sets of 32 elements are established → warp

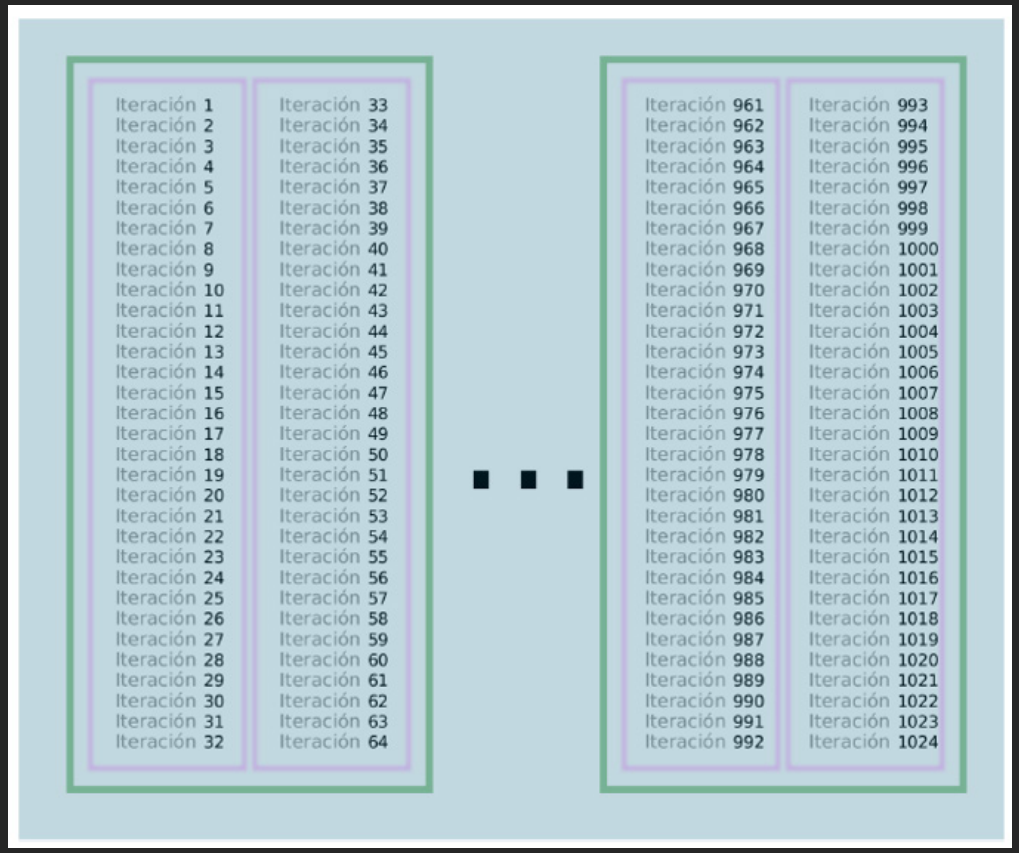


Understanding a loop

- Warps are grouped in sets of (let's say) 2 → blocks

Understanding a loop

- All the warps are regrouped → grid

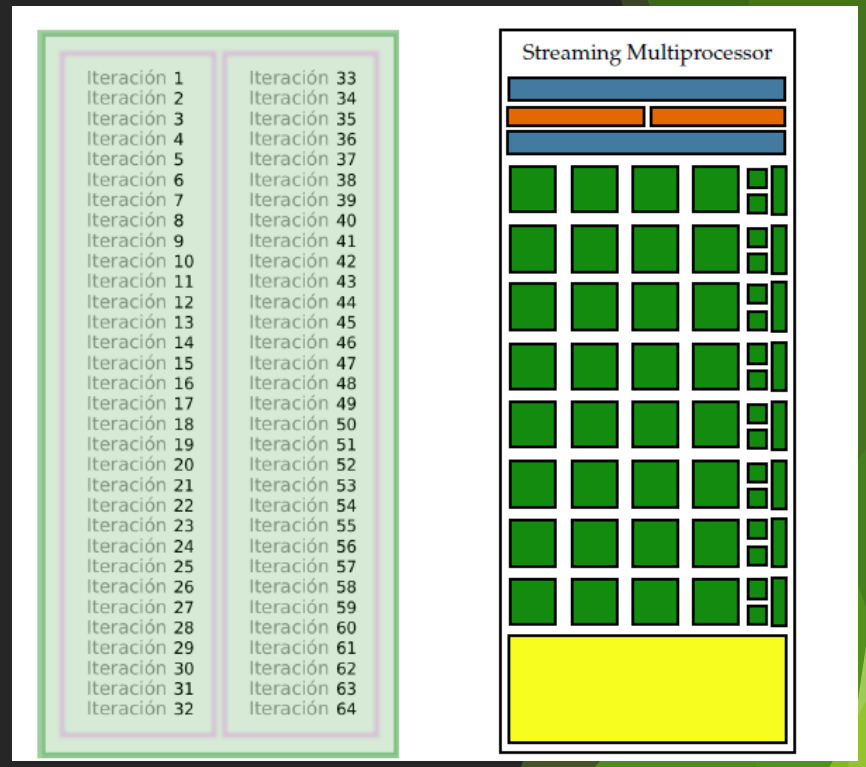


Understanding a loop

- The **grid** contains all the elements to be processed in the GPU

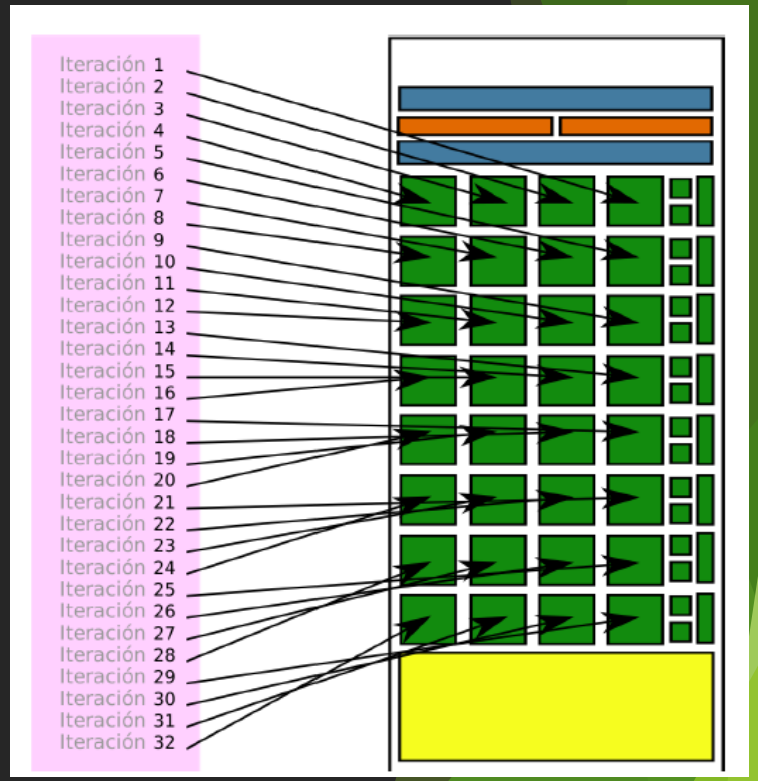
Understanding a loop

- The **block** includes all the elements to be processed in a Streaming Multiprocessor (SM/SMx)



