Introduction to writing and profiling GPU kernels

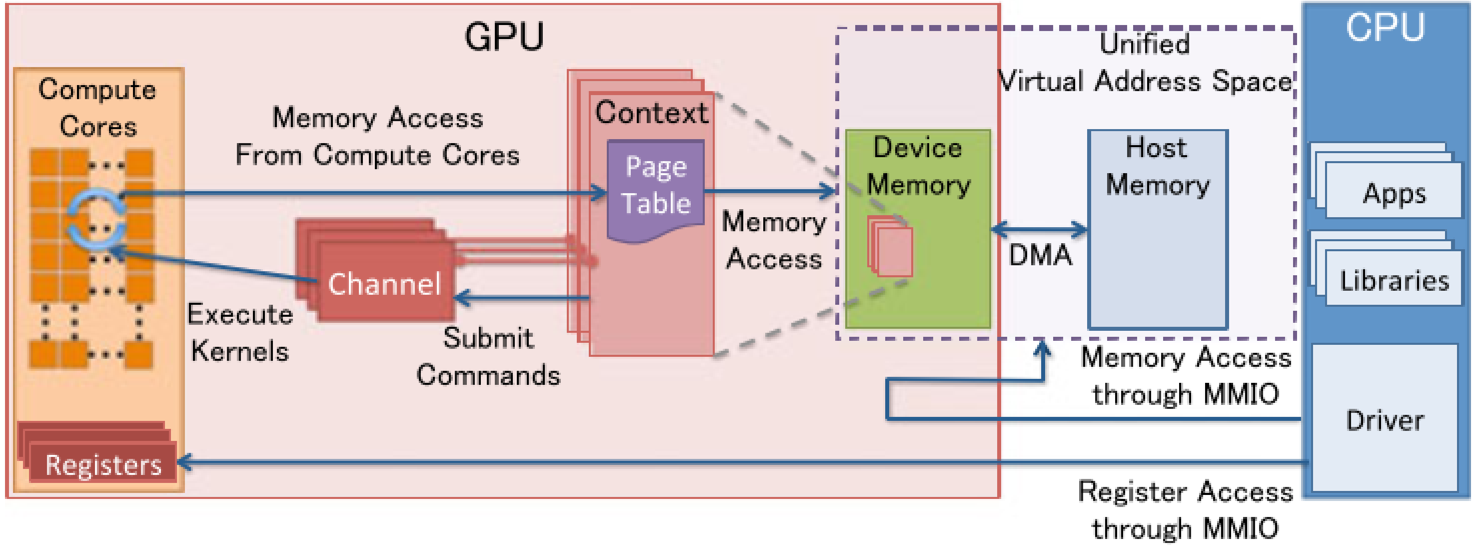
Fredrick Odhiambo

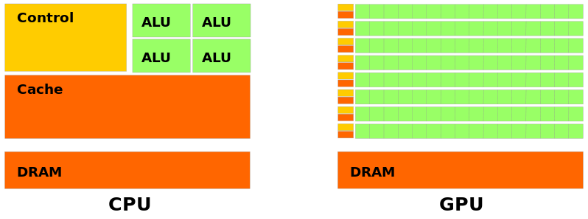
Software Optimization Engineer

Demystifying the Jargon

- dGPU = discrete Graphical Processing unit
- iGPU = integrated Graphical Processing unit
- FPS = Frames Per second
- FLOPs = Floating Point Operations per second
- TDP = Thermal Design Power
- VRAM = Video Random Access memory
- Overclocking = overclocking ☺

GPU vs CPU architectures – why bother





- CPUs are considered (to a degree ☺) "**smarter**" than GPUs (branch prediction, out of order execution) but GPUs can do **A LOT** of work quickly ---- > you decide
- GPUs use SIMD (All cores execute the same instruction)
- CPUs traditionally meant for sequential execution, CPU can offer a degree of parallelism but not comparable to GPU

When to use GPU and when not to

- Can a CPU handle the entire task within the required time?
- Can my code be parallelized?
- Can I fit all the data on a GPU? If not does it introduce an overhead? Memory bound? Memory access
- Target users? (probably most important)

Amdahl's law Speedup Factor = 1 / (1 – p)
p = fraction of the program that can be parallelized
Assume p = 1/2

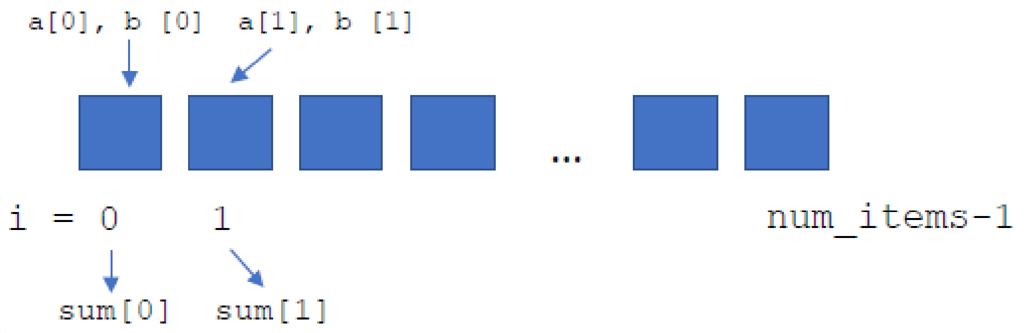
Parallelism….why bother?

