The Battle Over Heterogeneous Computing

Oren Benita Ben Simhon
Compilers & Runtime Manager

By the end of this presentation, you will know…

+ The benefits of heterogeneous systems
+ Existing ways to utilize them
+ The exciting news brought by C++26

Oren Benita Ben Simhon

Director of Software Engineering

B.Sc in Computer Engineering at the Technion

M.B.A at the Technion

10 years experience at Qualcomm

8 years experience at Intel / Mobileye

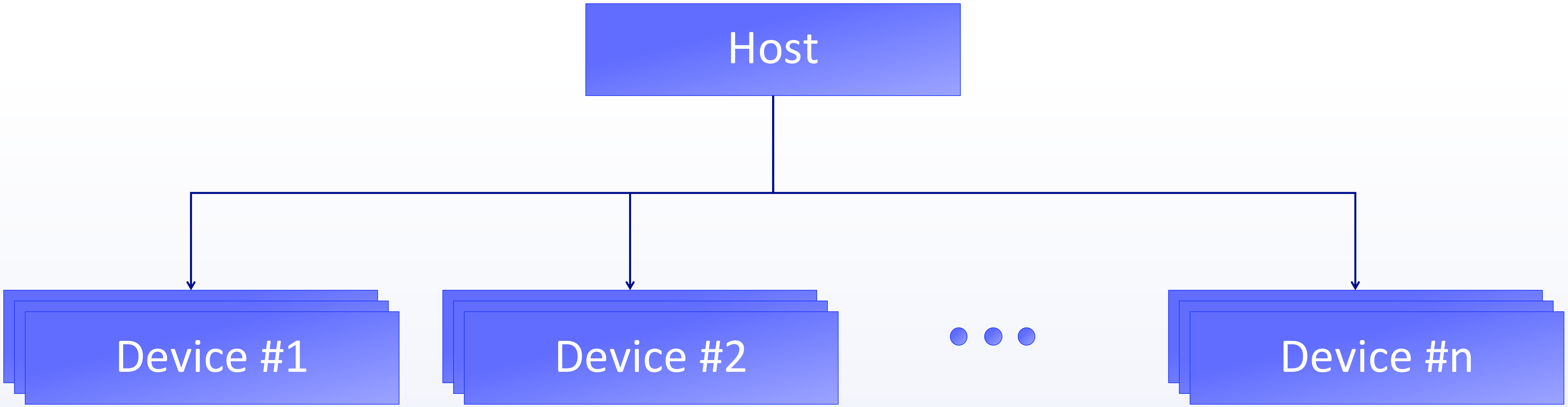
LLVM Based compilers

Theory of Compilation TA at the Technion

Israel LLVM Meetup

meetup

The Concept of Heterogeneous Systems



ACCELERATE

PARALLEL

VECTORIZE

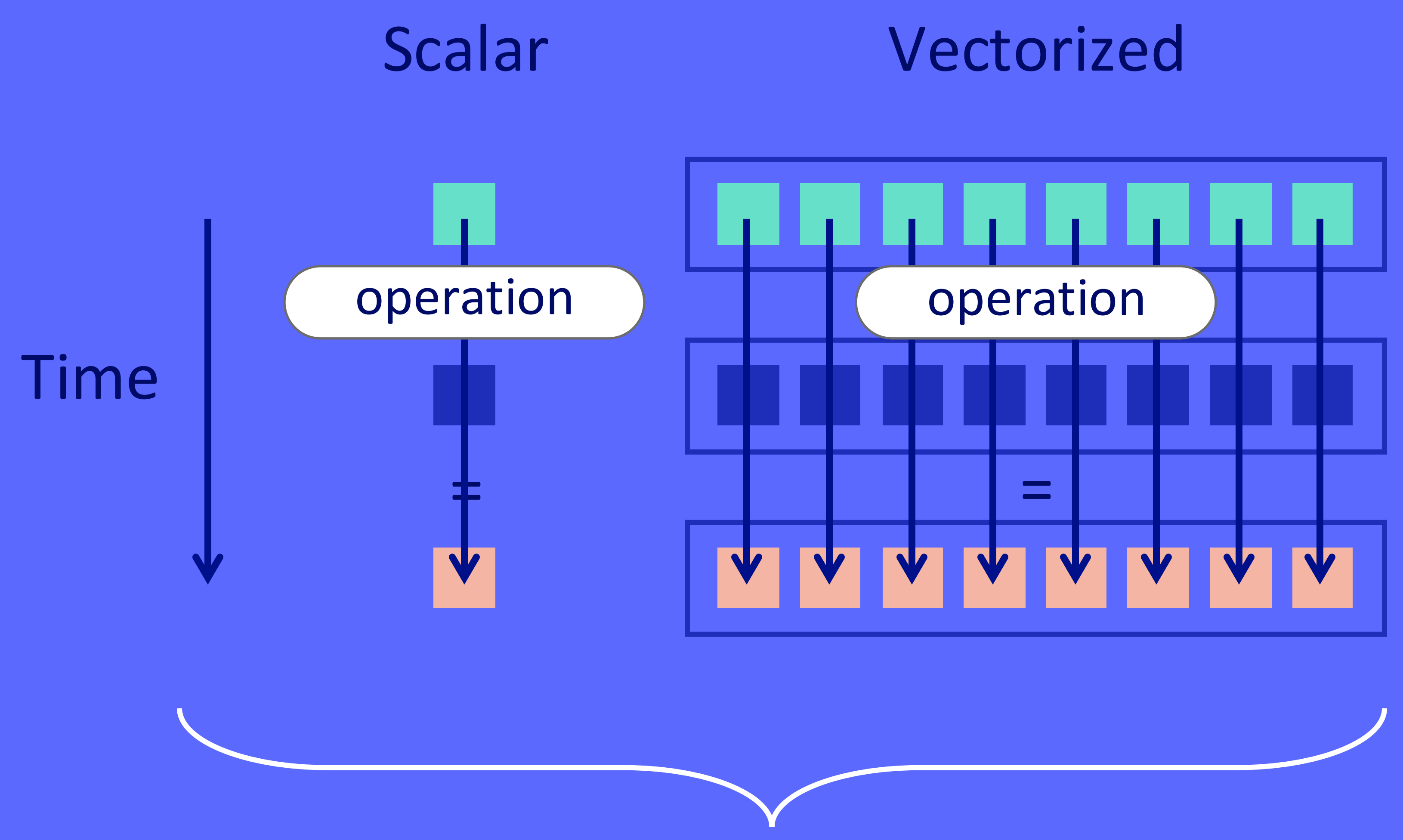
LOCALITY

PARALLEL

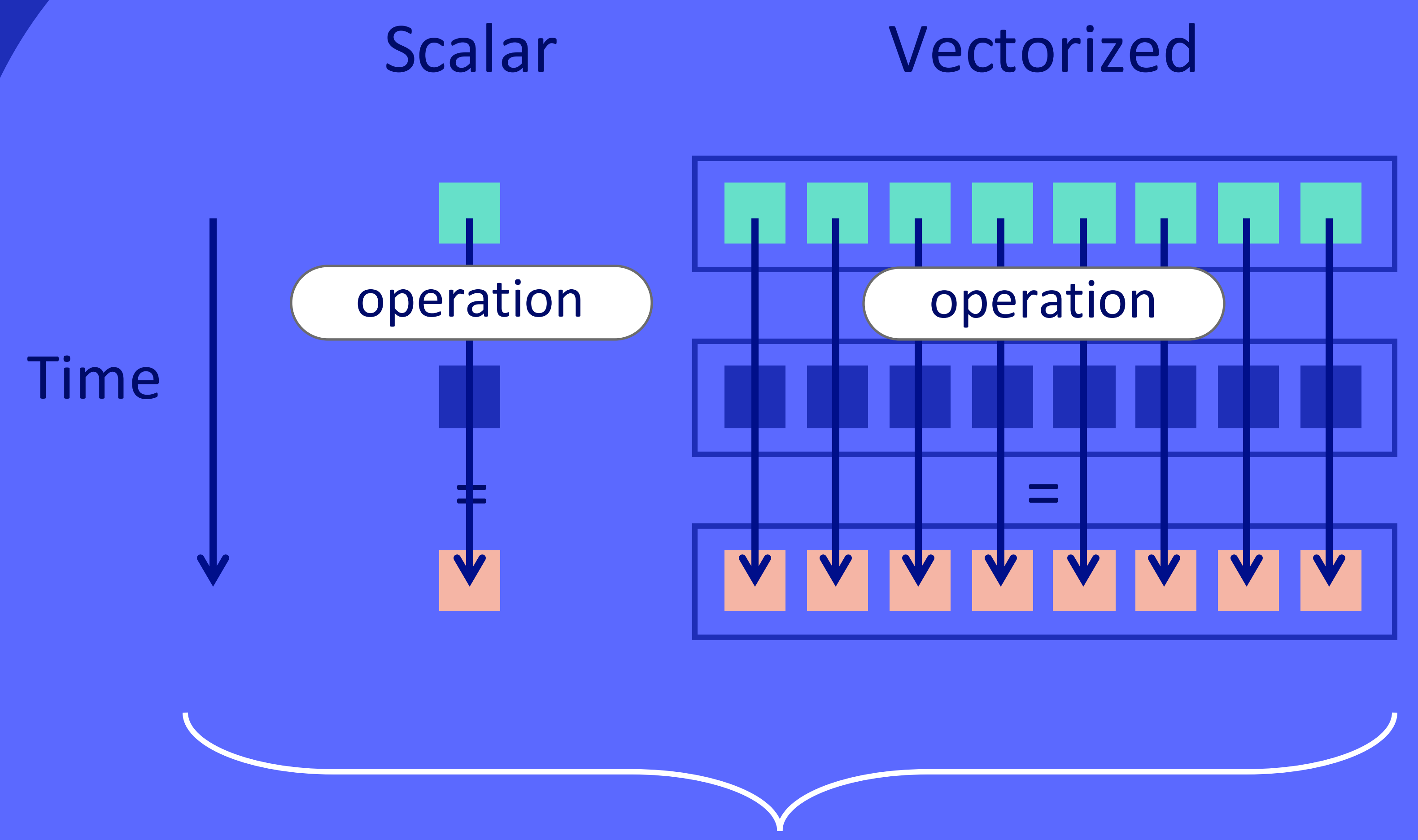
+ Runs code on multiple compute units
+ Executes single or multiple instruction streams
+ Offloading achieves data and task parallelism
+ Manual programming often needed
+ Enabled through asynchronous execution