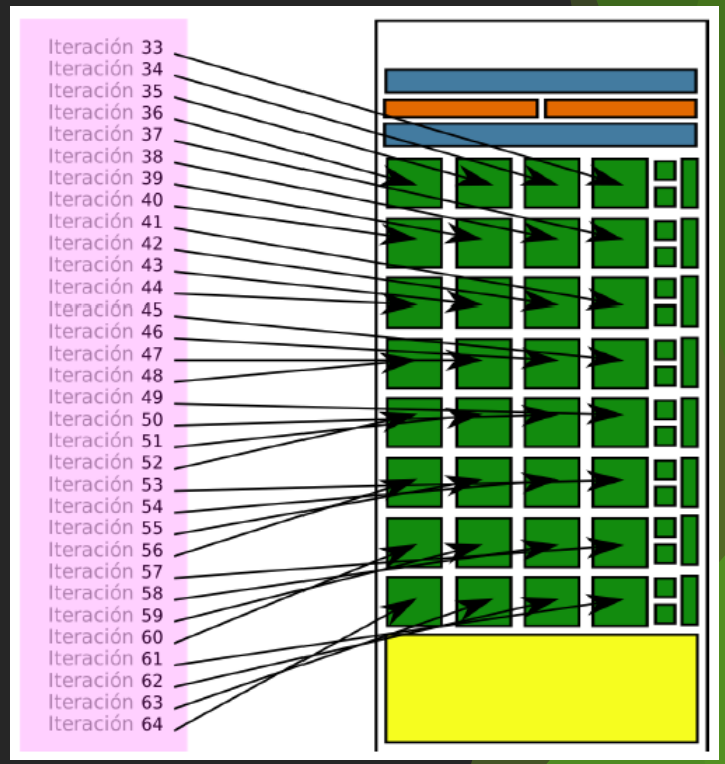
Understanding a loop

- Particulary, **blocks** are executed in sets of 32 elements (**warps**). First a warp...



Understanding a loop

- ...and after the previous one ends, the second warp is executed.



Basic concepts in GPU

- Massive parallelism through thousands of concurrent threads
- Hierarchical organization for efficient execution
- SIMD architecture for parallel data processing

Thread

Smallest unit of execution. Each thread processes a single data element in parallel.

Warp/wavefront

Group of threads (typically 32) that execute the same instruction simultaneously (SIMD)

Block

Collection of threads that can share resources and synchronize. Organized in 1D, 2D, or 3D

Grid

Array of blocks that represents the complete parallel computation structure

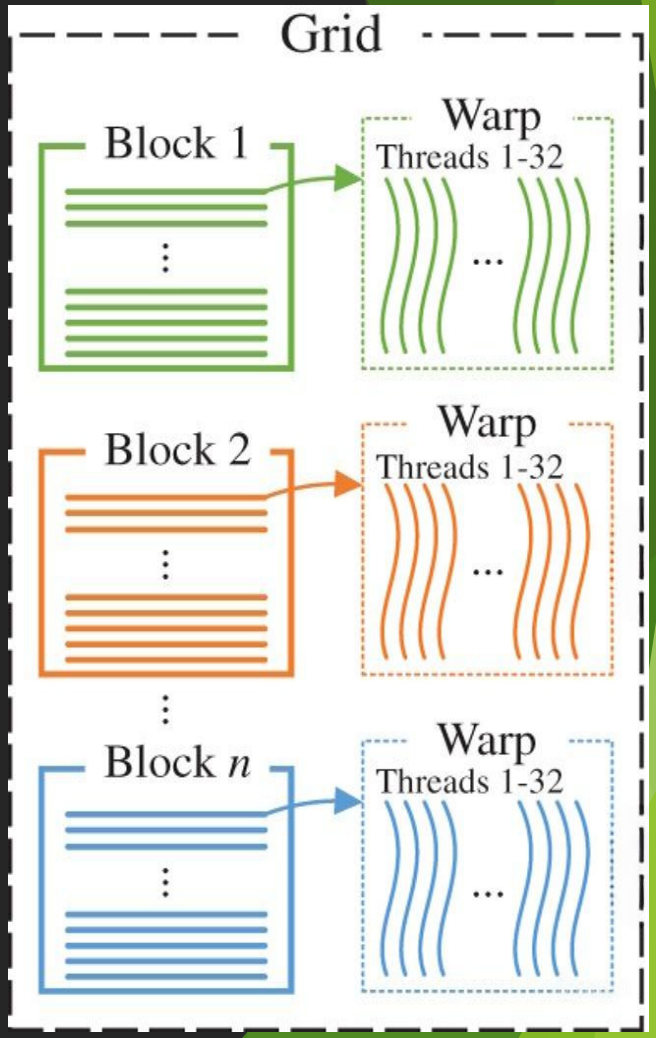
Basic concepts in GPU

Thread
Smallest unit of execution. Each thread processes a single data element in parallel.

Block
Collection of threads that can share resources and synchronize. Organized in 1D, 2D, or 3D

