All you need to know about GPUs

Understand and using GPUs

Lucas NETO
LI-PaRAD Laboratory
Université de Versailles – Saint-Quentin-en-Yvelines

May 14, 2025

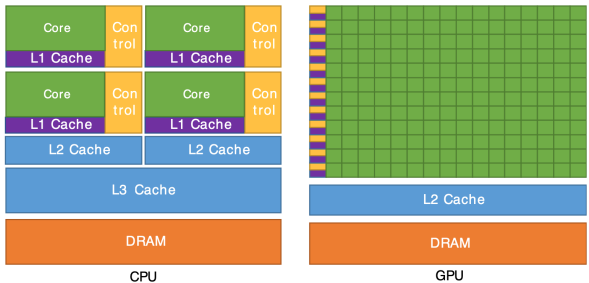
Introduction: Overview

Section 1

Introduction

Introduction: Why GPUs ?

GPUs (Graphics Processing Units) are powerful parallel processors originally designed for rendering graphics but now widely used in HPC and AI.



GPUs offer massive parallelism : GPUs have thousands of smaller, efficient cores designed to handle multiple tasks simultaneously.
But GPUs need to transfer memory : Moving data between CPU and GPU memory has a cost and can become a bottleneck.

Introduction: Kernel definition

A kernel is a function written to run on the GPU. It defines the instructions that every GPU thread will execute.

Think of it like this :

You want to offload a loop to the GPU for parallel execution. Each iteration of the loop will be represented by the kernel, where the loop index i corresponds to the built-in thread identifier TID provided by the GPU programming API.

Examples

If you're adding two large arrays element-wise, the kernel is the function that performs C[tid] = A[tid] + B[tid] for each thread tid, but thousands of indices run at once.

Introduction: Memory transfers

Memory transfer refers to moving data between the host (CPU) and the device (GPU). Before the GPU can work on data, you need to:

- Transfer data from CPU to GPU (host to device).
- Run the kernel on the GPU.
- Transfer results back to CPU (device to host).

Why it matters :

Memory transfers has a cost and can be slow relative to computation. Minimizing them or overlapping them with computation is key to good GPU performance.

Introduction: Terminology

AMD	NVIDIA	Definition
Work-item	Thread	A single and smallest unit of work that cooperates to perform computations.
Workgroup	Block	A collection of wavefronts in a multi-dimensional structure (1D, 2D, or 3D) that can synchronize and share data.
Wavefront	Wrap	A collection of 32 or 64 threads within a block.
Grid	Grid	A collection of thread blocks/workgroups arranged in a multi-dimensional structure (1D, 2D, or 3D).
Compute Unit (CU)	Streaming Multi-processor (SM)	A parallel vector processors in a GPU that contain parallel ALUs. All warps in a block are assigned to the same CU/SM.

Table: GPU Terminology

Section 2

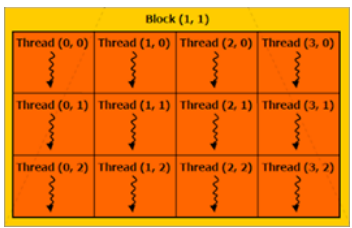
GPU Threading Geometry

GPU Threading Geometry: Thread

- A thread is the smallest unit of execution.
- Each thread runs the same kernel function but works on different data (typically indexed by its Thread ID).
- Thread can be identify with a build-in 1D, 2D or 3D variable named threadIdx (threadIdx.x, threadIdx.y, threadIdx.z)

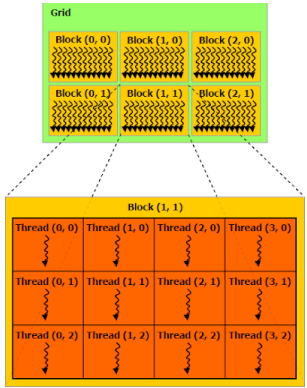
GPU Threading Geometry: Block

- A block is a group of threads.
- Threads in the same block:
  - Can share memory (shared memory).
  - Can synchronize with each other.
- Blocks are independent from each other—no communication between them.
- Blocks can be 1D, 2D, or 3D.
- Block can be identify with a build-in 1D, 2D or 3D variable named blockIdx (blockIdx.x, blockIdx.y, blockIdx.z)



GPU Threading Geometry: Grid

- A grid is a collection of thread blocks.
- Just like blocks can be multi-dimensional, so can grids: 1D, 2D, or 3D.
- You define the grid size when launching the kernel.
- Combined with block size, this defines the total number of threads.