VECTORIZE

```
for (int i = 0; i < 1024; i++) {
        a[i] = b[i] + c[i];
}
```

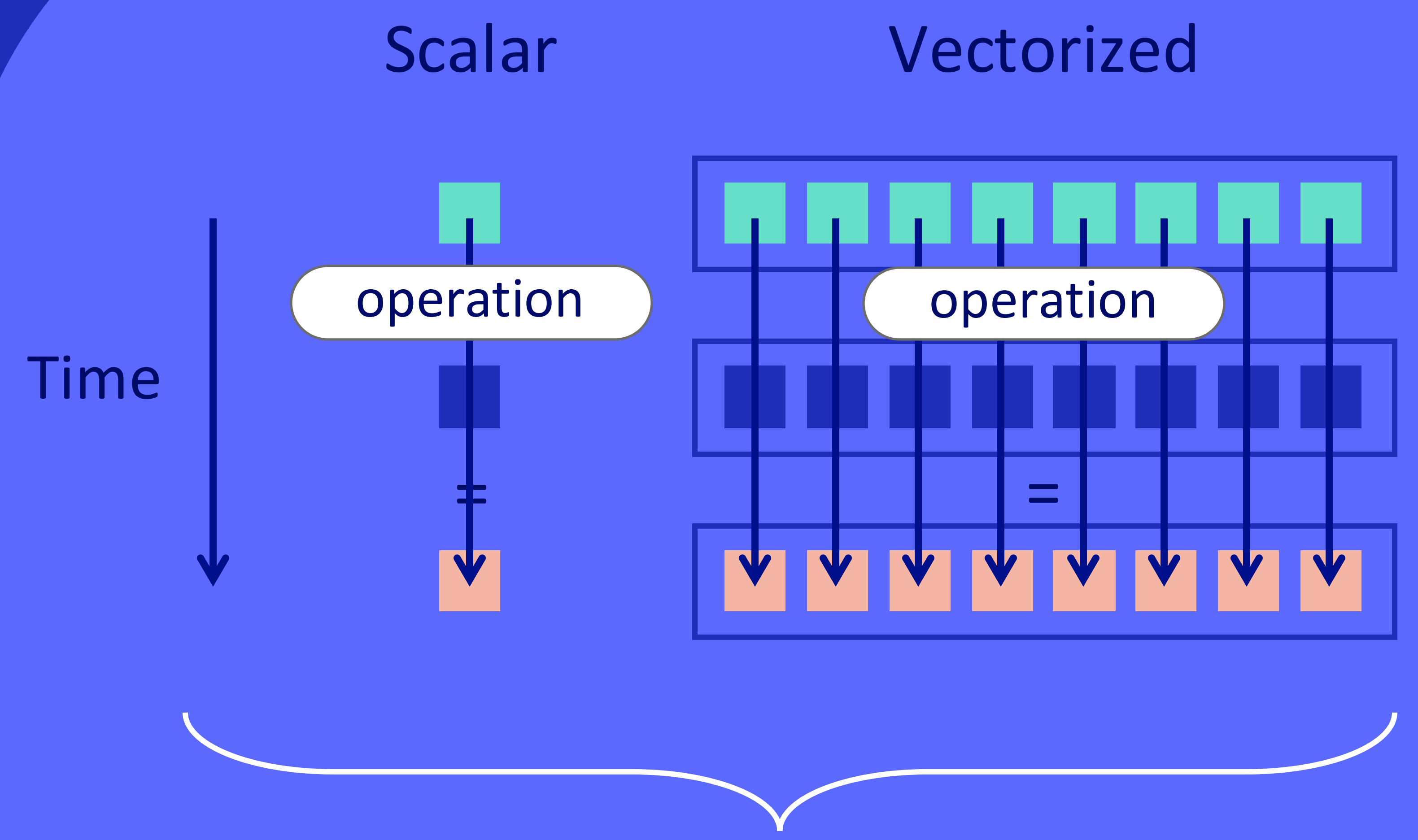



```
for (int i = 0; i < 1024; i+=8) {
        a[i...i+7] = b[i...i+7] + c[i...i+7];
}
```



VECTORIZE

- Runs on a single compute unit with multiple processing units
- Executes a single instruction stream
- A type of SIMD (Single Instruction Multiple Data)
- Could also be automatic by compiler
- Useful for loops



VECTORIZE

A WAY TO UTILIZE COMPUTE CAPABILITIES

LOCALITY

- To utilize compute power, memory bandwidth need to be sufficient
- Architecture can hide memory latency through execution strategies
- Caches may lower memory access cost
- Compiler can schedule transactions to reduce memory stalls
- Architecture can also work with local memory

Types Of Heterogeneous Devices

Graphic Processing Unit

Digital Signal Processor

Neural Processing Unit

Accelerated Processing Unit

How to program heterogeneous systems?