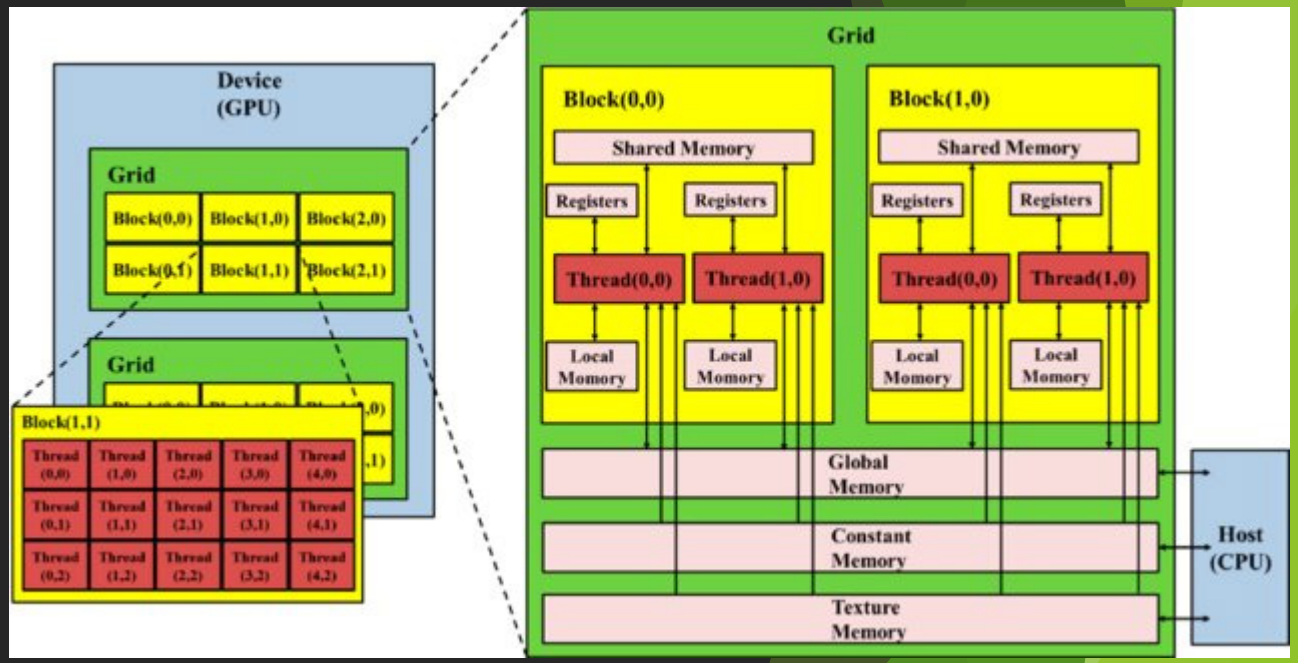
Warp/wavefront
Group of threads (typically 32) that execute the same instruction simultaneously (SIMD)

Grid
Array of blocks that represents the complete parallel computation structure



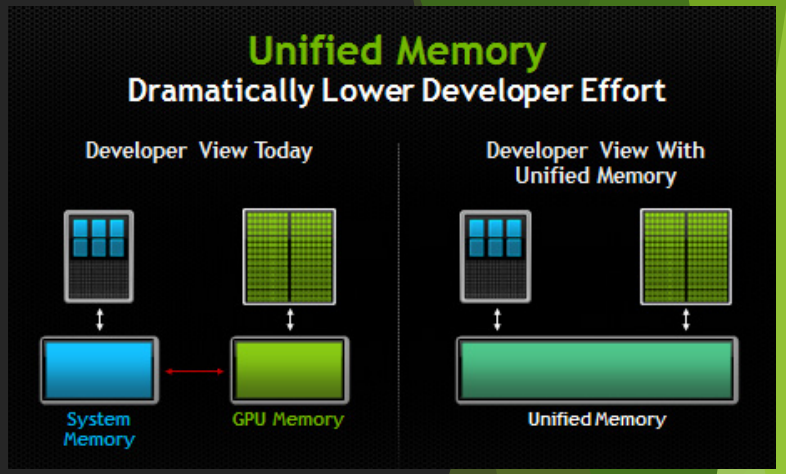
The GPU memory hierarchy

- **Multiple Memory Spaces**: local, global, shared, texture, and constant memories.
- **Local Memory**: Private to each thread.
- **Global Memory**: Accessible by all threads but has higher latency.
- **Shared Memory**: Available to all threads within a block, offers fast access for shared data, reducing access latency compared to global memory.
- **Texture and Constant Memories**: Read-only and accessible by all threads
- **Performance Optimization**: Effective use of memory types can enhance application performance and efficiency.

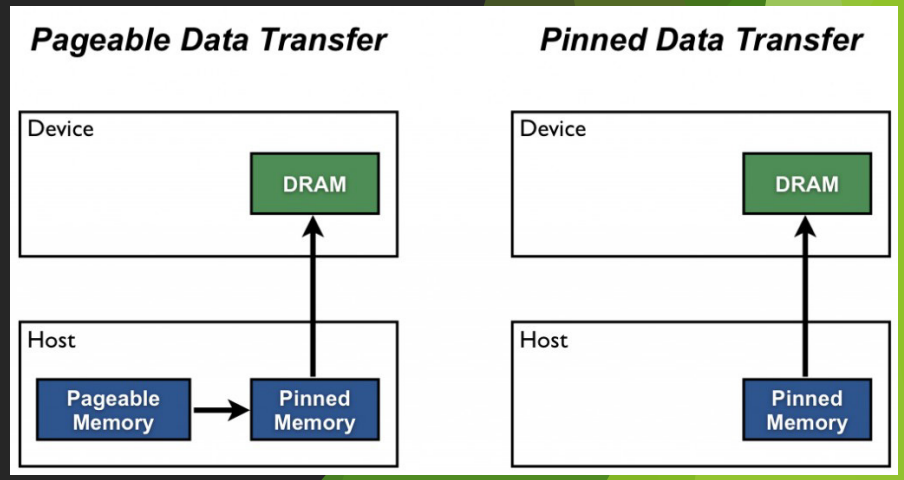


Two specials: unified and pinned memory

- Unified Memory:
  - Shared memory space between CPU and GPU.
  - Simplifies programming by removing explicit memory copies.
  - Useful for irregular memory access patterns.
  - May introduce overhead due to automatic data migration.



- Pinned Memory:
  - Host memory locked in physical memory.
  - Enables faster data transfers between CPU and GPU.
  - Supports asynchronous data transfers.
  - Ideal for frequent and large data transfers.



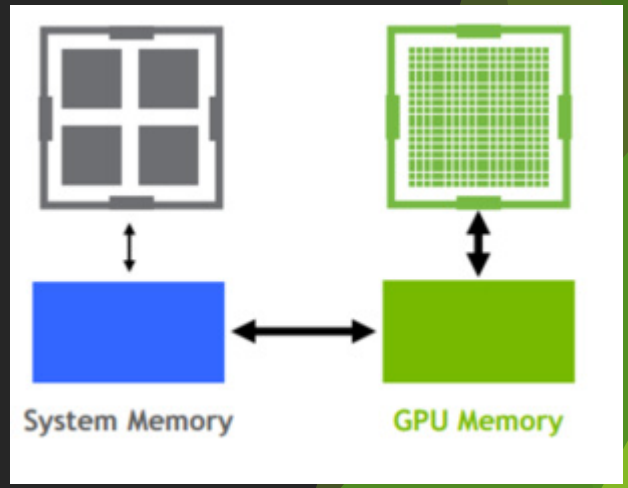
Memory Type	Access Scope	Latency	Bandwidth	Use Case
Local Memory	Private to each thread	Low	High	Temporary variables for individual threads
Global Memory	Accessible by all threads	High	Moderate	Large datasets shared across threads
Shared Memory	Within a thread block	Low	High	Data shared among threads in the same block
Texture/Constant	Read-only by all threads	Moderate	High	Frequently accessed data with spatial locality
Unified Memory	Shared by CPU and GPU	Moderate	Moderate	Simplifies programming by removing explicit memory copies
Pinned Memory	Host memory (CPU)	Low (transfer)	High (transfer)	Faster data transfers between CPU and GPU