GPU Threading Geometry: Thread Indexing

Each thread on a GPU needs to know which piece of data to work on. You calculate a global index (thread ID) like this:

```
int tidx = blockIdx.x * blockDim.x + threadIdx.x; // i - for 1D problem (e.g vector)
int tidy = blockIdx.y * blockDim.y + threadIdx.y; // j - for 2D problem (e.g matrix)
```

Where:

- threadIdx is the thread's ID within its block
- blockIdx is the block's ID within the grid
- blockDim is the number of threads per block

Constraints

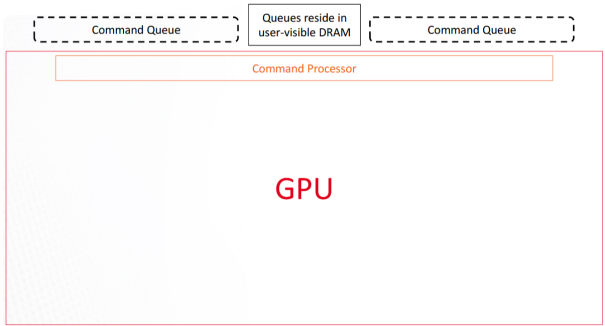
Each hardware has its own constraints, but in general we have those rules :

- Each block cannot have more than 1024 threads in total.
- The maximum dimensions of each block are limited to [1024,1024,1024]

Section 3

AMD GPU architecture

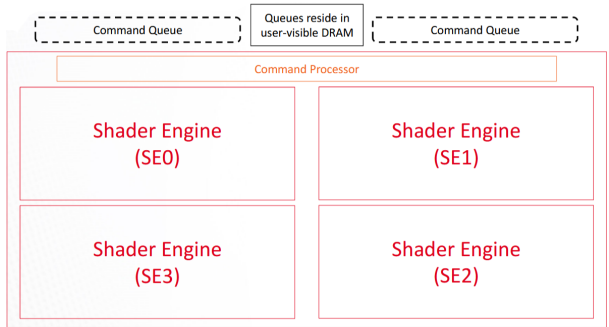
AMD GPU architecture: Global architecture



Command Processor

Translator of higher-level API commands into compute tasks. Compute task are then managed by the Asynchronous Compute Engines (ACE). Each of the ACEs can dispatch block of wavefronts to the Shader Engine. [4]

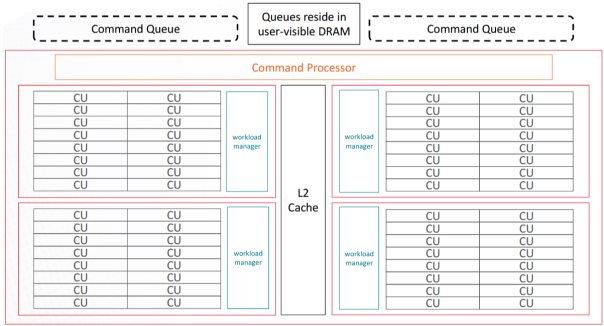
AMD GPU architecture: Global architecture



Shader Engine (or Compute Engine)

Groups of multiple Compute Units, typically sharing some fixed function units or memory resources. [2]

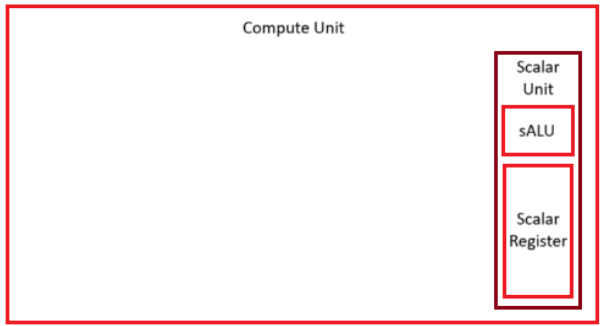
AMD GPU architecture: Compute architecture



Compute Units (CUs)

Basic processing unit. Each CU contains register files and SIMD pipelines optimized for scalar, vector, and matrix instructions. [5] Depending on their resource usage up to thousands of threads can reside on a CU. [3]