Examples of Heterogeneous Programming Languages

OpenMP

OpenCL

MPI

OpenACC
More Science, Less Programming

NVIDIA
CUDA

SYCL

Kernels

+ Functions that we will want to offload to the device
+ Usually contains a loop
+ It will be executed on the device
+ By offloading them we enable task parallelism
+ By running them on multiple compute units we enable data parallelism

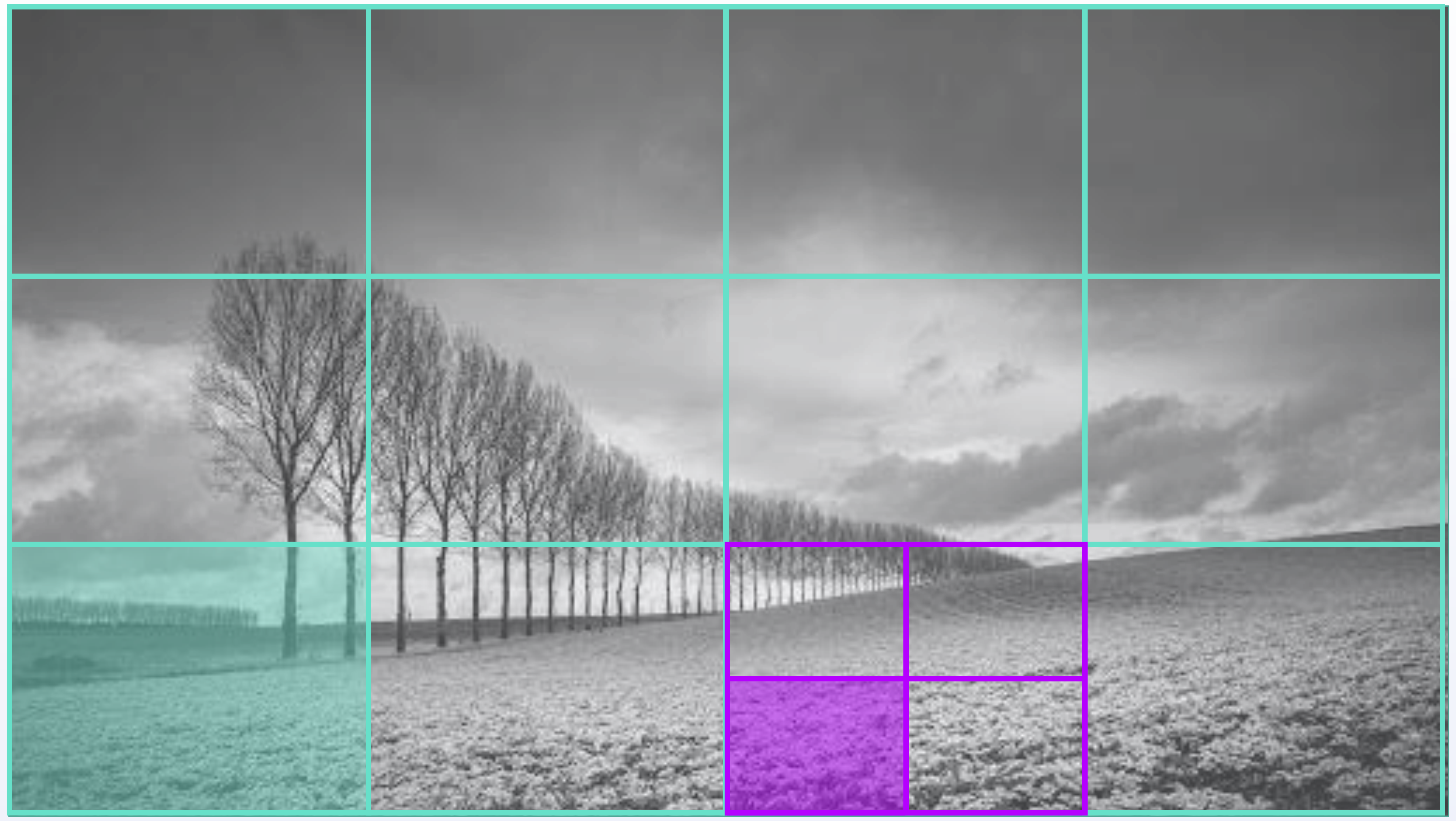
```
__kernel void multiply (. . .) {
        . . .
}
```

```
__global__ void multiply (. . .) {
        . . .
}
```

Data Parallelism Programming Model

The computational dimension is divided to work-groups / block

Each independent index or point is named a work-item / thread



Work Group (2,0)

Work Item
Global ID (5,4)
Local ID (1,0)

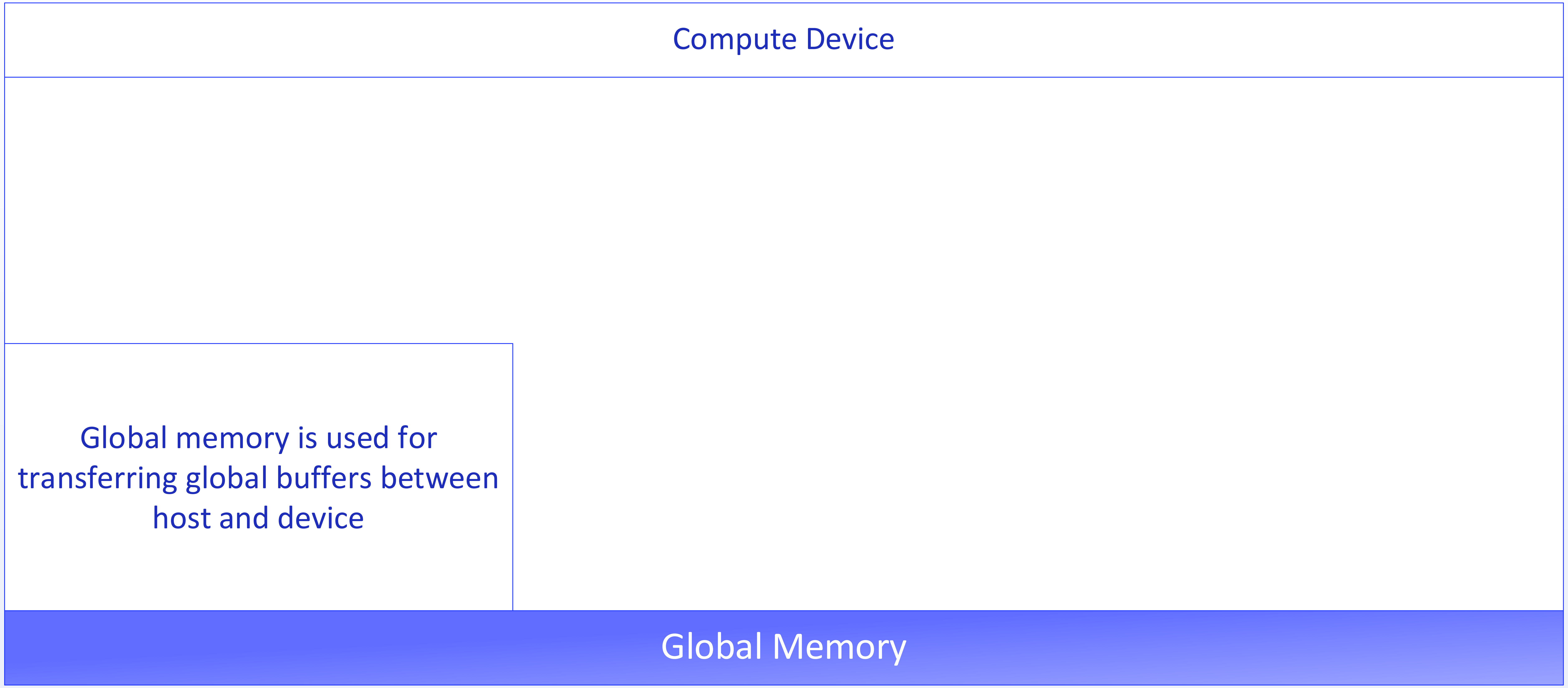
Each work item / thread identified by a global / thread ID

