



CUDA Managed Memory

Matti Kortelainen

TAC-HEP

1 March 2023

Who am I

- Co-convener of CMS Core Software group, lead of CMS' data processing framework, CMSSW, development
 - Been in CMS since 2008 doing physics analysis, reconstruction software, software performance ...
- PhD on search for light charged Higgs boson from University of Helsinki in 2013
- I have been working with CUDA for 5+ years

Introduction

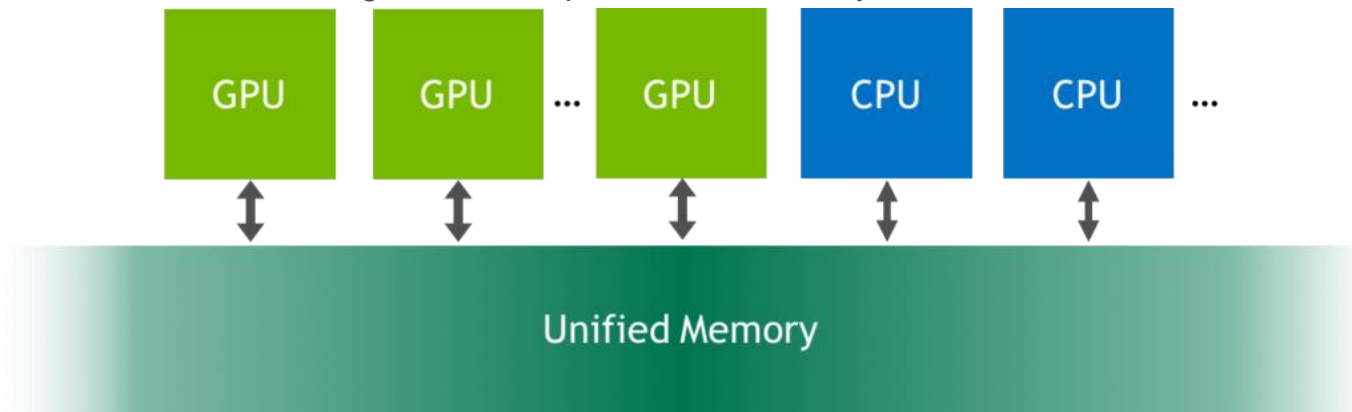
- By now you have learned about
 - Explicit memory management: (pinned) host memory, device memory, memcpy
 - Asynchronicity: execution on host and device proceed independently
--> need to explicitly synchronize host and device
 - Concurrent operations with CUDA streams

Introduction

- By now you have learned about
 - Explicit memory management: (pinned) host memory, device memory, memcpy
 - Asynchronicity: execution on host and device proceed independently
--> need to explicitly synchronize host and device
 - Concurrent operations with CUDA streams
- Looks complicated. Wouldn't it be nice to be able to simplify?

CUDA Unified Memory (or Managed Memory)

- Unified/managed memory or in CUDA is memory that can be accessed on the host and on the device, and the CUDA runtime+driver automatically migrate the memory between the two
 - SYCL calls it "Unified Shared Memory"
 - "Shared memory" in CUDA: fast memory that can be accessed by a threads of the same block, kind of programmable L1 cache
 - "Unified virtual addressing" in CUDA: pinned host memory can be accessed from the device



Simple example

```
__global__ void kernel(int *array, int n) {
    int ind = threadIdx.x + blockIdx.x * blockDim.x;
    if (ind < n) {
        array[ind] *= 2;
    }
}

int main() {
    int n = 128;
    int* array;
    cudaMallocManaged(&array, n);

    for (int i=0; i<n; ++i) {
        array[i] = i*10 - 5;
    }

    kernel<<<1, n>>>(array, n);
    cudaDeviceSynchronize();

    for(int i=0; i<n; ++i) {
        std::cout << i << " " << array[i] << std::endl;
    }

    cudaFree(array);

    return 0;
}
```

Simple example

```
__global__ void kernel(int *array, int n) {  
    int ind = threadIdx.x + blockIdx.x * blockDim.x;  
    if (ind < n) {  
        array[ind] *= 2;  
    }  
}
```

```
int main() {  
    int n = 128;  
    int* array;  
    cudaMallocManaged(&array, n);
```

Memory is allocated with
`cudaMallocManaged()`

```
    for (int i=0; i<n; ++i) {  
        array[i] = i*10 - 5;  
    }
```

```
    kernel<<<1, n>>>(array, n);  
    cudaDeviceSynchronize();
```

```
    for(int i=0; i<n; ++i) {  
        std::cout << i << " " << array[i] << std::endl;  
    }
```

and deallocated with `cudaFree()`

```
    cudaFree(array);
```

```
    return 0;
```

```
}
```

Simple example

```
__global__ void kernel(int *array, int n) {  
    int ind = threadIdx.x + blockIdx.x * blockDim.x;  
    if (ind < n) {  
        array[ind] *= 2;  
    }  
}
```

```
int main() {  
    int n = 128;  
    int* array;  
    cudaMallocManaged(&array, n);  
  
    for (int i=0; i<n; ++i) {  
        array[i] = i*10 - 5;  
    }  
  
    kernel<<<1, n>>>(array, n);  
    cudaDeviceSynchronize();  
  
    for(int i=0; i<n; ++i) {  
        std::cout << i << " " << array[i] << std::endl;  
    }  
  
    cudaFree(array);  
  
    return 0;  
}
```