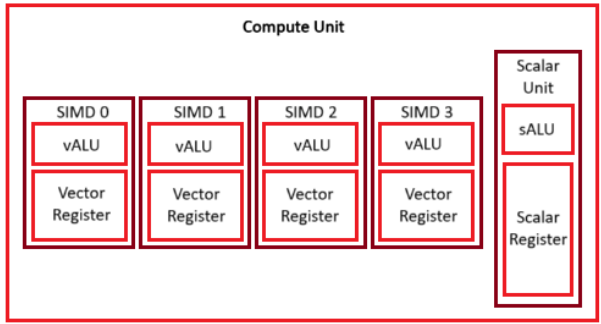
AMD GPU architecture: Compute architecture



Scalar Unit

The scalar unit performs instructions that are uniform within a warp. [3] Used for control flow, pointer arithmetic, dispatch a common constant value, etc.

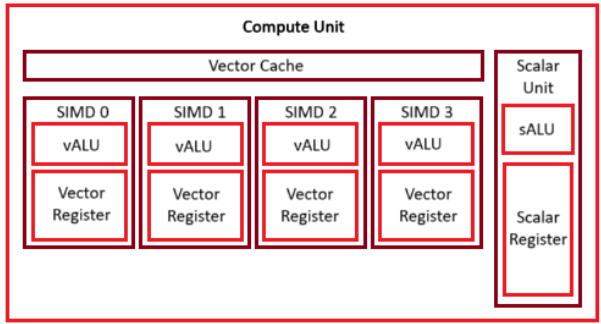
AMD GPU architecture: Compute architecture



SIMDs

Each compute unit is subdivided into four SIMD16 units that process SIMD instructions of 16 data elements per instruction. Each SIMD always executes the same instruction for the whole VALU.[1] Each SIMD have an instruction buffer for 10 wavefronts.

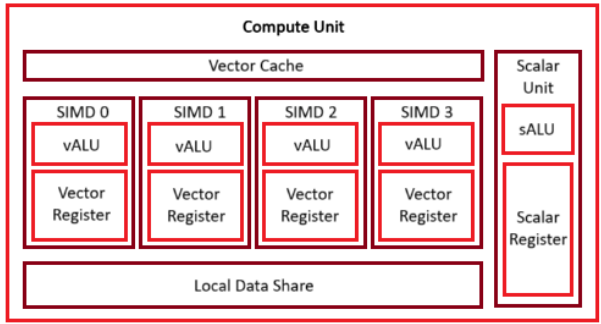
AMD GPU architecture: Compute architecture



Vector Cache

Cache L1 used to coalesce memory accesses of the warps in order to reduce the amount of accesses to device memory, and make that memory available for other warps that currently reside on the compute unit. [1]

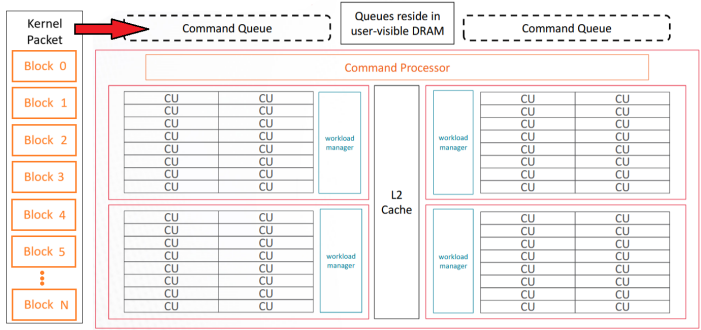
AMD GPU architecture: Compute architecture



Local Data Share

Memory accessible to all threads within a block. Its latency and bandwidth is comparable to that of the vector cache. It can be used to share memory between the threads in a block, or as a software managed cache. [1]

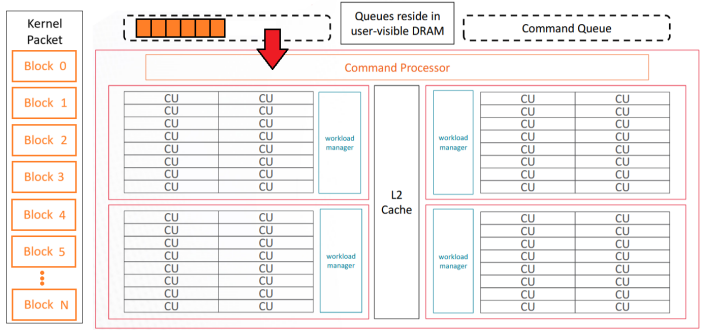
AMD GPU architecture: Kernel execution



Step 1

A kernel packet that contains data such as arguments' address, dimension size, memory size, etc. is enqueued.