ND Range defines the total number of work items

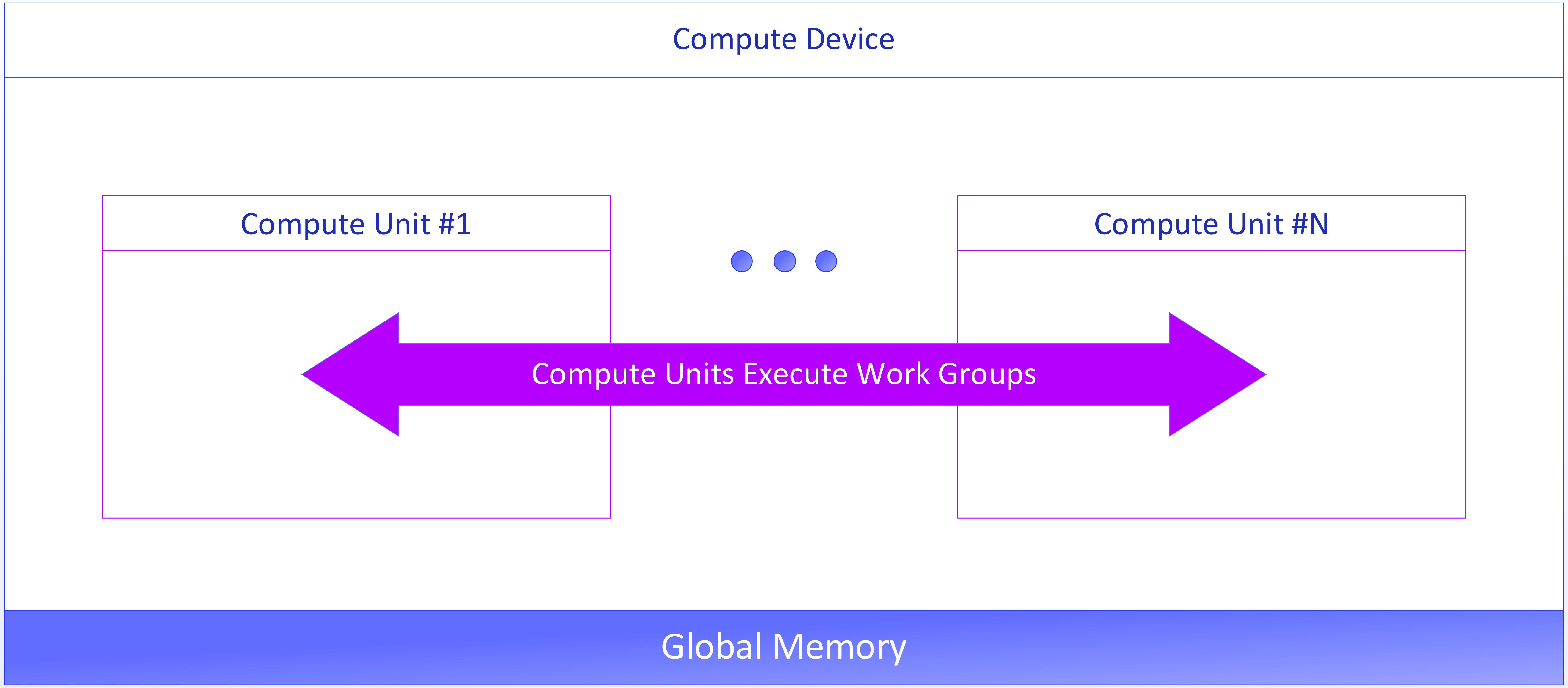
Comparison of terminology

CUDA	SYCL/OpenCL
Thread	Work Item
Block	Work Group
Grid	ND - Range

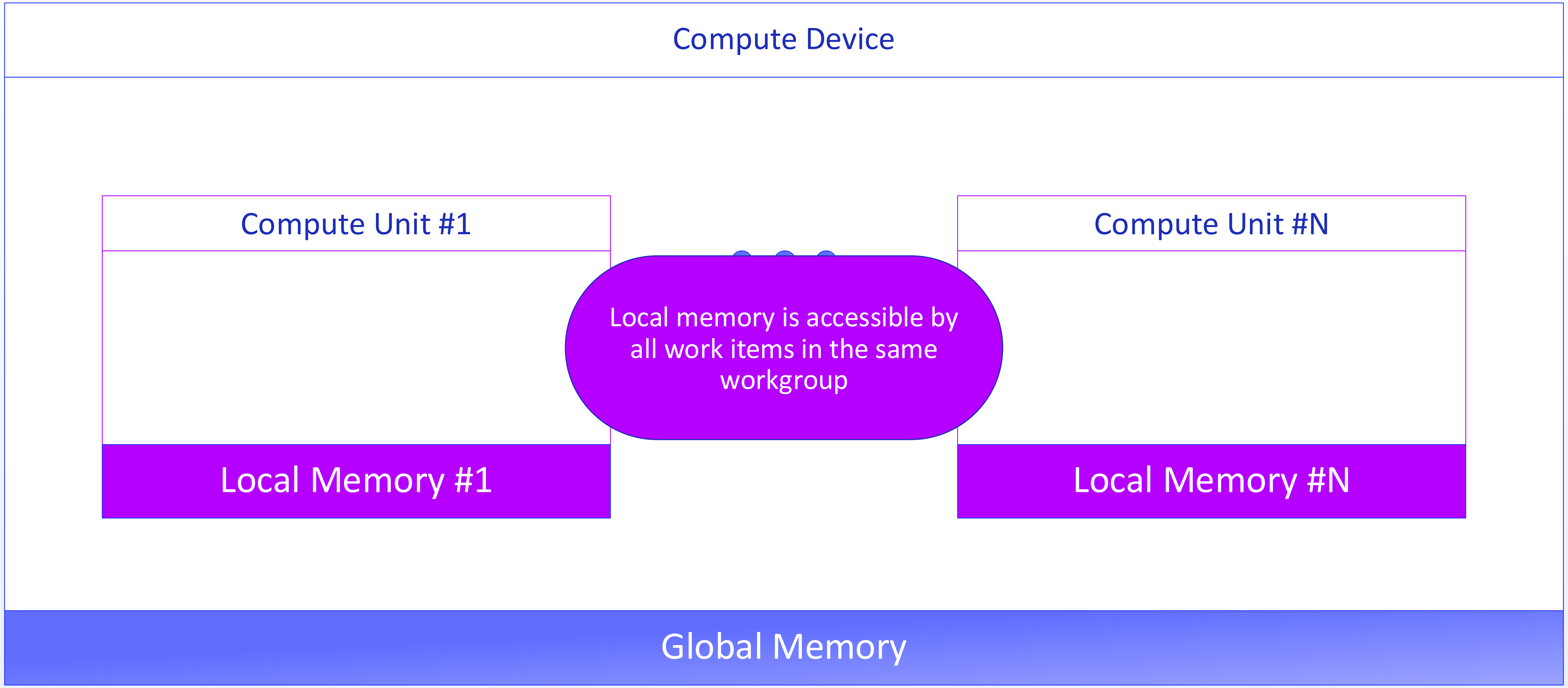
Device Memory Model