No explicit memory copies needed

Simple example

```
__global__ void kernel(int *array, int n) {
    int ind = threadIdx.x + blockIdx.x * blockDim.x;
    if (ind < n) {
        array[ind] *= 2;
    }
}

int main() {
    int n = 128;
    int* array;
    cudaMallocManaged(&array, n);

    for (int i=0; i<n; ++i) {
        array[i] = i*10 - 5;
    }


    kernel<<<1, n>>>(array, n);
    cudaDeviceSynchronize();

    for(int i=0; i<n; ++i) {
        std::cout << i << " " << array[i] << std::endl;
    }

    cudaFree(array);

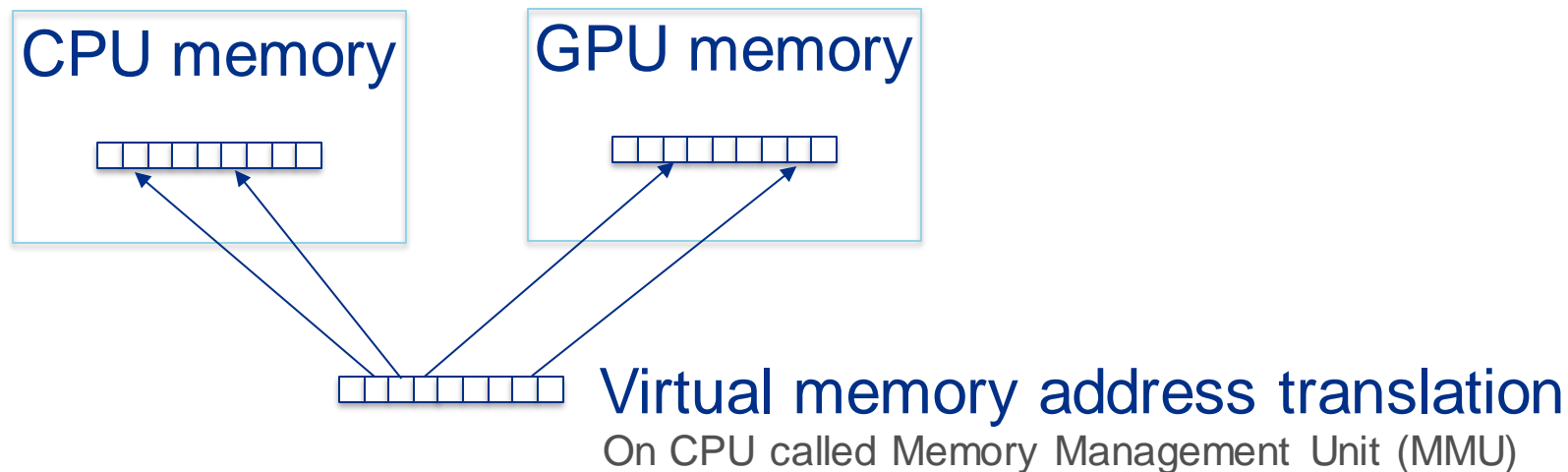
    return 0;
}
```

Still need to synchronize before
accessing the data on host!



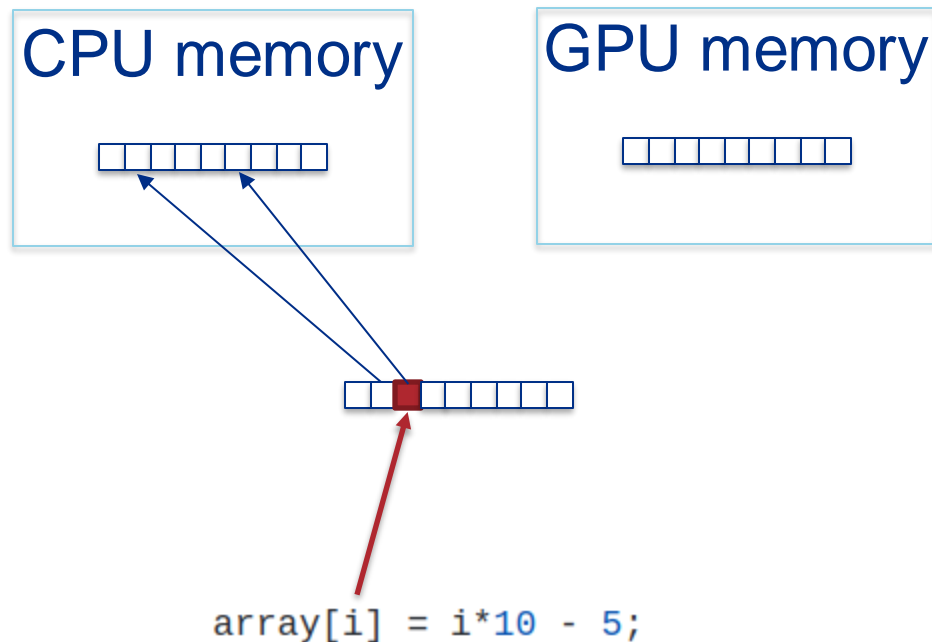
How does it work

- CUDA provides a unified virtual memory address space on both host and device



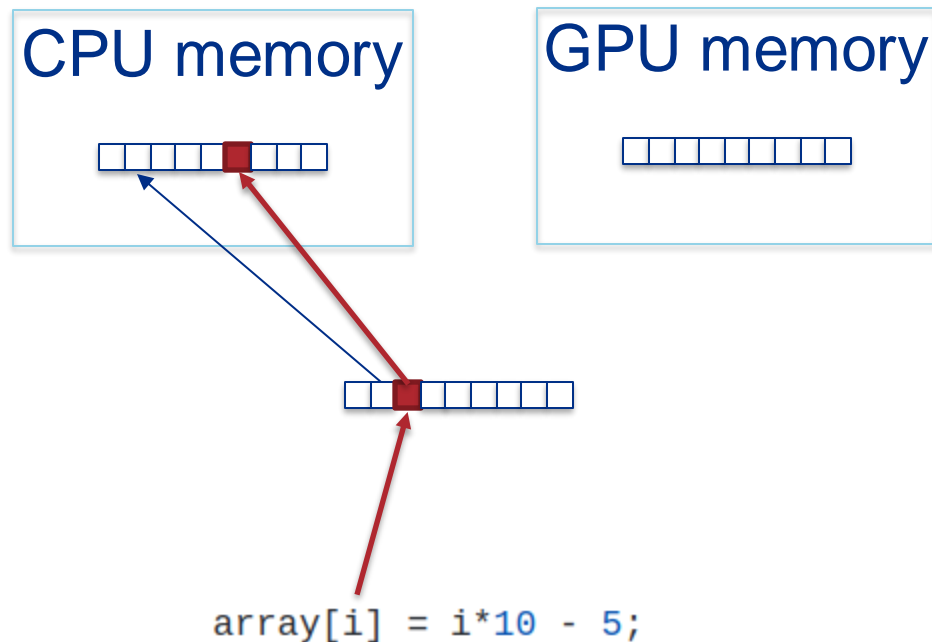
How does it work

- When a virtual memory address is accessed, the CPU (GPU) Memory Management Unit checks if it already knows the virtual memory address