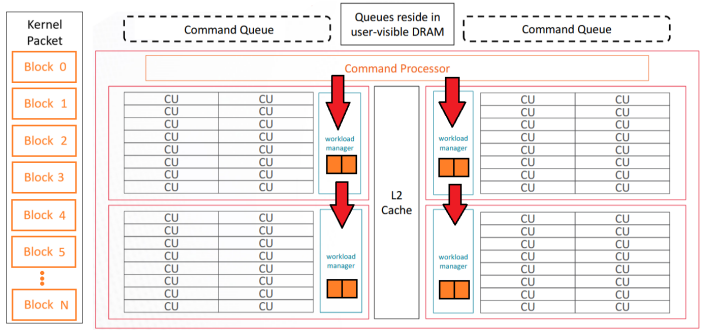
AMD GPU architecture: Kernel execution



Step 2

The command processor receives the kernel launch commands and fragments the kernel into blocks of wavefronts according to the developer directive.

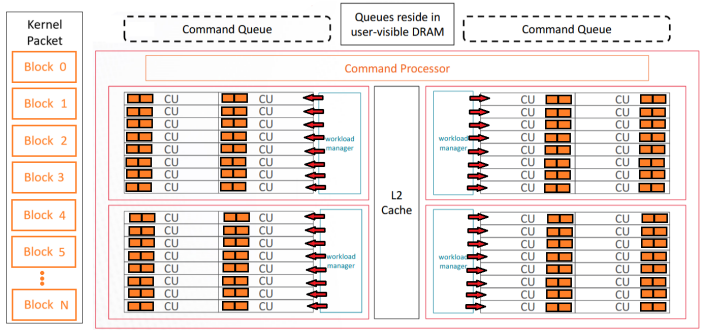
AMD GPU architecture: Kernel execution



Step 3

Those blocks are dispatched to the Compute/Shader Engine

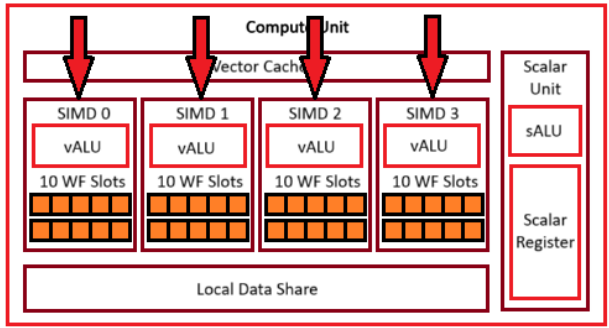
AMD GPU architecture: Kernel execution



Step 4

Thread blocks are scheduled on CUs. Each block is assigned to an individual CU, and a CU can accommodate several blocks.

AMD GPU architecture: Kernel execution



Step 5

The threads are dispatched in wavefronts to the available SIMDs in each CU. Each SIMD has a HW defined number of slots available for assigned wavefronts. So, multiple wavefronts can be assigned to a single SIMD but only one wavefront can be executed at a time on the SIMD.

AMD GPU architecture: Kernel execution

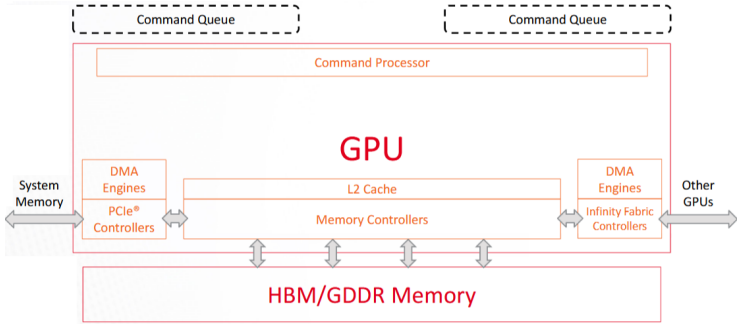
The GPU can execute any assigned wavefront on a given SIMD, switching between them as necessary during execution.

Examples

If wavefront A is picked up by the SIMD and begins performing ALU computations, it might be block to fetch data from memory, causing a latency. To mitigate this latency, the GPU can switch contexts and execute another wavefront B from the available slots during the fetch.