Data Transfer CPU ↔ GPU

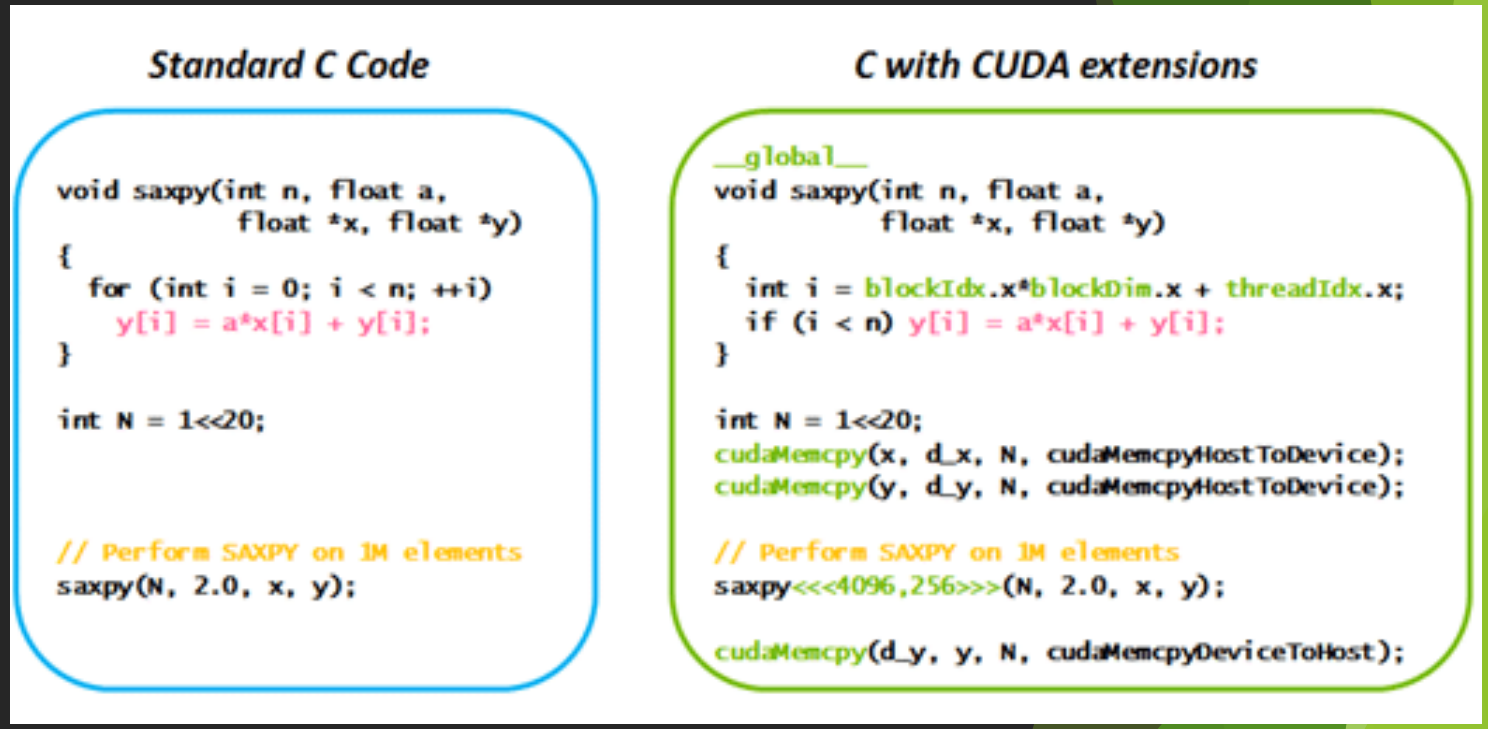
Transfer Methods
- Explicit Memory Copy (cudaMemcpy, hipMemcpy, etc)
- Pinned Memory for faster transfers
- Asynchronous transfers with streams
- Unified Memory (automatic management)

Best Practices
- Minimize CPU-GPU transfers
- Batch small transfers together
- Use async transfers to overlap computation
- Consider using Unified Memory for simplicity

Key Performance Considerations
- Memory transfers are often the bottleneck in GPU applications
- Use shared memory for frequently accessed data
- Coalesce global memory accesses when possible
- Balance register usage vs occupancy
- Profile memory access patterns to identify bottlenecks