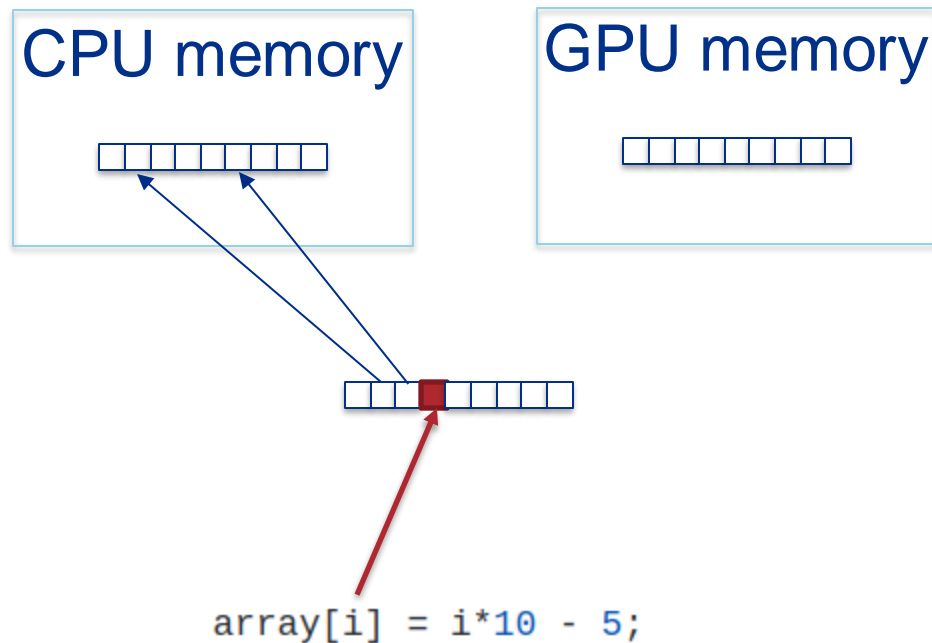
How does it work

- If MMU knows the physical address, it forwards the access to the corresponding physical memory address