Context switching between warps residing on a CU incurs no overhead, as the context for the warps is stored on the CU and does not need to be fetched from memory.

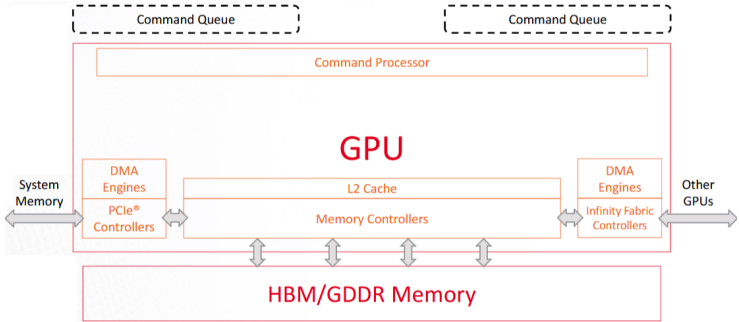
AMD GPU architecture: Memory architecture



PCIe Controllers

PCIe is the high-speed interface that connects the GPU to the CPU and system memory. Think of it as the "data highway" between your GPU and the rest of your computer. PCIe speed (e.g., Gen 4.0, Gen 5.0) can significantly affect data transfer performance.

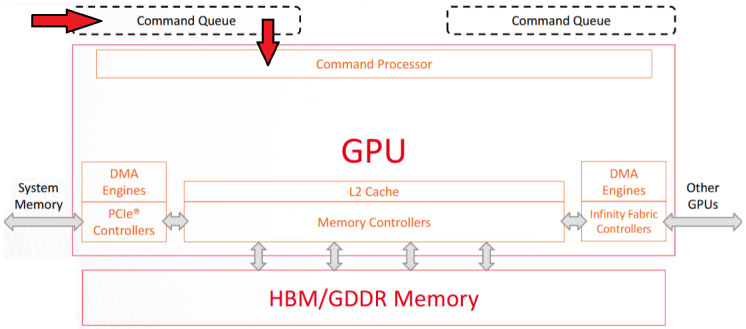
AMD GPU architecture: Memory architecture



DMA Engines

DMA engines allow data to be moved between host and GPU memory without involving the CPU. This means transfers can happen in the background, freeing up the CPU to do other tasks. GPUs have multiple DMA engines for asynchronous transfers (e.g., copying data HtD and DtH at the same time).

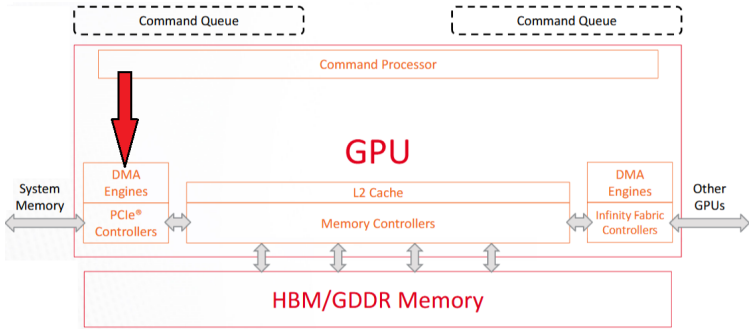
AMD GPU architecture: Memory transfers



Step 1

Enqueue memory transfers packet that contains data such as memory address, DMA engine to use, size etc. and CP reads the packet.

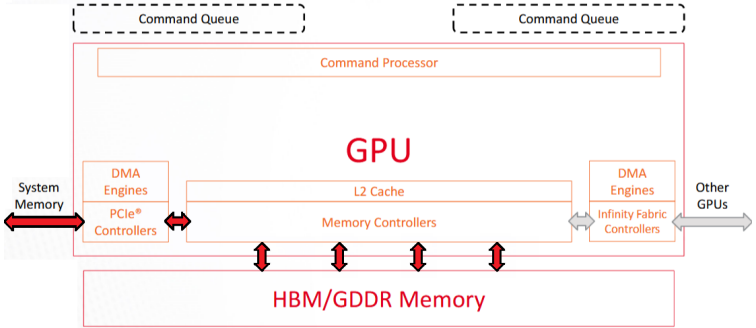
AMD GPU architecture: Memory transfers



Step 2

CP transfers the command to the corresponding DMA engine.

AMD GPU architecture: Memory transfers



Step 3

DMA Engine transfers data through PCIe to/from CPU. This can take place in parallel with other computation work or with transfers from another DMA.