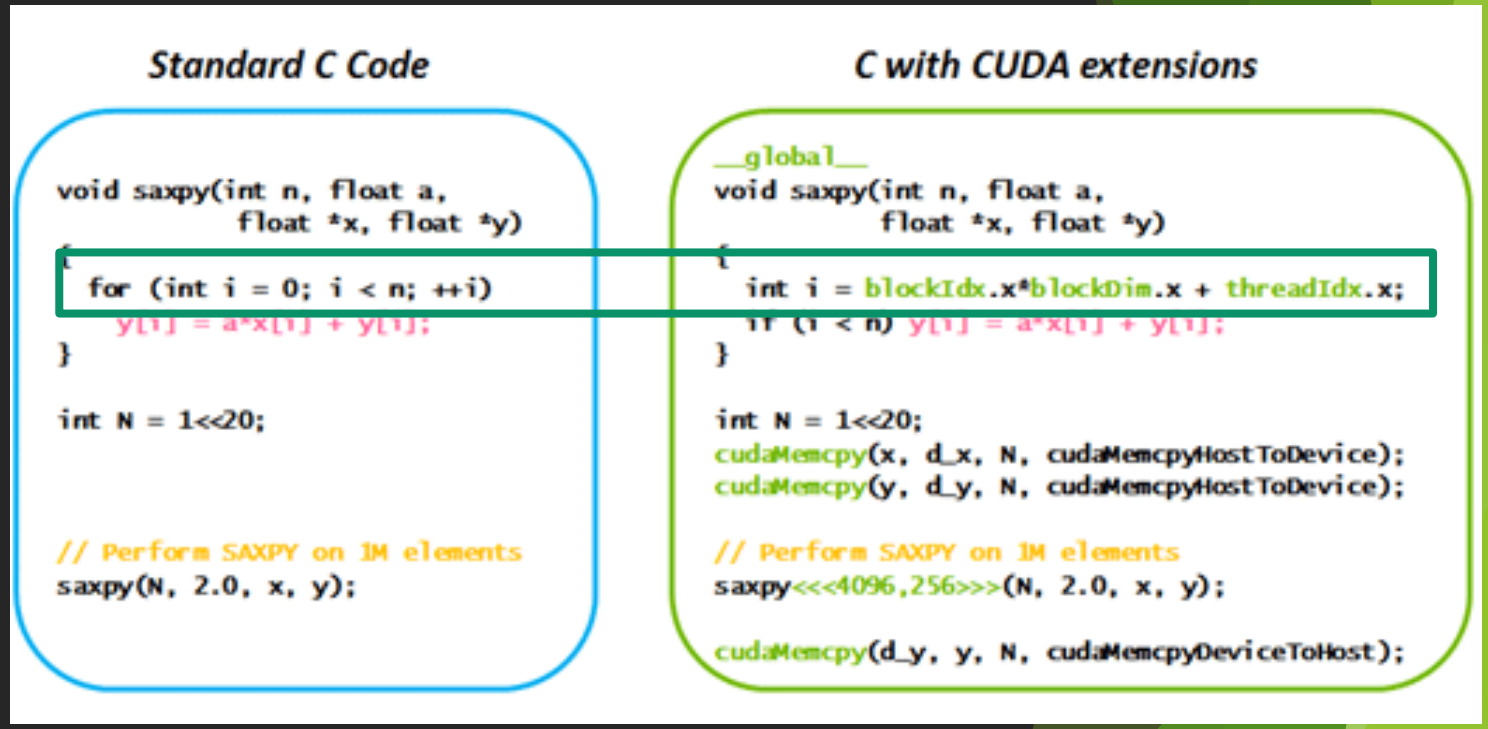


CUDA example



CUDA example

- `for` **`i=1..nElements`**
- `A[i]=...`
- `end for`

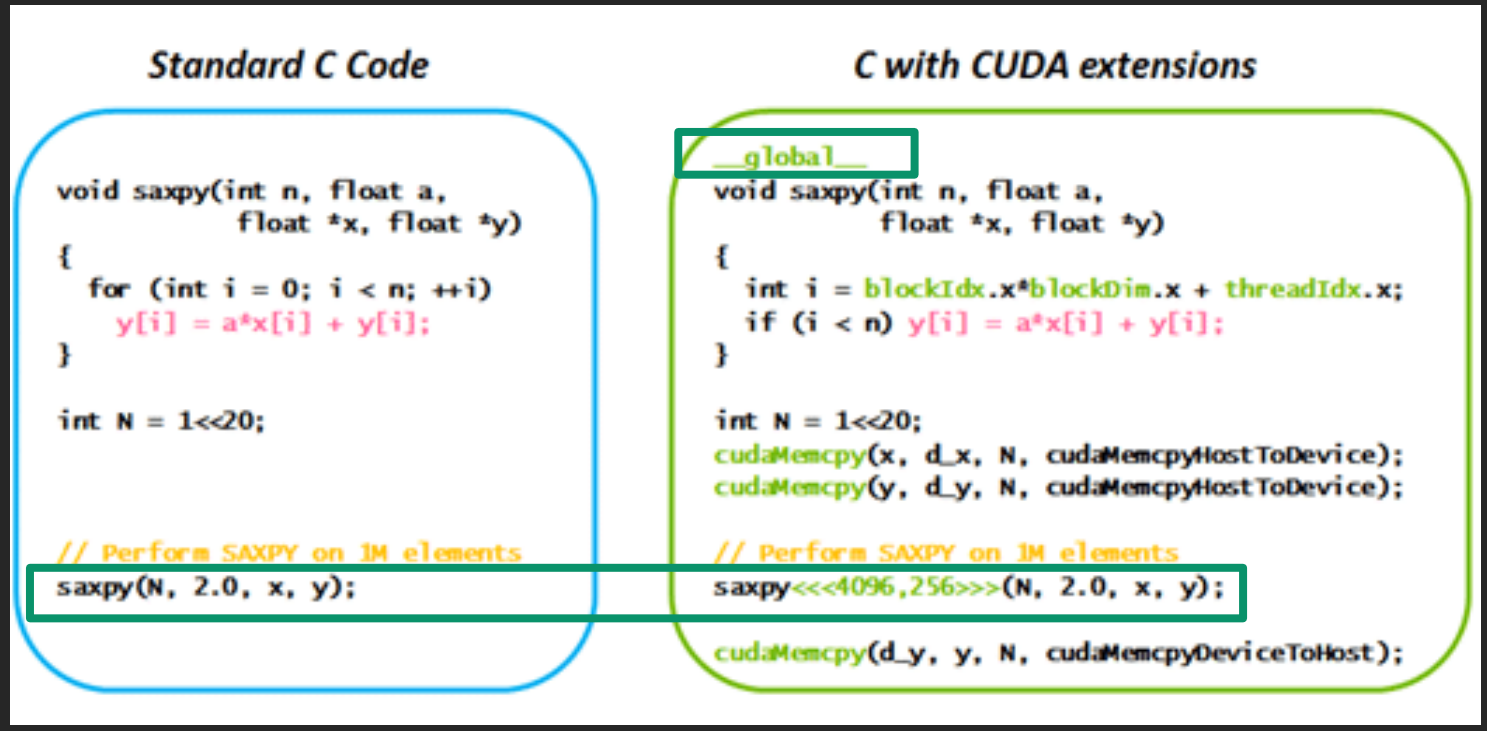


CUDA example

- This is where **thread** appears
  - Each **thread** can establish its unique identifier within the **block**
  - Each **block** can establish its identifier within the **grid**

```
i = threadIdx.x+(blockIdx.x*blockDim.x)
```