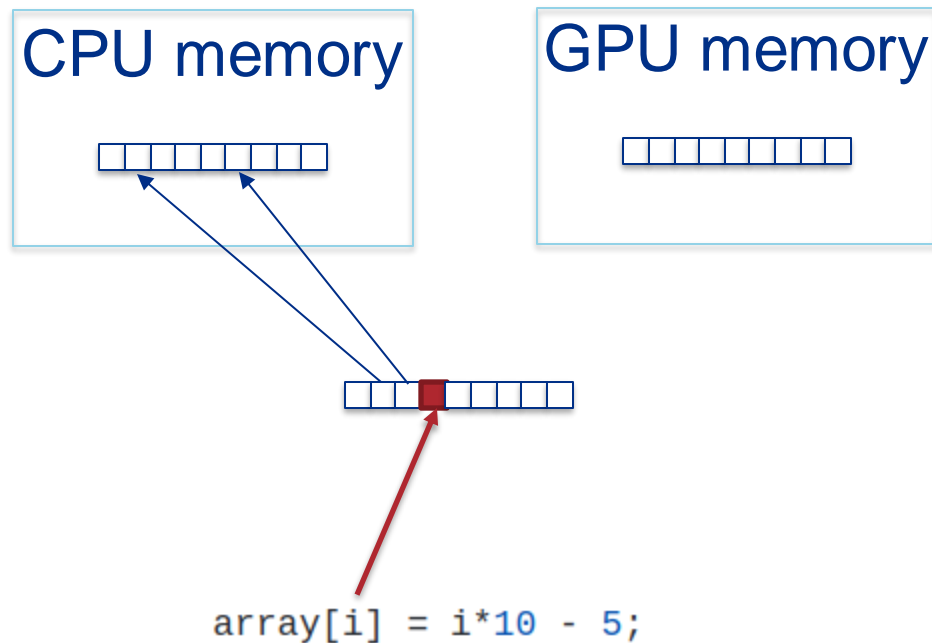
How does it work

- If the MMU does not know the physical address



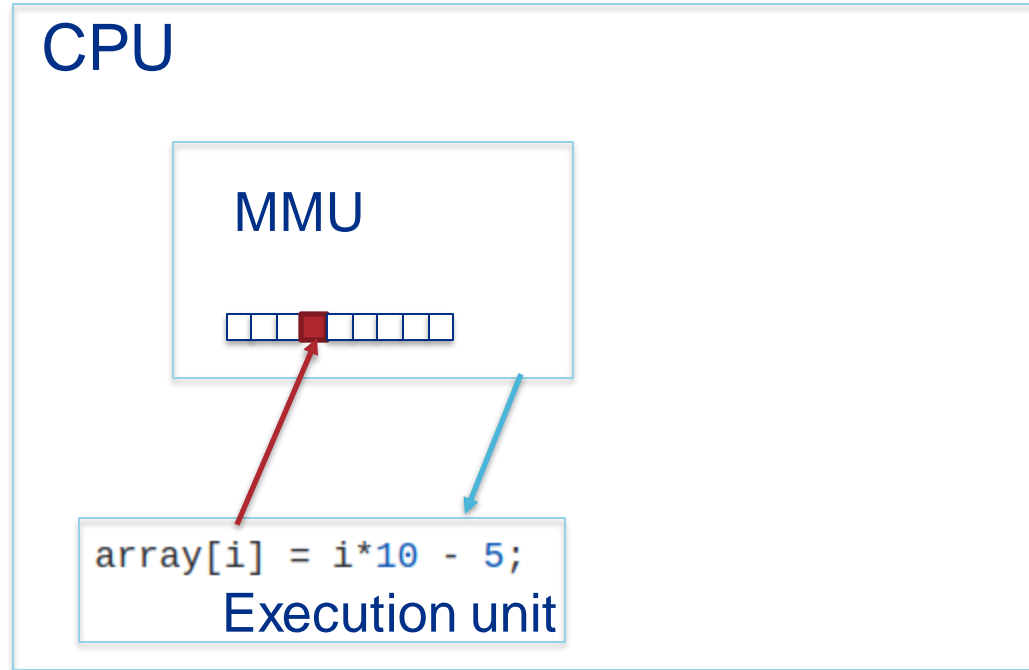
How does it work

- If the MMU does not know the physical address, it generates a page fault



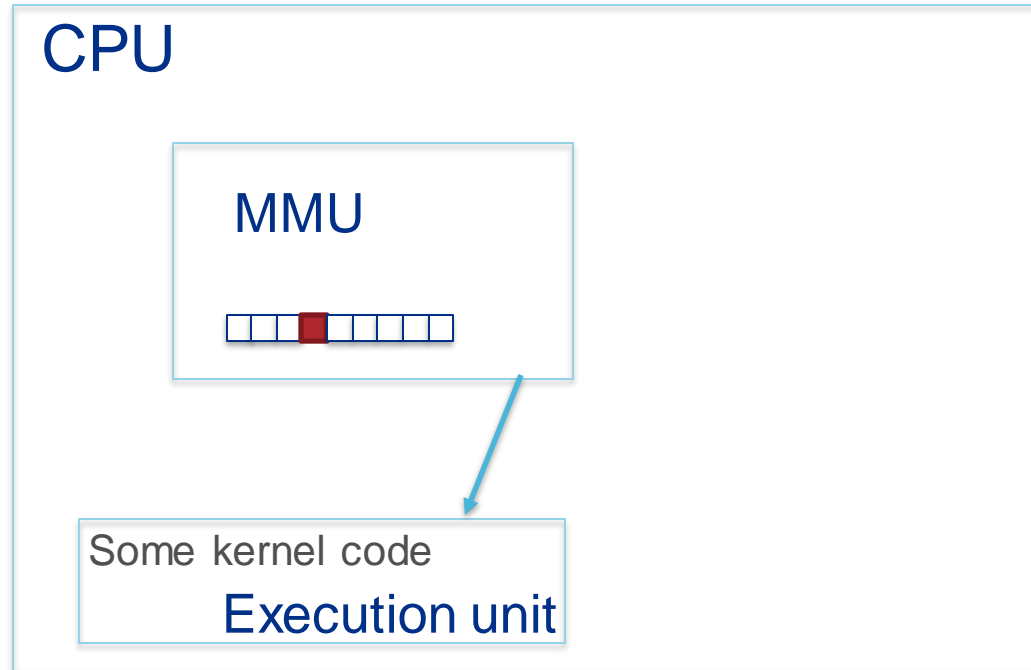
What is a page fault?

- CPU stops executing the user code



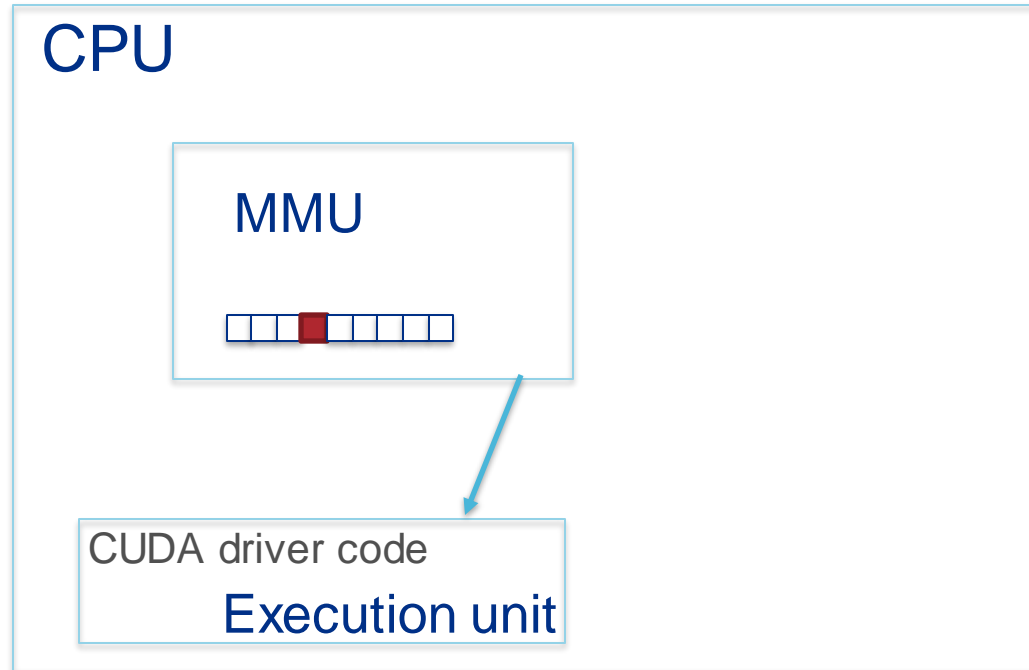
What is a page fault?