Using GPUs: APIs

NVIDIA :

- CUDA,
- HIP (used as wrapper for CUDA)
- OpenMP
- OpenACC

AMD :

- HIP
- OpenMP
- OpenACC (with Cray compiler)
- HSA (low level library used behind all of the above)

Using GPUs: CUDA/HIP

- CUDA : Proprietary API developed by NVIDIA
- HIP : Open-source API developed by AMD
- Provides direct access to GPU features: memory management, kernel launching, thread hierarchy, etc.
- HIP code is very similar to CUDA
- HIP code can run on both AMD and NVIDIA GPUs
- Requires writing custom code using the CUDA/HIP API and managing resources explicitly.
- Requires deeper understanding of hardware details.

Using GPUs: CUDA/HIP GPU Code

Examples

Kernel written in CUDA/HIP :

```
__global__ void add (int *a, int *b, int *c, int N) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

- '`__global__`' qualifier indicates that this function will be executed on the GPU and can be called from the host (CPU) code.
- Each thread will calculate its unique index tid based on its thread index and block index
- The if statement ensures that the thread operates only within the array bounds (N).

Using GPUs: CUDA/HIP CPU Code

First allocate and initialize host data :

Examples

```
int size = sizeof(int) * n;
int *a = malloc_and_init(size);
int *b = malloc_and_init(size);
int *c = malloc(size);
```

Using GPUs: CUDA/HIP CPU Code

Then allocate memory on GPU for device data :

Examples

```
int *d_a, *d_b, *d_c;

[cuda|hip]Malloc(&d_a, size);
[cuda|hip]Malloc(&d_b, size);
[cuda|hip]Malloc(&d_c, size);
```

Using GPUs: CUDA/HIP CPU Code

Finally copy data from CPU to GPU, execute kernel with parameters and copy back the result:

Examples

```
[cuda|hip]Memcpy(d_a, a, size, [cuda|hip]MemcpyHostToDevice);
[cuda|hip]Memcpy(d_b, b, size, [cuda|hip]MemcpyHostToDevice);

add<<<dim3(ceil(n / 128)), dim3(128), 0, 0>>>(d_a, d_b, d_c);

[cuda|hip]Memcpy(c, d_c, size, [cuda|hip]MemcpyDeviceToHost);
```

Using GPUs: CUDA/HIP CPU Code

Don't forget to free your memory:

Examples

```
[cuda|hip]Free(d_a);
[cuda|hip]Free(d_b);
[cuda|hip]Free(d_c);

free(a);
free(b);
free(c);
```

Using GPUs: CUDA/HIP Kernel launch syntax

The '<<<...>>>' syntax in used to launch kernel in both CUDA and HIP API.

Examples

```
add<<<dim3(ceil(n / 128)), dim3(128), 0, 0>>>(d_a, d_b, d_c);
```

This syntax need 4 arguments :

- GridDim : The grid dimension (number of block per dimension) as a integer (if 1D) or a dim3(x,y,z) value.
- BlockDim : The block dimension (number of thread per block) as a integer (if 1D) or a dim3(x,y,z) value.
- SharedMem : Specifies how much size shared memory each block should get.
- Stream : Specifies the stream ID to which the kernel launch belongs. A stream is essentially a sequence of operations that are executed in order on the GPU. By using different streams, you can achieve asynchronous execution of multiple tasks (e.g., kernels, memory copies) on the GPU.

Using GPUs: OpenMP

OpenMP and HIP/CUDA are very different :

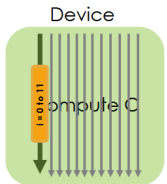
- OpenMP uses directives.
- OpenMP works on both CPUs and GPUs with the same source code (the offloaded one).
- OpenMP can be use with AMD and NVIDIA GPUs.
- OpenMP uses a higher-level system of thread grouping : teams (no blocks, no grids).
- OpenMP need minimal changes to existing code to be offloaded.
- OpenMP can manages task distribution and resource allocation automatically.

Using GPUs: OpenMP - How to offload

Target directive :

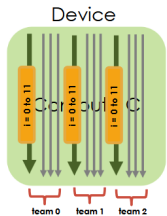
target