CUDA example



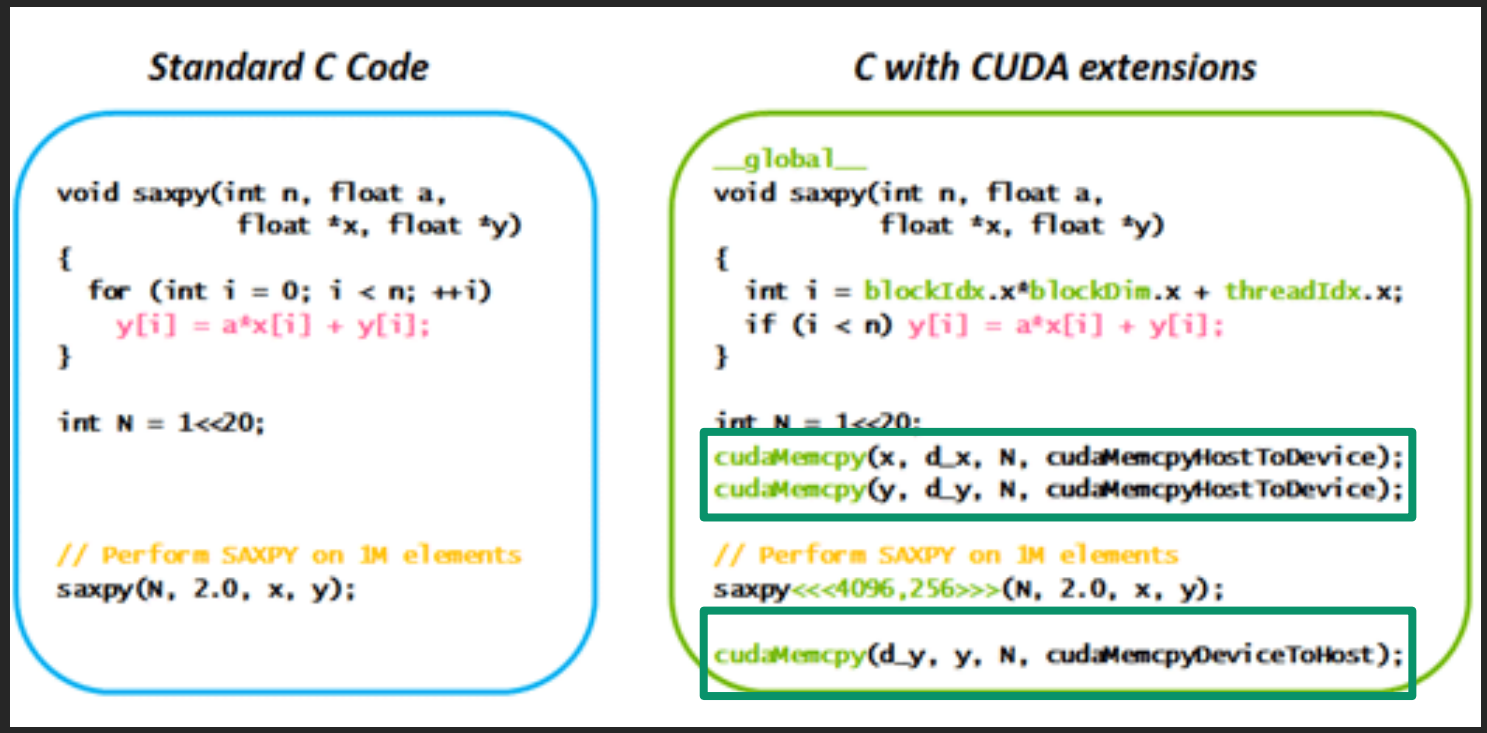
CUDA example

- The kernels are executed N times in parallel by N different CUDA threads
- A kernel is defined using the **__global__** declaration
- The number of CUDA threads that execute that kernel for a given kernel call is specified using a <<< ... >>> execution configuration syntax.

```
int blockSize = 128;
int gridSize = N / blockSize + 1;
or int gridSize = (N + blockSize - 1) / blockSize;
myKernel <<<gridSize,blockSize>>>(arguments)
```

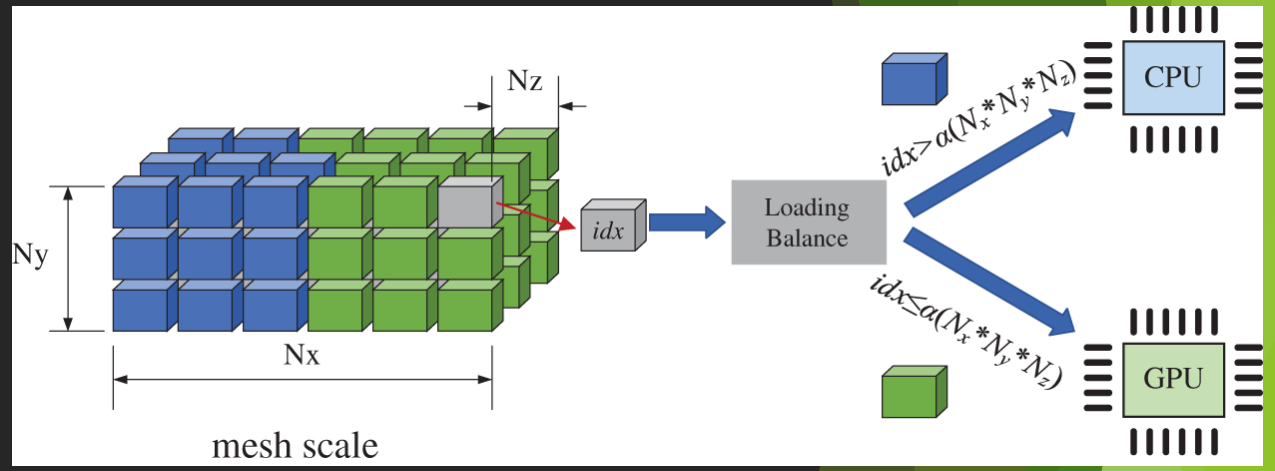
blockSize should be a round multiple of the warp size! It can be tuned to maximize the "occupancy" of the GPU

CUDA example



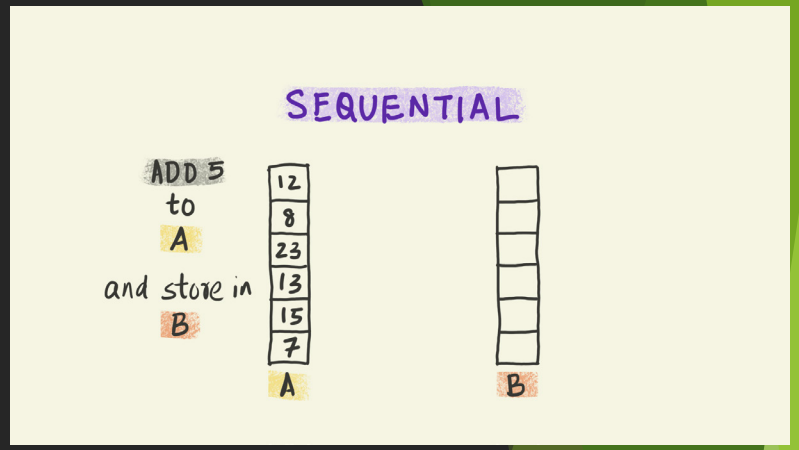
Performance optimization strategies

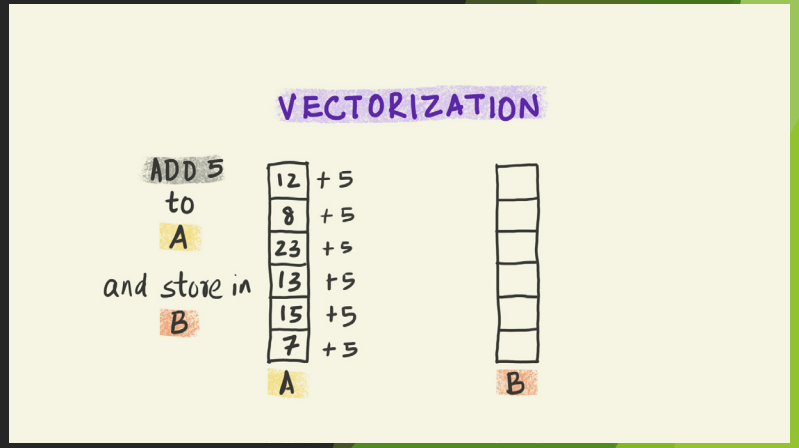
- Vectorization
- Memory access optimization
- Kernel/subroutine implementation
- Thread management and load balancing
- Computation optimization and best practices