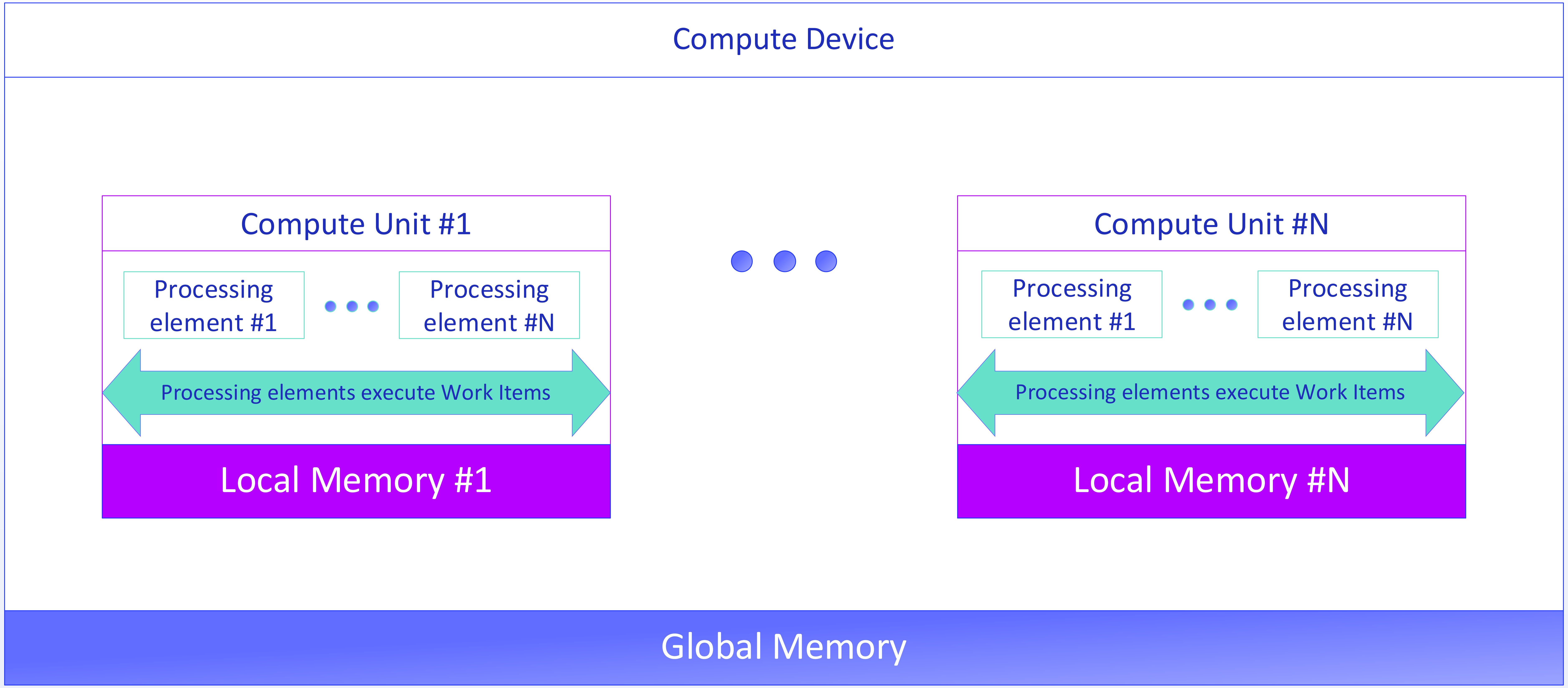
Device Memory Model



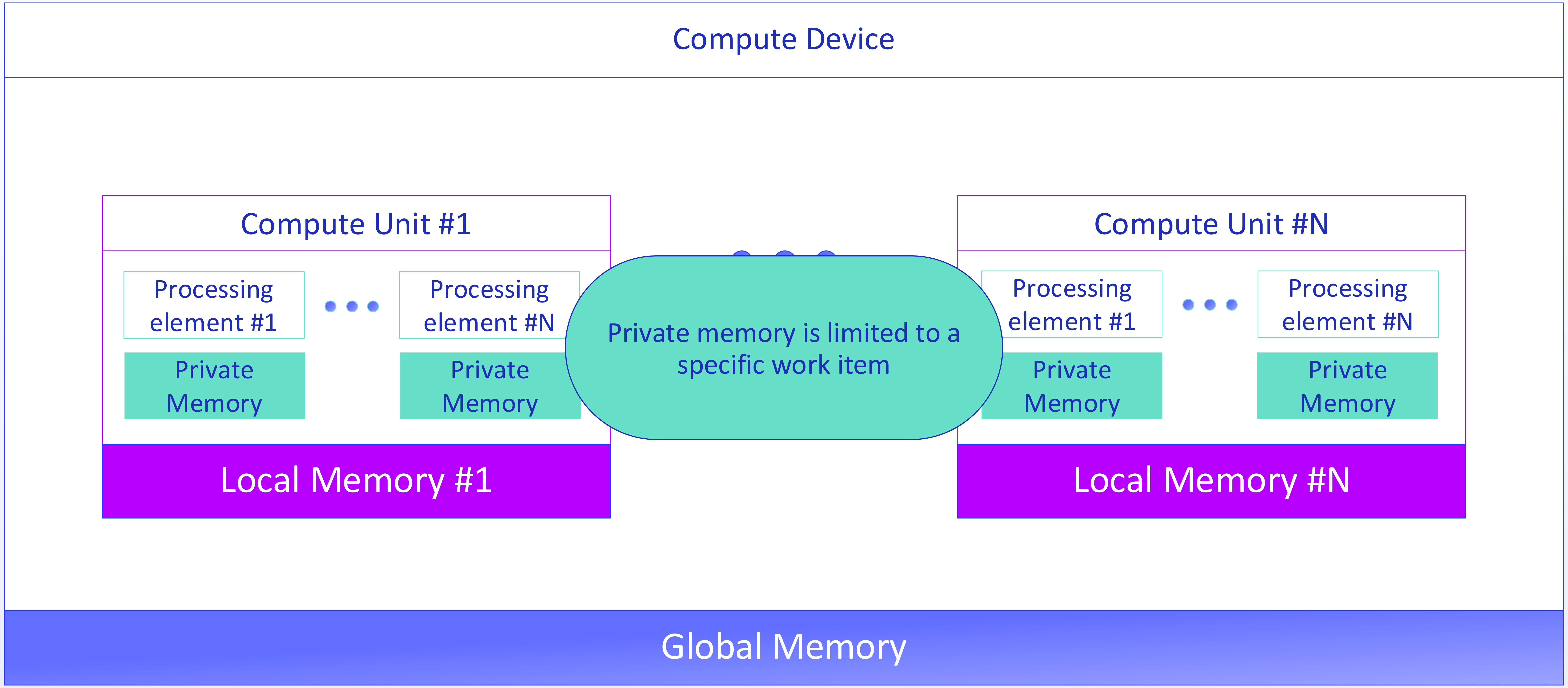
Device Memory Model



Device Memory Model



Device Memory Model



Comparison of terminology

CUDA	SYCL/OpenCL
Registers	Private
Shared	Local
Global / Constant	Global / Constant