- CPU calls specific Operating System kernel code



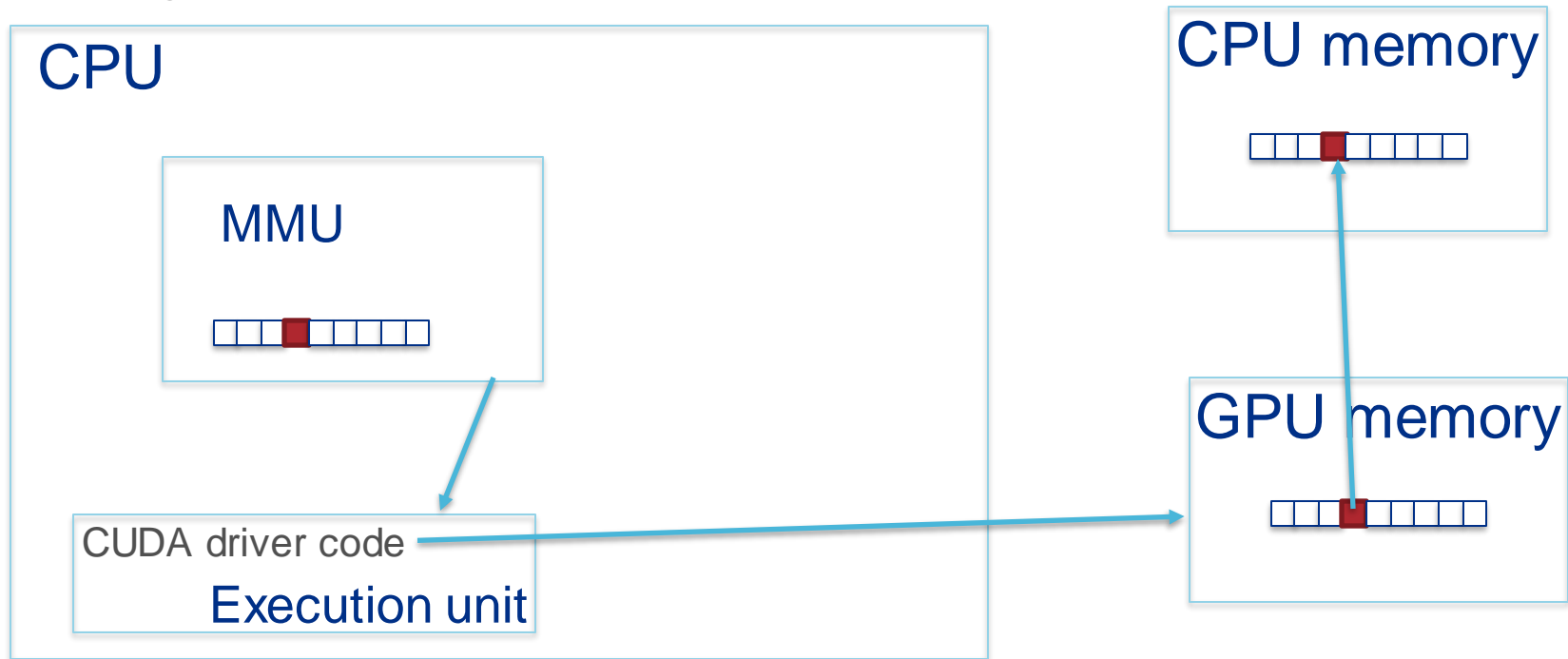
What is a page fault?

- Kernel calls CUDA driver if the driver knows the physical address



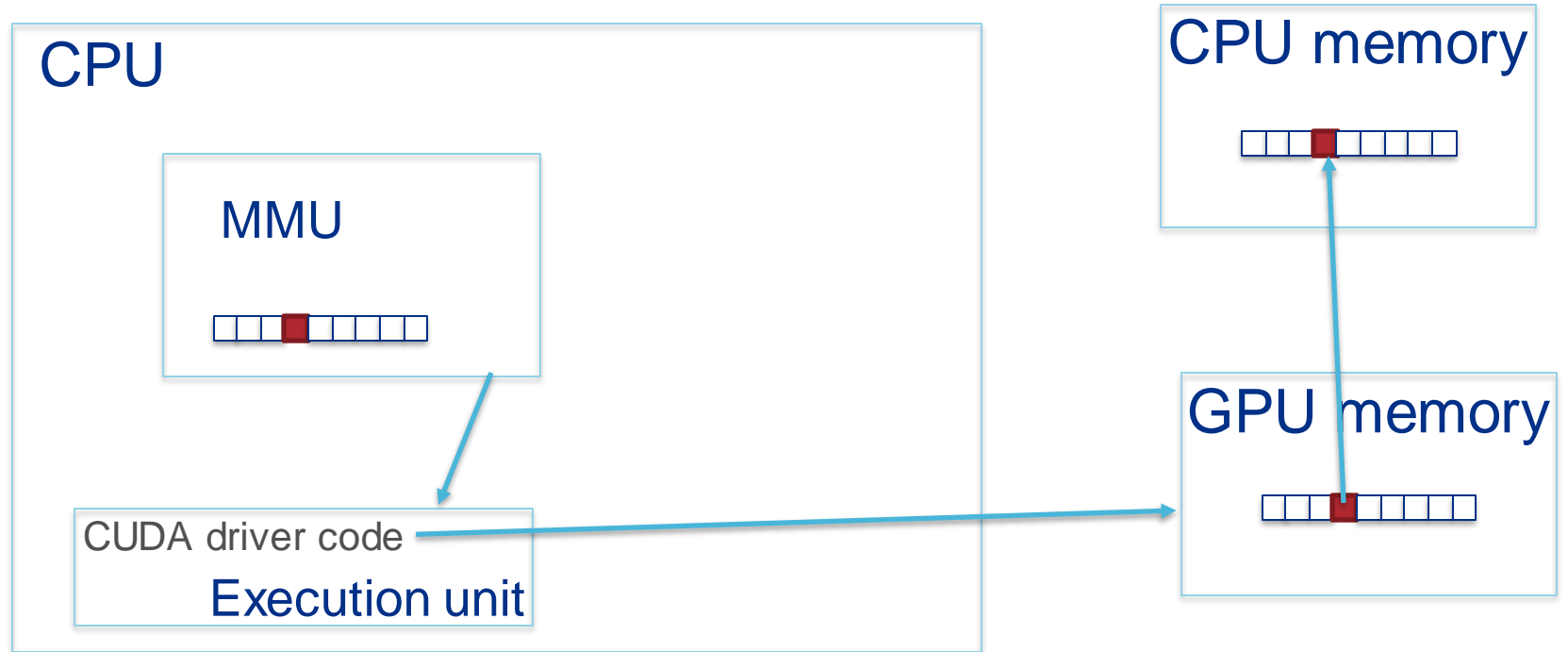
What is a page fault?

