GPU Computations

Ilia Nechaev

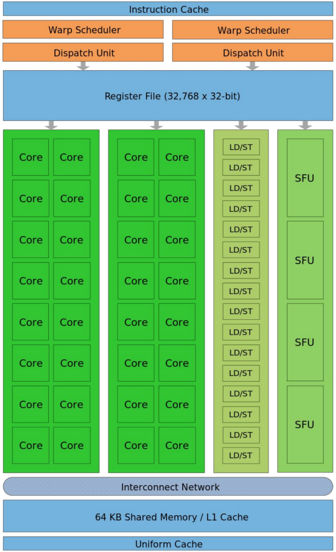
10.01.2025

Overview

GPU vs CPU: Main Differences

- **CPU (Central Processing Unit):**
  - Optimized for low latency and complex tasks
  - Few powerful cores (usually 4–16)
  - Large cache memory per core
  - Good at serial (sequential) processing
- **GPU (Graphics Processing Unit):**
  - Optimized for high throughput
  - Hundreds or thousands of simpler cores
  - Highly parallel architecture
  - Single instruction pointer per warp/wavefront
  - Cache per warp/wavefront, but without synchronisation between them

GPU Architectures



Wavefront and Other Vendor Terminology

- **Wavefront (AMD)**: The set of work-items executed in lockstep (often 32 or 64 threads)
- **Warp (NVIDIA)**: Similar concept to Wavefront; typically 32 threads
- **Wave (Intel)**: May use different names or sizes depending on the architecture
- Conceptually, these terms all refer to a hardware-level group of threads that share instruction decoding

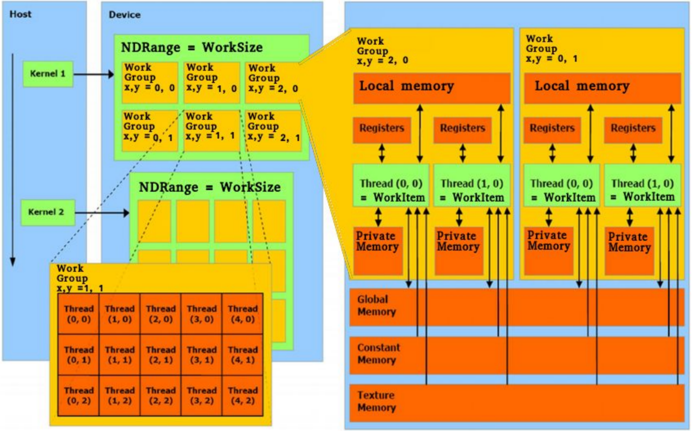
Why Massive Parallelism on GPUs?

- GPUs can execute thousands of threads simultaneously
- Hardware schedulers distribute threads among available cores
- Fine-grained parallelism allows splitting large tasks into many small subtasks
- Achieves significant speedups for vectorized and matrix-based operations

Basic OpenCL Terminology

- **Kernel**: A function executed on a device (GPU) across a set of work-items
- **Work-Item**: A single execution instance of a kernel
- **Work-Group**: A group of work-items that execute together on a compute unit
- **Global Size**: The total number of work-items
- **Local Size**: The number of work-items in a single work-group

OpenCL approach



```c
#define VALUES_PER_WORK_ITEM 32
#define WORKGROUP_SIZE 32

__kernel void atomic_sum(__global const int *arr,
                         __global unsigned int *sum,
                         unsigned int n)
{
    unsigned int id = get_global_id(0);
    if (id < n)
    {
        atomic_add(sum, arr[id]);
    }
}

__kernel void loop_sum(__global const int *arr,
                       __global unsigned int *sum,
                       unsigned int n)
{
    const unsigned int idx = get_global_id(0);
    unsigned int res = 0;
    for (int i = idx * VALUES_PER_WORK_ITEM; i < (idx + 1) * VALUES_PER_WORK_ITEM; ++i)
    {
        if (i < n)
        {
            res += arr[i];
        }
    }

    atomic_add(sum, res);
}
```

What just happened?

Important syntax

- **__kernel**: Specifies a kernel function
- **__global**: Denotes a pointer to global memory
- **__local**: Denotes a pointer to local memory (cache)
- **barrier(int flag)**: Synchronizes work-items within a work-group
  - **CLK_LOCAL_MEM_FENCE**: Synchronizes local memory
  - **CLK_GLOBAL_MEM_FENCE**: Synchronizes global memory
- **get_global_id(int dim)**: Returns the global ID of the current work-item
- **get_local_id(int dim)**: Returns the local ID of the current work-item
- **get_group_id(int dim)**: Returns the group ID of the current work-item

Coalesced Memory Access

When different work items access consecutive memory locations, the GPU can combine these requests into a single memory transaction. This is called **coalesced memory access**.

Example: Sum of vector elements (Coalesced vs Non-coalesced Access)

```
__kernel void loop_sum
(
    __global const int *arr
  , __global unsigned int *sum
  , unsigned int n
)
{
  const unsigned int idx = get_global_id(0);
  unsigned int res = 0;
  for (int i = idx * VALUES_PER_WORK_ITEM; i <
        (idx + 1) * VALUES_PER_WORK_ITEM; ++i)
  {
    if (i < n)
    {
      res += arr[i];
    }
  }

  atomic_add(sum, res);
}
```

```

__kernel void loop_coalesced_sum
(
    __global const int *arr
  , __global unsigned int *sum
  , unsigned int n
)
{
  const unsigned int lid = get_local_id(0);
  const unsigned int wid = get_group_id(0);
  const unsigned int grs = get_local_size(0);

  unsigned int res = 0;
  for (int i = 0; i < VALUES_PER_WORK_ITEM; ++
        i)
  {
    int idx = wid * grs * VALUES_PER_WORK_ITEM
          + i * grs + lid;
    if (idx < n)
    {
      res += arr[idx];
    }
  }

  atomic_add(sum, res);
}
```