CUDA

```
__global__ void Add(char4 *vec_array) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        vec_array[idx] += {5,5,5,5};
}


int main() {
        char4 *device_ptr;
        char4 *host_ptr; // Will be allocated with some values

        ...

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        cudaMalloc((void **)&device_ptr, num_elements * sizeof(char4))
        cudaMemcpyAsync(device_ptr, host_ptr, num_elements * sizeof(char4), cudaMemcpyHostToDevice);
        Add<<<gpu_workgroups, gpu_workitems_per_workgroup>>>(device_ptr);
        cudaMemcpyAsync(host_ptr, device_ptr, num_elements * sizeof(char4), cudaMemcpyDeviceToHost));
        cudaDeviceSynchronize();

}
```

CUDA

```
__global__ void Add(char4 *vec_array) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        vec_array[idx] += {5,5,5,5};
}

int main() {
        char4 *device_ptr;
        char4 *host_ptr; // Will be allocated with some values

        ...
```

CUDA

```
__global__ void Add(char4 *vec_array) {
                int idx = blockIdx.x * blockDim.x + threadIdx.x;
                vec_array[idx] += {5,5,5,5};
}


int main() {
                char4 *device_ptr;
                char4 *host_ptr; // Will be allocated with some values

                ...

                int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
                cudaMalloc((void **)&device_ptr, num_elements * sizeof(char4))
                cudaMemcpyAsync(device_ptr, host_ptr, num_elements * sizeof(char4), cudaMemcpyHostToDevice);
                Add<<<gpu_workgroups, gpu_workitems_per_workgroup>>>(device_ptr);
                cudaMemcpyAsync(host_ptr, device_ptr, num_elements * sizeof(char4), cudaMemcpyDeviceToHost));
                cudaDeviceSynchronize();

}
```

CUDA

```
__global__ void Add(char4 *vec_array) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        vec_array[idx] += {5,5,5,5};
}


int main() {
        char4 *device_ptr;
        char4 *host_ptr; // Will be allocated with some values

        ...

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        cudaMalloc((void **)&device_ptr, num_elements * sizeof(char4))
        cudaMemcpyAsync(device_ptr, host_ptr, num_elements * sizeof(char4), cudaMemcpyHostToDevice);
        Add<<<gpu_workgroups, gpu_workitems_per_workgroup>>>(device_ptr);
        cudaMemcpyAsync(host_ptr, device_ptr, num_elements * sizeof(char4), cudaMemcpyDeviceToHost));
        cudaDeviceSynchronize();

}
```

CUDA

```
__global__ void Add(char4 *vec_array) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        vec_array[idx] += {5,5,5,5};
}


int main() {
        char4 *device_ptr;
        char4 *host_ptr; // Will be allocated with some values

        ...

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        cudaMalloc((void **)&device_ptr, num_elements * sizeof(char4))
        cudaMemcpyAsync(device_ptr, host_ptr, num_elements * sizeof(char4), cudaMemcpyHostToDevice);
        Add<<<gpu_workgroups, gpu_workitems_per_workgroup>>>(device_ptr);
        cudaMemcpyAsync(host_ptr, device_ptr, num_elements * sizeof(char4), cudaMemcpyDeviceToHost));
        cudaDeviceSynchronize();

}
```

Open CL

```
int main() {

        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values

        const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
        const Program program(context, {myDevice}, {binary});
        status = program.build();
        Kernel kernel(program, "add");

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
        queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

        kernel.setArg(0, device_ptr);

        queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),             NDRange(gpu_workitems_per_workgroup));

        Event e;
        queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
        e.wait();

}
```

```
__kernel void add(global char* array) {

        int idx = get_global_id(0);

        array[idx] += 5;

}
```

Open CL

```
__kernel void add(global char* array) {
        int idx = get_global_id(0);
        array[idx] += 5;
}
```

```
int main() {
        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values
```

Open CL

```
int main() {
        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values

        const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
        const Program program(context, {myDevice}, {binary});
        status = program.build();
        Kernel kernel(program, "add");

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
        queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

        kernel.setArg(0, device_ptr);

        queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),            NDRange(gpu_workitems_per_workgroup));

        Event e;
        queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
        e.wait();
}
```

```
__kernel void add(global char* array) {
        int idx = get_global_id(0);
        array[idx] += 5;
}
```

Open CL

```
int main() {

    const Platform platform = Platform::getDefault();
    const Device myDevice = getDevice(platform, DeviceName);
    const Context context(myDevice);
    const CommandQueue queue(context, mycDevice);
    char* host_ptr; // Will be allocated with some values

    const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
    const Program program(context, {myDevice}, {binary});
    status = program.build();
    Kernel kernel(program, "add");

    int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
    Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
    queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

    kernel.setArg(0, device_ptr);

    queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),        NDRange(gpu_workitems_per_workgroup));

    Event e;
    queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
    e.wait();
}
```

```
__kernel void add(global char* array) {
        int idx = get_global_id(0);
        array[idx] += 5;
}
```

Open CL

```
int main() {

        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values

        const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
        const Program program(context, {myDevice}, {binary});
        status = program.build();
        Kernel kernel(program, "add");

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
        queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

        kernel.setArg(0, device_ptr);

        queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),          NDRange(gpu_workitems_per_workgroup));

        Event e;
        queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
        e.wait();
}
```

```
__kernel void add(global char* array) {

                int idx = get_global_id(0);

                array[idx] += 5;

}
```

Open CL

```
int main() {

        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values

        const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
        const Program program(context, {myDevice}, {binary});
        status = program.build();
        Kernel kernel(program, "add");

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
        queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

        kernel.setArg(0, device_ptr);

        queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),          NDRange(gpu_workitems_per_workgroup));

        Event e;
        queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
        e.wait();
}
```

```
__kernel void add(global char* array) {
        int idx = get_global_id(0);
        array[idx] += 5;
}
```

Open CL

```
int main() {

        const Platform platform = Platform::getDefault();
        const Device myDevice = getDevice(platform, DeviceName);
        const Context context(myDevice);
        const CommandQueue queue(context, mycDevice);
        char* host_ptr; // Will be allocated with some values

        const std::vector<unsigned char> binary = readProgramBinary(MY_PROG_FILE_NAME);
        const Program program(context, {myDevice}, {binary});
        status = program.build();
        Kernel kernel(program, "add");

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        Buffer device_ptr(context, CL_MEM_READ_WRITE, num_elements);
        queue.enqueueWriteBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr);

        kernel.setArg(0, device_ptr);

        queue.enqueueNDRangeKernel(kernel, NDRange(), NDRange(num_elements),          NDRange(gpu_workitems_per_workgroup));

        Event e;
        queue.enqueueReadBuffer(device_ptr, CL_FALSE, 0, num_elements, host_ptr, NULL, &e);
        e.wait();
}
```

```
__kernel void add(global char* array) {

        int idx = get_global_id(0);

        array[idx] += 5;

}
```