- If CUDA driver knows the corresponding memory block is on the GPU, it copies the memory page (typically 4 kB) to the CPU memory



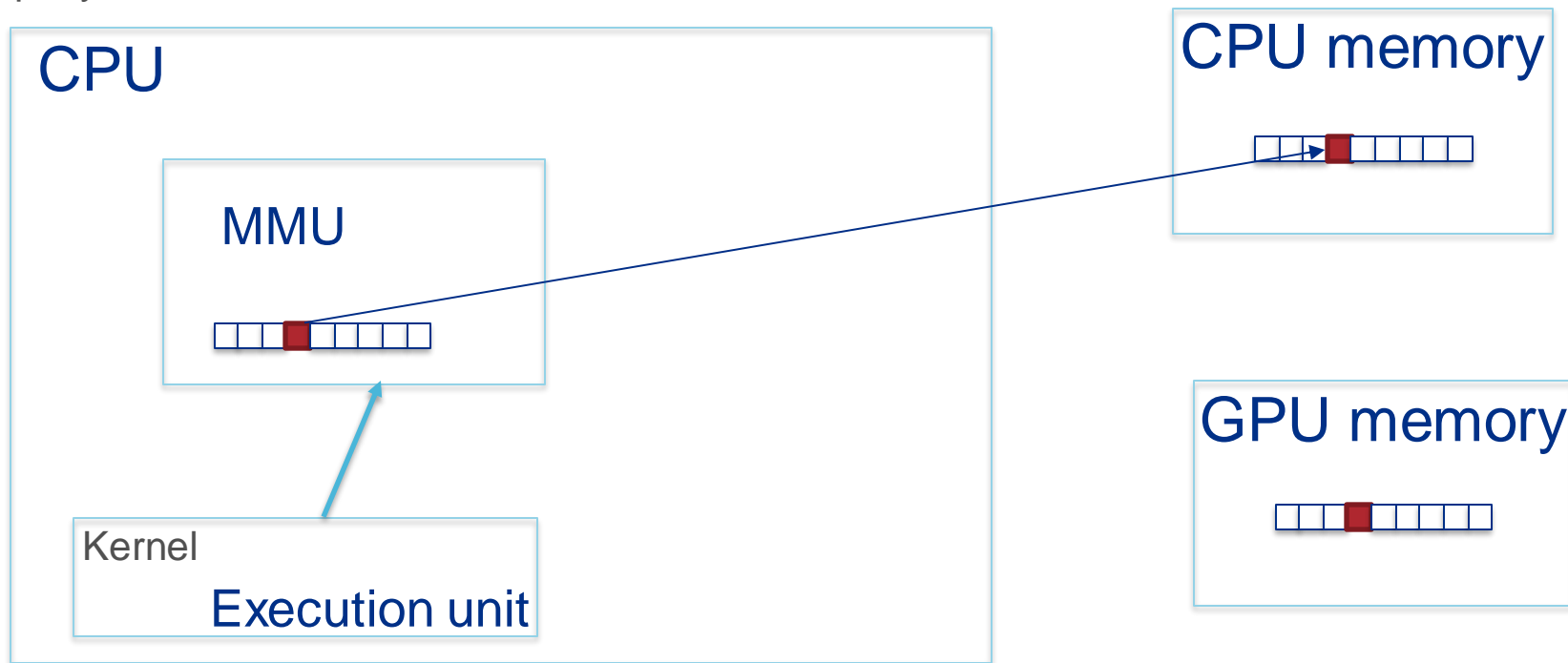
What is a page fault?

- Driver waits for the memory copy to complete



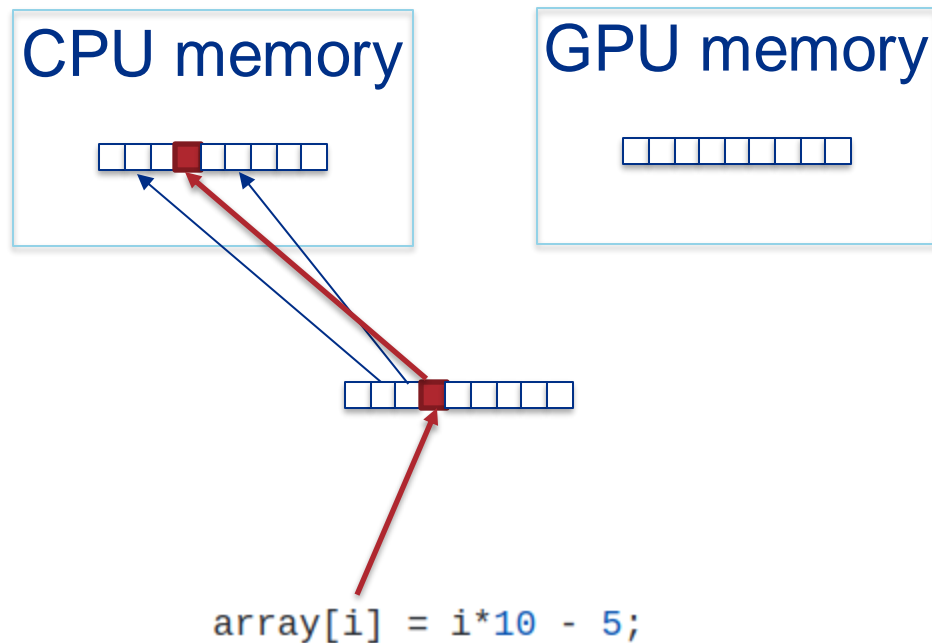
What is a page fault?