Vectorization

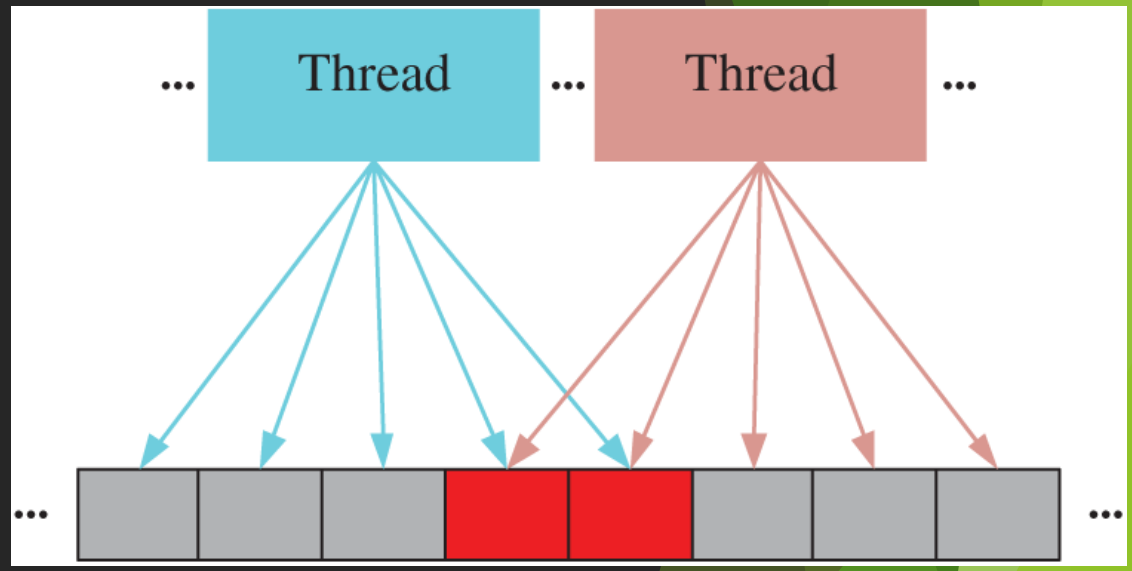
- **Vectorization** refers to the process of organizing computations to take advantage of the SIMD (Single Instruction, Multiple Data) nature of GPU hardware.
- **Data Organization**
  - Structure arrays-of-structures vs **structures-of-arrays**
  - Align data for coalesced memory access
  - Pack data for SIMD operations
- **Computation patterns**
  - Loop unrolling for vector operations
  - Vectorization-Friendly Patterns
  - Replace sequential dependencies with parallel alternatives
  - Use tiling and blocking for better data locality
  - Minimize divergent execution paths





Vectorization & Race conditions

A 'race condition' arises if two or more threads access the same variables or objects concurrently and at least one does updates.



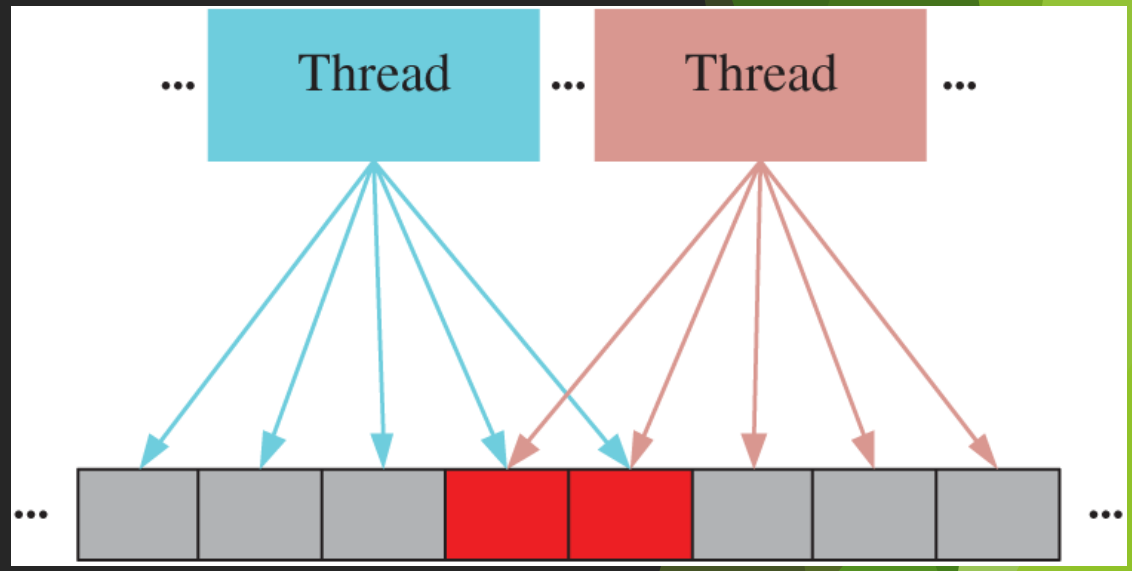
Example: Suppose t_1 and t_2 simultaneously execute the statement x = x + 1; for some static global x.

- Internally, this involves loading x, adding 1, and storing x.
- If t_1 and t_2 do this concurrently, the statement is executed twice, but x may only be incremented once.
- t_1 and t_2 'race' to do the update.

- Initial value of x = 0.
- t_1 reads x (0), adds 1, and stores 1.
- t_2 reads x (0), adds 1, and stores 1.
- Final value of x = 1 (instead of

Vectorization & Race conditions

Common Race Conditions
- Multiple threads updating shared memory
- Concurrent access to boundary elements
- Reduction operations (sum, max, min)