SYCL

```
int main()
        queue q(default_selector{});
        char* host_ptr; // Will be allocated with some values
        ...
        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        char *device_ptr = malloc_device<char>(num_elements, q);
        q.memcpy(device_ptr, host_ptr, num_elements);

        q.submit([&](auto &cgh) {
                cgh.parallel_for((num_elements, gpu_workitems_per_workgroup) ,
                                [=](auto i) {device_ptr[i] += 5;});
        });

        q.memcpy(host_ptr, device_ptr, num_elements).wait();
}
```

SYCL

```
int main()
        queue q(default_selector{});
        char* host_ptr; // Will be allocated with some values
        ...
        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        char *device_ptr = malloc_device<char>(num_elements, q);
        q.memcpy(device_ptr, host_ptr, num_elements);

        q.submit([&](auto &cgh) {
                cgh.parallel_for((num_elements, gpu_workitems_per_workgroup) ,
                                [=](auto i) {device_ptr[i] += 5;});
        });

        q.memcpy(host_ptr, device_ptr, num_elements).wait();

}
```

SYCL

```
int main()
        queue q(default_selector{});
        char* host_ptr; // Will be allocated with some values
        ...
        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        char *device_ptr = malloc_device<char>(num_elements, q);
        q.memcpy(device_ptr, host_ptr, num_elements);

        q.submit([&](auto &cgh) {
                cgh.parallel_for((num_elements, gpu_workitems_per_workgroup) ,
                                [=](auto i) {device_ptr[i] += 5;});
        });

        q.memcpy(host_ptr, device_ptr, num_elements).wait();

}
```

SYCL

```
int main()
        queue q(default_selector{});
        char* host_ptr; // Will be allocated with some values
        ...
        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        char *device_ptr = malloc_device<char>(num_elements, q);
        q.memcpy(device_ptr, host_ptr, num_elements);

        q.submit([&](auto &cgh) {
                cgh.parallel_for((num_elements, gpu_workitems_per_workgroup) ,
                                [=](auto i) {device_ptr[i] += 5;});
        });

        q.memcpy(host_ptr, device_ptr, num_elements).wait();
}
```

Is it really C++?

- No Exceptions
- No standard C++ libraries
- No new/delete operations in SYCL
- Function pointers cannot be passed between host and device
- Classes with virtual functions cannot be passed to a kernel
- Kernel functions cannot be recursive

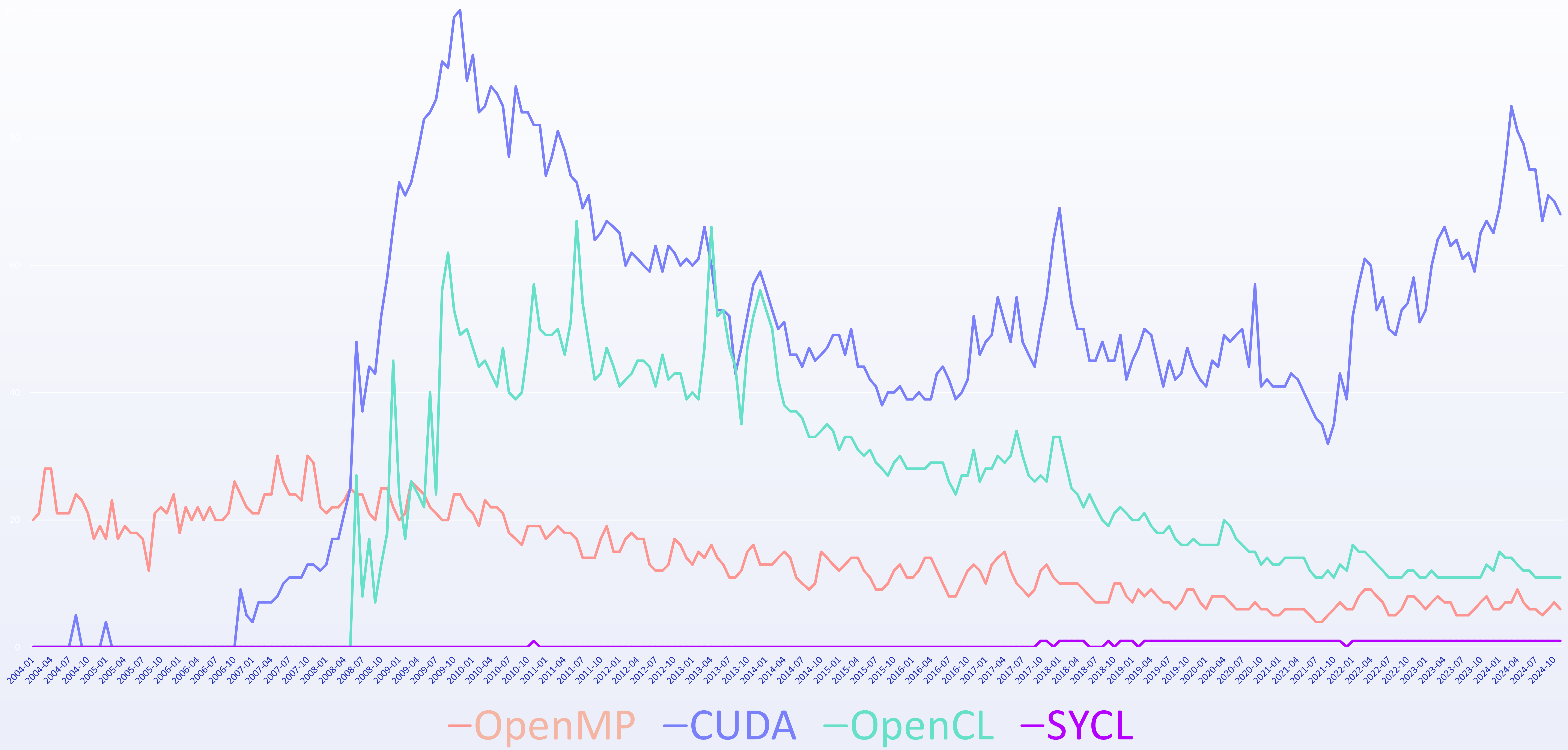
Open MP

```
int main() {
    char *array;
    ...

    int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
    #pragma omp target
    {
        #pragma omp parallel for simd num_threads(num_elements)
        for (int i=0; i < num_elements; i++)
            array[i] += 5;
    }
}
```