- Driver returns the new physical address to the kernel, who then sets up the MMU properly



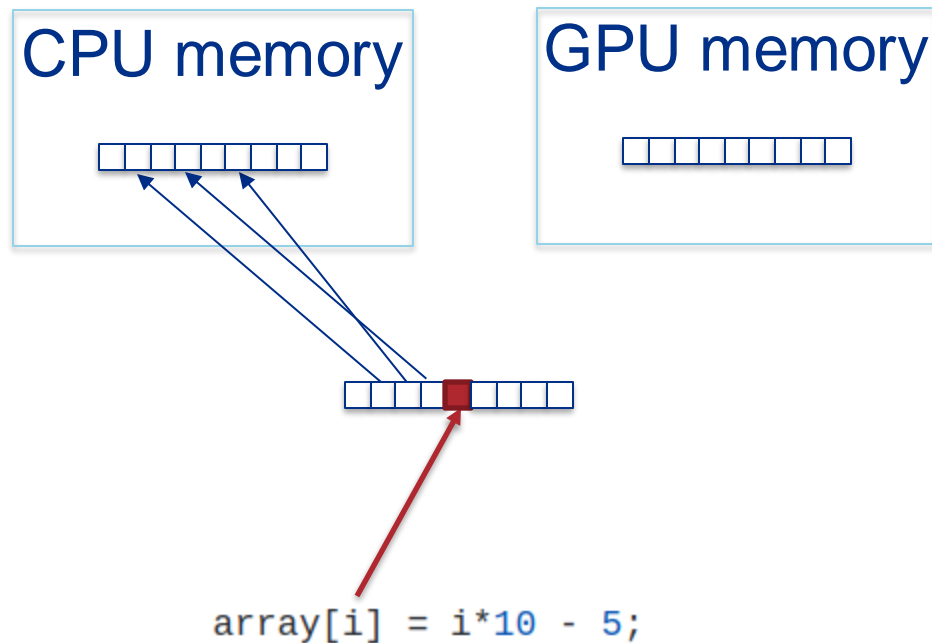
How does it work

- User code execution resumes



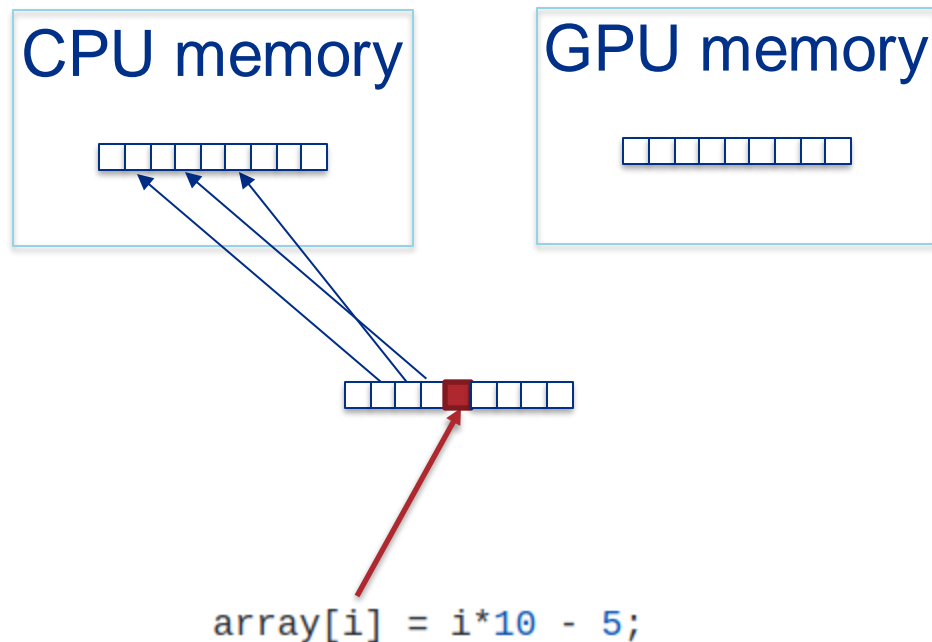
How does it work

- If CUDA driver is not able to translate the virtual address to physical address on the GPU



How does it work

- If CUDA driver is not able to translate the virtual address to physical address on the GPU, kernel asks from other drivers, and eventually aborts the program with segmentation fault



Optimizations

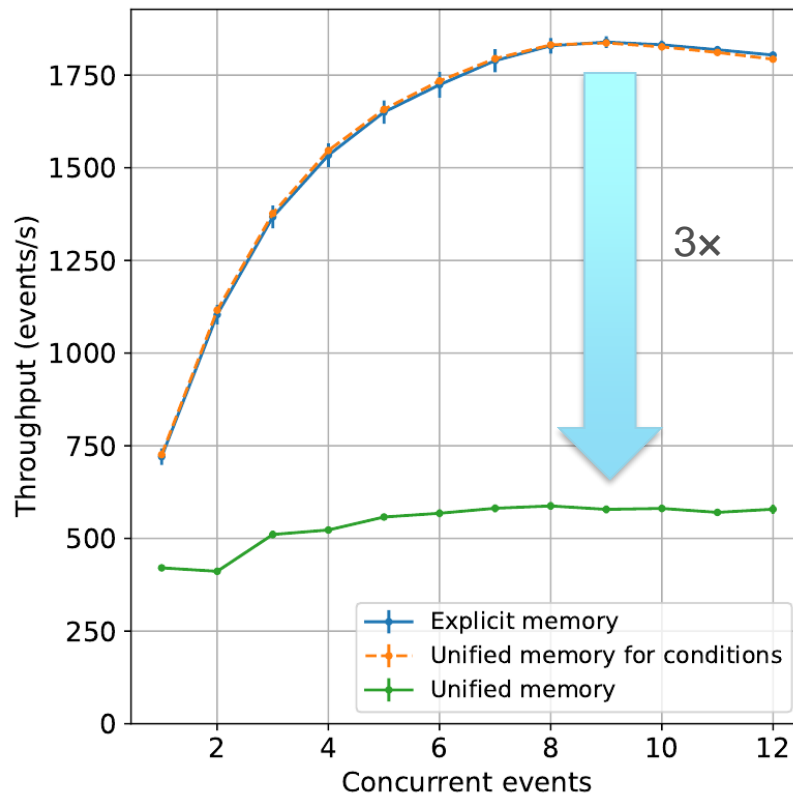
- As you can see, page faults are quite expensive
 - Cost becomes prohibitive if page faults occur often
- Often user code knows beforehand what memory it is going to access
 - User code can call `cudaMemPrefetchAsync()` to request the CUDA runtime+driver to prefetch a given memory block from host/device to device/host
 - Works best if user code can execute other code on host while the memory transfer is going on, i.e. ask to prefetch early
- User code can also give hints on memory block usage with `cudaMemAdvise()`
 - Like is it mostly read only, what is the preferred device, etc