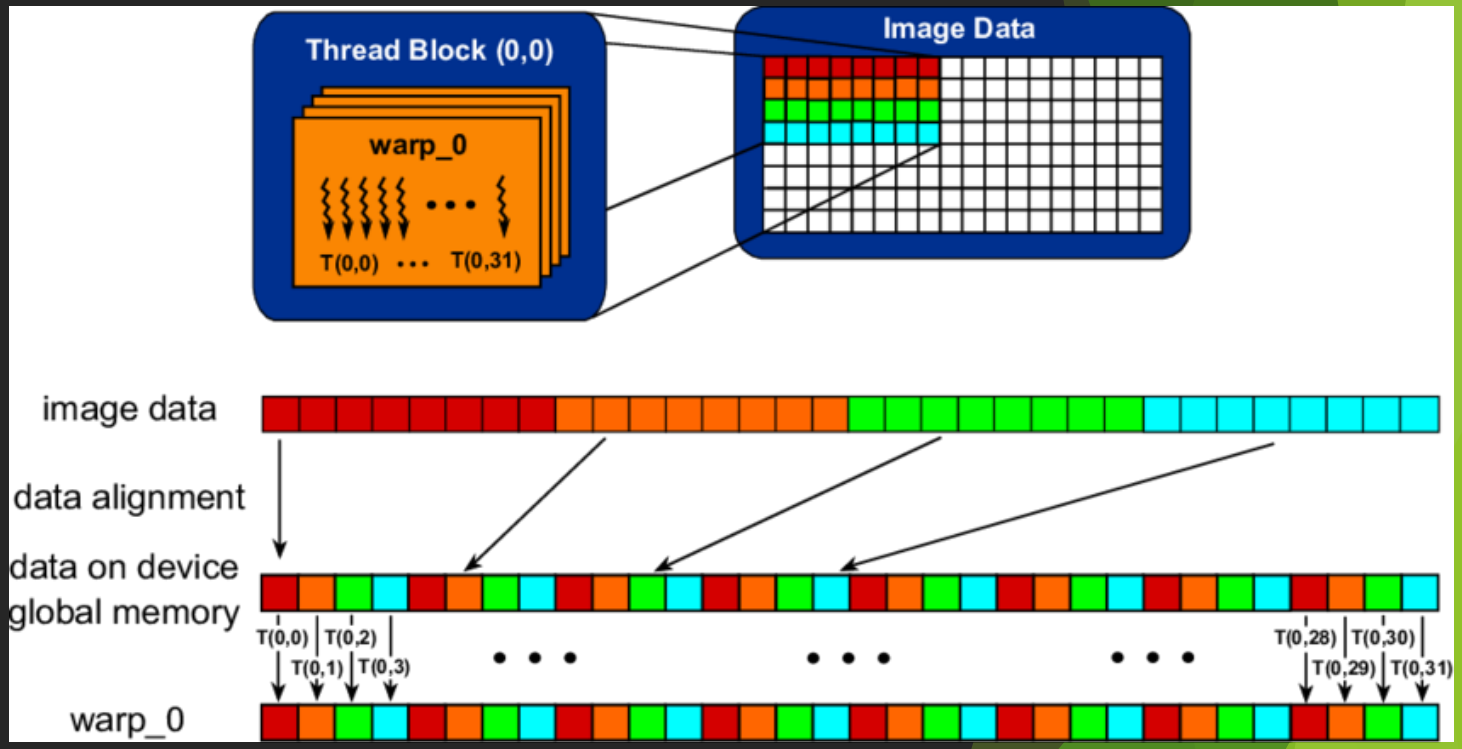


Prevention Strategies
- Atomic operations for critical sections
- Memory barriers and synchronization
- Thread-local storage for intermediate results

Memory access optimization

Coalesced Memory Access

- Adjacent threads should access adjacent memory locations
- Example: Thread 0 → Address N, Thread 1 → Address N+1
- Can improve memory bandwidth utilization by up to 32x



Kernel/subroutine implementation

- Followwed the so-called Embedded Domain Specific Language (eDSL)
- Each module has the same CPU subroutine and CUDA kernel
- During compilation, appropriate computing modules are compiled
- Each kernel is programmed once, avoiding duplicated information
- Readability + reproducibility and trustfulness between different architectures
- Eliminates eventual human mistakes usually made when porting the code

```

//variable definitions
ACTIVE_GPU: flag to enable GPU compilation              ACTIVE_OMP: flag to enable CPU compilation
nrows: number of rows                                  ncols: number of columns
THREAD_BLOCK: group of threads executed in parallel    global_dt: time step size
device_vec: array containing all the GPU vectors       host_vec: array containing all the CPU vectors
H,HU,HV,DEM: integers pointing to the beginning of     id: temporary integer ranging all the cells
the water depth, x-unit discharge, y-unit-discharge
and elevation vectors respectively

//kernel/subroutine call
#ifdef ACTIVE_GPU
  Kernels::wet_dry<< <(nrows*ncols+THREAD_BLOCK-1)/THREAD_BLOCK,THREAD_BLOCK,0,streams>> >(nrows*ncols,
     nrows, ncols, global_dt, device_vec[H], device_vec[HU], device_vec[HV], device_vec[DEM]);
#else
  Kernels::wet_dry(nrows*ncols, nrows, ncols, global_dt, host_vec[H], host_vec[HU], host_vec[HV],
     host_vec[DEM]);
#endif

//kernel/subroutine declaration
  template<typename T>
#ifdef ACTIVE_GPU
  __global__
#endif
  void wet_dry(int size, int nrows, int ncols, T dt, T *h_arr, T *hu_arr, T *hv_arr, T *dem)
  {
  #ifdef ACTIVE_GPU
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id >= size)
      return;
  #else
    #pragma omp parallel for
    for (int id = 0; id < size; id++)
    {
  #endif
        ....................................
        //Kernel/subroutine implementation. Note that this is common for both architectures
        ....................................
#ifdef ACTIVE_OMP
    }
#endif
  }
```

Thread management

Thread Block Optimization

- Optimal block size: Multiple of 32 (warp size)
- Common sizes: 128, 256, or 512 threads per block
- Balance block size with resource usage

Warp Management

- Warp Size: 32 threads execute in lockstep (NVIDIA)
- Avoid warp divergence in conditional statements
- Keep warps fully occupied when possible

Common Pitfalls

- Thread divergence in loops with varying iterations
- Unbalanced workload distribution
- Excessive synchronization points

Performance Tips

- Use occupancy calculator for optimal configuration
- Monitor warp execution efficiency
- Consider dynamic parallelism for irregular workloads

Load balancing

Static Distribution
- Equal-sized blocks for uniform work
- Ideal for regular grid computations
- Example: Basic fluid grid calculations

Dynamic Distribution
- Adaptive work assignment
- Work queues and task pools
- Example: Adaptive mesh refinement

Common Load Imbalance Scenarios
- Boundary Regions
- Different computation requirements at boundaries vs. interior
- Interface Tracking
- Varying computational intensity near interfaces
- Adaptive Time Stepping
- Different regions requiring varied temporal resolution
- Complex Physics Regions
- Areas with additional physical phenomena

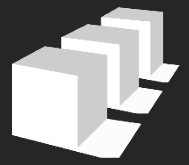
Optimization Techniques
- Task merging
- Combine small tasks to reduce overhead
- Work stealing
- Dynamic redistribution of tasks
- Persistent threads
- Maintain active threads for varying workloads

Computation optimization and best practices

- Maximize computations per memory access
- Reduce redundant calculations by precomputing constants
- Overlap independent instructions to maximize throughput
- Avoid dependencies between consecutive instructions
- Use compiler optimizations to schedule instructions efficiently
- Unroll loops to reduce overhead and improve performance
- Minimize loop-carried dependencies
- Use shared memory for intermediate results in iterative computations
- Balance computation and memory access
- Avoid excessive synchronization points
- Profile kernels to identify hotspots

Universidad Zaragoza

1542



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

GPU programming foundations and performance optimization

Mario Morales Hernández (mmorales@unizar.es)

March18 2025