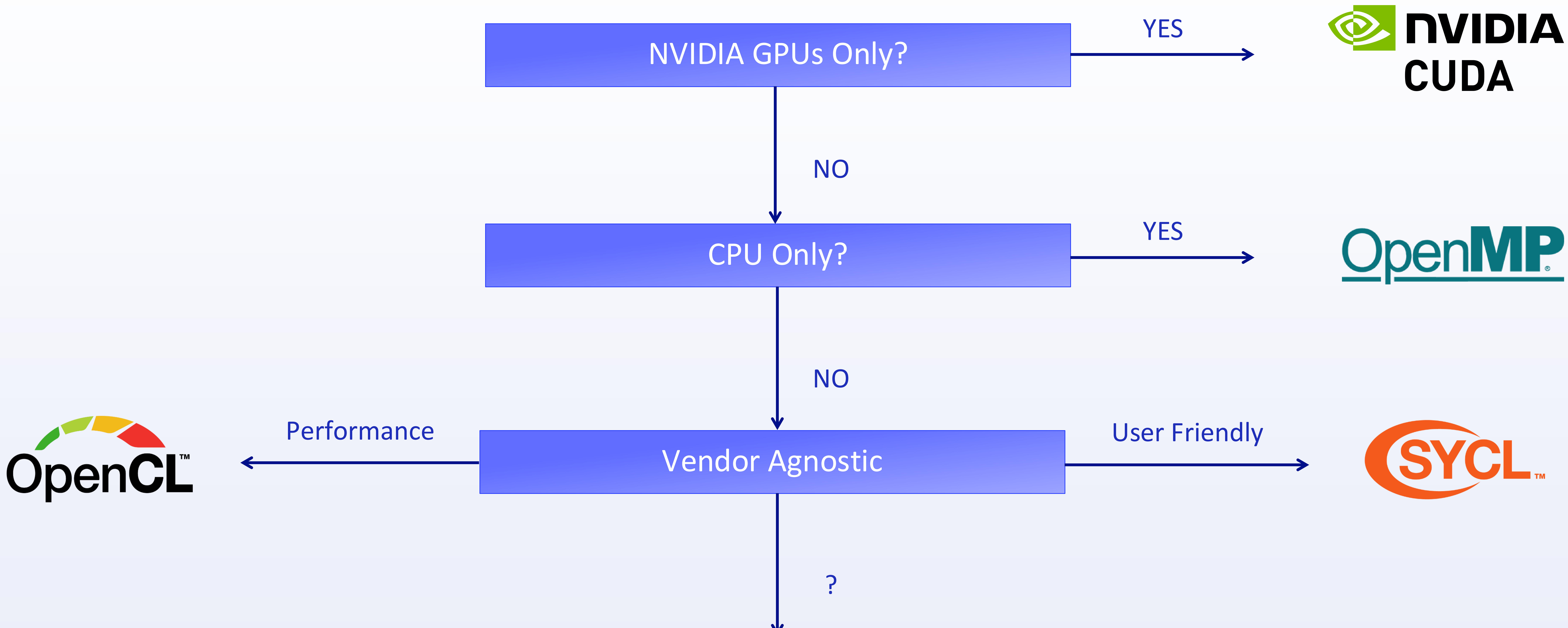
Open MP

```
int main() {
        char *array;
        ...

        int num_elements = gpu_workgroups * gpu_workitems_per_workgroup;
        #pragma omp target
        {
                #pragma omp parallel for simd num_threads(num_elements)
                for (int i=0; i < num_elements; i++)
                        array[i] += 5;
        }
}
```

Programming languages Trends



Which Programming Language To Choose?



Based on "A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware" by Marcel Breyer, Alexander Van Craen and Dirk Pflunger (University of Stuttgart)

What about C++?

std::async

```cpp
int add(int n) {
        return n + 5;
}

int main() {
        int out;
        int in = 6;

        out = add(in);

        cout << "The result is: " << out << endl;
}
```

```cpp
int add(int n) {
        return n + 5;
}

int main() {
        int out;
        int in = 6;

        std::future<int> fu = std::async(add, in);

        out = fu.get();
        cout << "The result is: " << out << endl;
}
```

```cpp
int add(int n) {
        return n + 5;
}

int main() {
        int out;
        int in = 6;

        std::future<int> fu = std::async(add, in);

        out = fu.get();
        cout << "The result is: " << out << endl;
}
```

```cpp
int add(std::future<int>& n) {
        return n.get() + 5;
}

int main() {
        int out;
        std::promise<int> in;
        std::future<int> n = in.get_future();

        std::future<int> fu = std::async(add, std::ref(n));

        in.set_value(6);

        out = fu.get();

        cout << "The result is: " << out << endl;
}
```

```cpp
int add(std::future<int>& n) {
        return n.get() + 5;
}

int main() {
        int out;
        std::promise<int> in;
        std::future<int> n = in.get_future();

        std::future<int> fu = std::async(add, std::ref(n));

        in.set_value(6);

        out = fu.get();

        cout << "The result is: " << out << endl;
}
```

```cpp
int add(std::future<int>& n) {
        return n.get() + 5;
}

int main() {
        int out;
        std::promise<int> in;
        std::future<int> n = in.get_future();

        std::future<int> fu = std::async(add, std::ref(n));

        in.set_value(6);

        out = fu.get();

        cout << "The result is: " << out << endl;
}
```

```
int add(std::future<int>& n) {
        return n.get() + 5;
}

int main() {
        int out;
        std::promise<int> in;
        std::future<int> n = in.get_future();

        std::future<int> fu = std::async(add, std::ref(n));

        in.set_value(6);

        out = fu.get();

        cout << "The result is: " << out << endl;
}
```

```cpp
int add(std::future<int>& n) {
        return n.get() + 5;
}

int main() {
        int out;
        std::promise<int> in;
        std::future<int> n = in.get_future();

        std::future<int> fu = std::async(add, std::ref(n));

        in.set_value(6);

        out = fu.get();

        cout << "The result is: " << out << endl;
}
```

Conducting a Heterogeneous System

std::execution (P2300)

Main Classes

Scheduler – schedules work on execution resources (e.g. GPU)

Senders – work

Receivers – where work is terminates using the channels:
Value, error or stopped

```cpp
#include <stdexec/execution.hpp>
#include <exec/static_thread_pool.hpp>


int main(){
    exec::static_thread_pool pool;

    stdexec::scheduler auto sch = pool.get_scheduler();

    stdexec::sender auto work = stdexec::schedule(sch) |
                stdexec::then([] {return 5;}) |
                stdexec::then([](int arg) { return arg + 7;});

    auto res = stdexec::sync_wait(work).value();

    return 0;
}
```

```cpp
#include <stdexec/execution.hpp>
#include <exec/static_thread_pool.hpp>


int main(){
    exec::static_thread_pool pool;

    stdexec::scheduler auto sch = pool.get_scheduler();

    stdexec::sender auto work = stdexec::schedule(sch) |
                stdexec::then([] {return 5;}) |
                stdexec::then([](int arg) { return arg + 7;});

    auto res = stdexec::sync_wait(work).value();

    return 0;
}
```

```cpp
#include <stdexec/execution.hpp>
#include <exec/static_thread_pool.hpp>


int main(){
    exec::static_thread_pool pool;

    stdexec::scheduler auto sch = pool.get_scheduler();

    stdexec::sender auto work = stdexec::schedule(sch) |
                stdexec::then([] {return 5;}) |
                stdexec::then([](int arg) { return arg + 7;});

    auto res = stdexec::sync_wait(work).value();

    return 0;
}
```

How Standards Proliferate

(See: A/C charges, character encodings, instant messaging, ETC)

SITUATION:
THERE ARE 14
COMPETING
STANDARDS

How Standards Proliferate

(See: A/C charges, character encodings, instant messaging, ETC)

How Standards Proliferate

(See: A/C charges, character encodings, instant messaging, ETC)

mobileye

Thank you!

Linkedin

Oren Benita Ben Simhon

Israel LLVM Meetup

Meetup

mobileye™