```
#pragma omp target
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



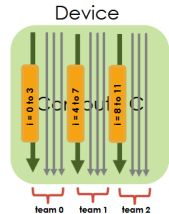
target teams

```
#pragma omp target teams
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



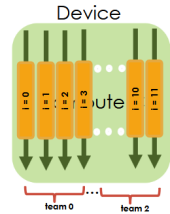
target teams distribute

```
#pragma omp target teams
distribute num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



target teams distribute parallel

```
#pragma omp target teams
distribute parallel for
num_teams(3)
for (int i = 0; i < 12; ++i)
{
    C[i] = A[i] + B[i];
}
```



Using GPUs: OpenMP - How to copy memory

Many ways to transfers memory :

- Using map clause to target directive : map([[map-type-modifier[,] [map-type-modifier[,] ...]] map-type:] locator-list)
- Using OpenMP routines : `omp_target_alloc`, `omp_target_memcpy_async`, etc. Developers need to manage resource allocation and transfers on their own. Also, target directive will need the `is_device_ptr` clause to tell the GPU that pointers for the task are already on the GPU.
- Using CUDA/HIP resources management ([cuda—hip]Memcpy, etc.). Similar to OpenMP routines.

Using GPUs: OpenMP - How to copy memory : Map clause

```
#pragma omp target map(to: a[0:N]) map(from: b[0:N])
{
    // Offloaded kernel region with 'a' and 'b'
}
```

The data is moved to the device before execution or moved back to the host after target region execution.

Examples

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams distribute parallel for simd \
    num_teams(num_blocks) map(to:x[0:sz]) map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++)
        y[i] = a * x[i] + y[i];
}
```

Using GPUs: OpenMP - How to copy memory : Map clause

Examples

```
#pragma omp target data map(tofrom: a[0:N]) map(tofrom: b[0:N])
{
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < N; i++) {
        // Device Kernel Computation
    }

    // Host computation

    #pragma omp target {
        // Device Kernel Region
    }
}
```

A data region is defined with the '{}'.
The data is moved to the device and stay until the end of the data region.
The offloaded data can be used by all device kernel inside the data region.

Using GPUs: OpenMP - How to copy memory : Map clause

Examples

```
# pragma omp target enter data map(to: a[0:N]) \
                               map(to: b[0:N])

// Device computation
// Host computation
// ...

# pragma omp target exit data map(from: a[0:N]) \
                              map(from: b[0:N])
```

A data region is defined between 'enter' and 'exit' clauses.
Same behavior as previous.

Using GPUs: OpenMP - How to copy memory : Routines

Examples

```
int device_id = omp_get_default_device();
int host_id   = omp_get_initial_device();

double *a_dev = omp_target_alloc(N * sizeof(double), device_id);
double *b_dev = omp_target_alloc(N * sizeof(double), device_id);

omp_target_memcpy(a_dev, a_host, N * sizeof(double),
                  0, 0, device_id, host_id);  // host to device

// Offload kernel using device pointers
...

omp_target_memcpy(b_host, b_dev, N * sizeof(double),
                  0, 0, host_id, device_id);  // device to host

omp_target_free(a_dev, device_id);
omp_target_free(b_dev, device_id);
```

Using GPUs: OpenMP - How to copy memory : Routines

Offloaded kernel will looks like this :

Examples

```
#pragma omp target is_device_ptr(a_dev, b_dev) \
                    device(device_id)
for (int i = 0; i < N; i++) {
    // Computation with 'a_dev' and 'b_dev'
}
```

Conclusion: Summary

Key concepts :

- Thread and Block Geometry
- Kernel and Memory transfers
- CUDA / HIP : Provides low-level control and maximum performance, but requires more effort.
- OpenMP : Easier to use, cross-platform, and ideal for general parallel programming.

Conclusion: To go further

More details about AMD GPU architecture and HIP programming in: `https://youtube.com/playlist?list=PLB1fSi1mbw6IKbZSPz9a2r2DbnHWnLbF-&feature=shared`

Conclusion: References

[1] AMD. ROCm Documentation. URL: https://rocm.docs.amd.com/en/docs-5.7.1/understand/gpu_arch/mi250.html.

[2] AMD. ROCm HIP Documentation. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/understand/programming_model.html.

[3] AMD. ROCm HIP Documentation. URL: https://rocmdocs.amd.com/projects/HIP/en/latest/understand/hardware_implementation.html.

[4] AMD. CDNA1 White Paper. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna-white-paper.pdf.

[5] AMD. CDNA2 White Paper. URL: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.

The End