Performance

- Some applications show good performance
 - Typically those have heavy computations

Performance

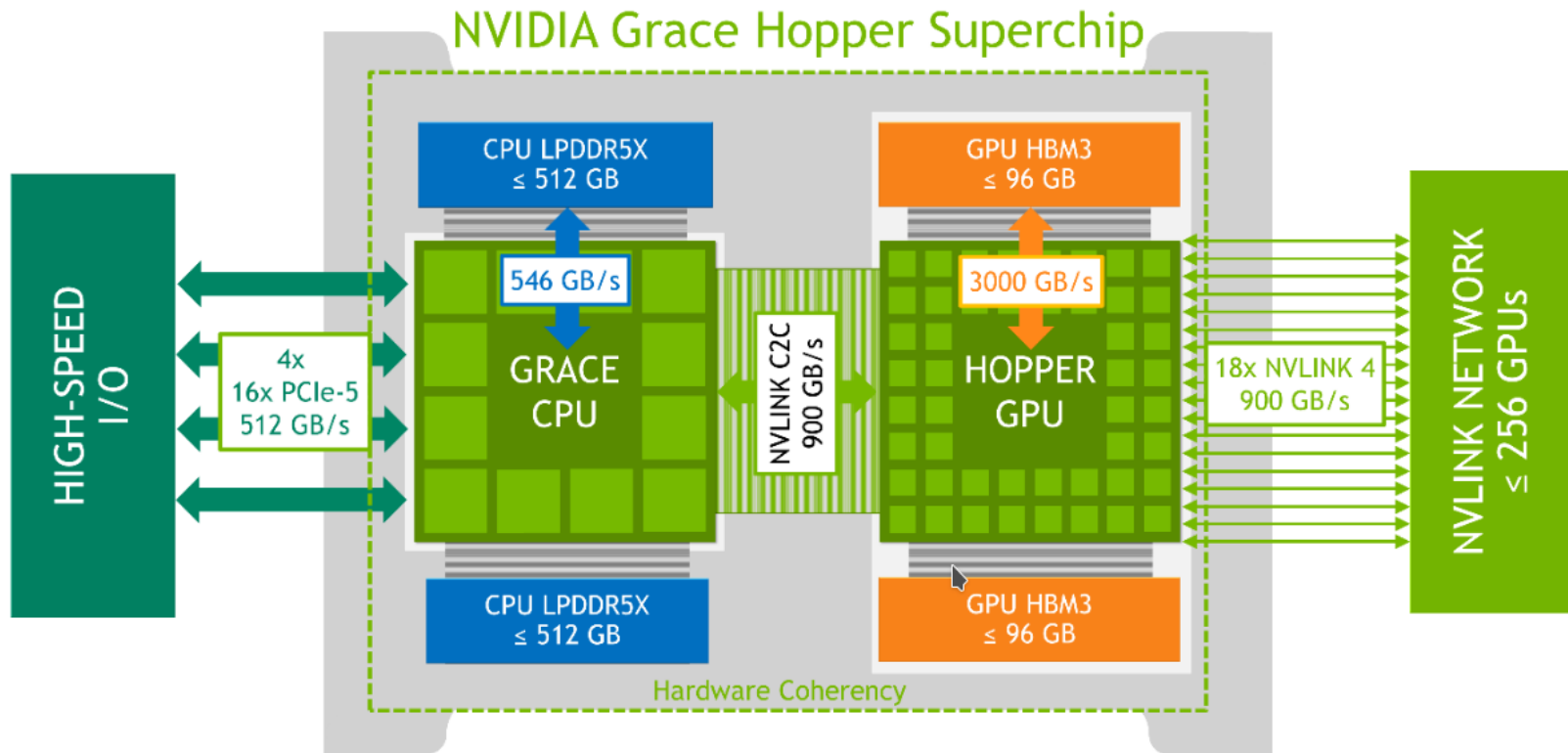
- Some other applications do not
 - Example from CMS heterogeneous pixel reconstruction test using CUDA Unified Memory
 - More details in [doi:10.1051/epjconf/202125103035](https://doi.org/10.1051/epjconf/202125103035)



When is managed memory useful?

- On a true unified memory system
 - CPU and GPU use the same physical memory
- With heavy computation kernels
 - Where memory transfer overheads don't matter in practice
- Complex data structures, with many pointers to other parts of the data structure
 - With explicit memory management, upon copying the data structure to device developer would have to update all the pointers to point to device memory
 - Becomes tedious quickly
- Allows overcommitting GPU memory in a transparent way
- (Large) data structure with sparse access pattern
- More difficult to change program from managed memory to explicit memory in case of performance problems than vice versa

When is managed memory useful?



Summary

- CUDA Unified / Managed memory provides an alternative memory management approach to the explicit management
 - In many cases the use is simpler
- The actual usefulness depends
 - On the application
 - Good for complex data structures, sparsely accessed large data structures, overcommitting GPU memory
 - Bad for applications where the cost of overheads is visible
 - On the hardware: **integrated** vs. **discrete GPU**
- NVIDIA's implementation on standard C++ parallelization relies on unified memory