`for_loop`: Iterate over data using the same operation:

```
For (int i = 0; < num_times;
 ++i) {sum[i] = a[i] + b[i];
};
```

`kernel`: Launch `num_items` kernel instances (work items) to be executed in parallel. Each instance uses separate pieces of data. `i` is a unique identifier in the execution range 0 … `num_items-1`:

```
h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] +
b[i]; });
```



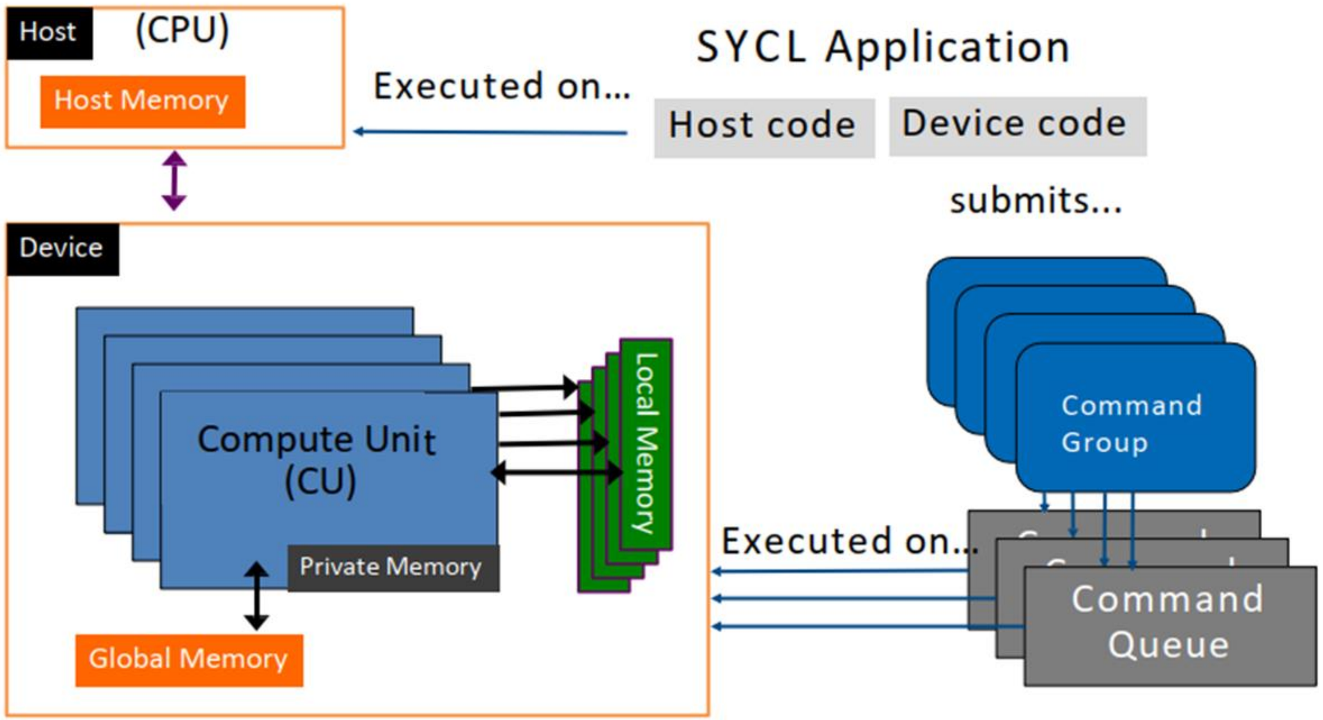
Num_items > 1000000

GPU Programming Frameworks

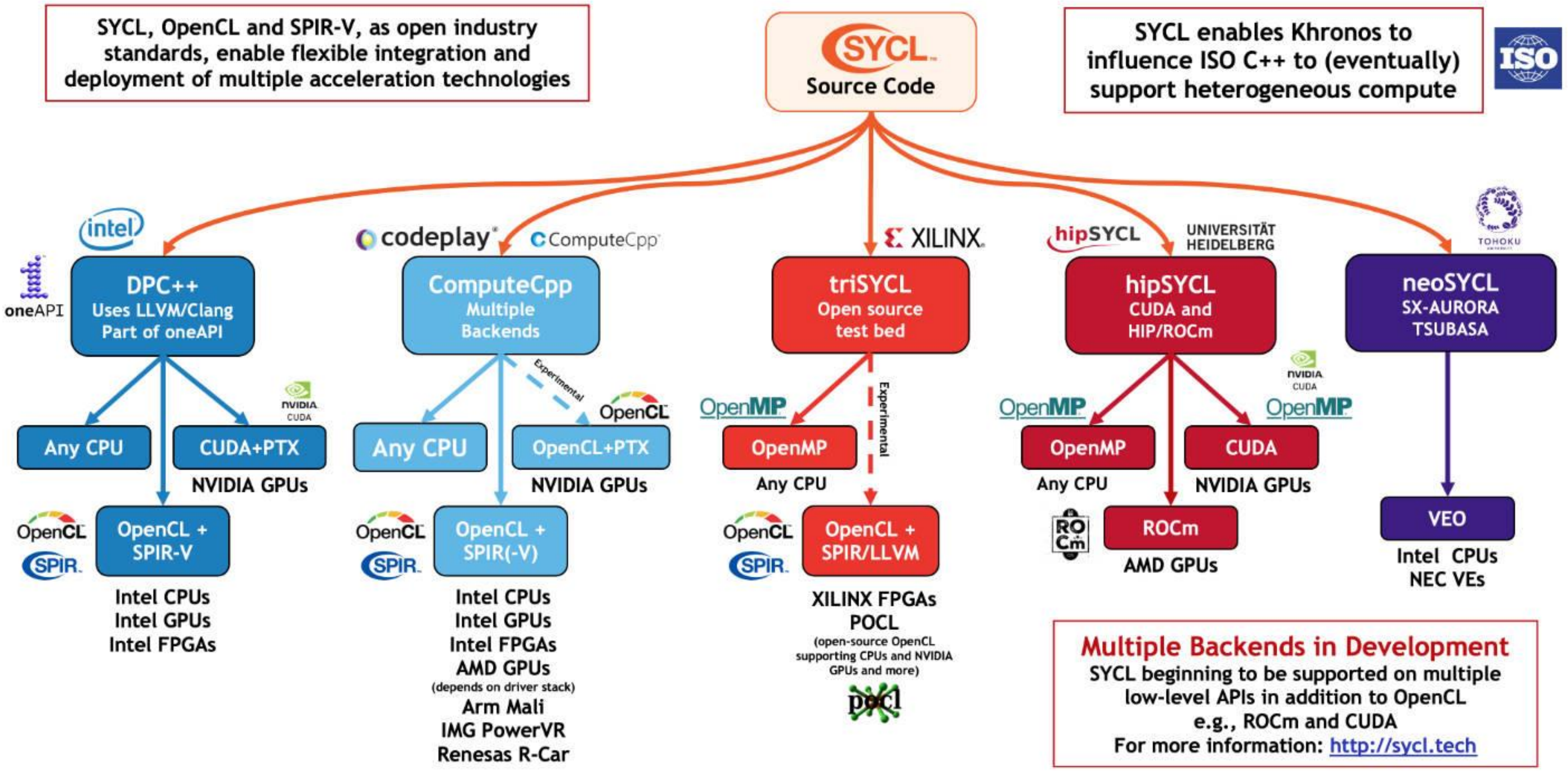
Framework	Maintainer/Vendor	Supported HW	Notes
OpenCL	Khronos Group	CPUs, FPGAs, GPUS	• Kernel based execution • OpenCL C (subset of C) • combination with other frameworks ie Vulkan, OpenGL • Multi vendor support (fragmentation) • Earliest in class
SYCL	Khronos Group	CPUs, FPGAs, GPUs	• Single source both host and device code written on the same file in C++ • Leverages templates & lambda functions • Based on modern C++ and builds on OpenCL • Interoperable with existing OpenCL kernel code • Interoperable across vendors*
CUDA	NVIDIA	Nvidia GPUs	• C-like API • Highly optimized libs (cuDNN, cuBLAS) • Kernel based execution (kernels = functions that run on GPU)
Metal	Apple	OSX,tvOS,iOs,iPadOS	• C-like API • Kernel based execution • Uses precompiled shaders and contains optimized shader libs
Vulkan	Khronos Group	CPUs, GPUs	• Low level access to GPU • Successor of OpenGL (rendering & gfx pipeline) • Highly portable across Operating systems • Explicit memory management • Steep learning curve • Promising

SYCL Anatomy

- Royalty free, cross platform
- Uses modern C++
- Borrows considerably from battle tested OpenCL
- Single source (repetition)



SYCL interfacing with ecosystem



SYCL CONSTRUCTS

Program structure

Queue - object that holds command groups to be executed on a SYCL device

Kernel – A function that executes in the device

`parallel_for` – A kernel invocation command that defines a kernel that is executed in parallel over a specified range of elements,

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

int main() {
    int data[1024]; // Allocate data to be worked on

    // Create a default queue to enqueue work to the default device
    queue myQueue;

    // By wrapping all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block,
    // because the destructor of resultBuf will wait

    {
        // Wrap our data variable in a buffer
        buffer<int, 1> resultBuf { data, range<1> { 1024 } };

        // Create a command group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // Request write access to the buffer without initialization
            accessor writeResult { resultBuf, cgh, write_only, no_init };

            // Enqueue a parallel-for task with 1024 work-items
            cgh.parallel_for(1024, [=](id<1> idx) {
                // Initialize each buffer element with its own rank number starting at 0
                writeResult[idx] = idx;
            }); // End of the kernel function
        }); // End of our commands for this queue
    } // End of scope, so we wait for work producing resultBuf to complete

    // Print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data[" << i << "] = " << data[i] << std::endl;

    return 0;
}
```

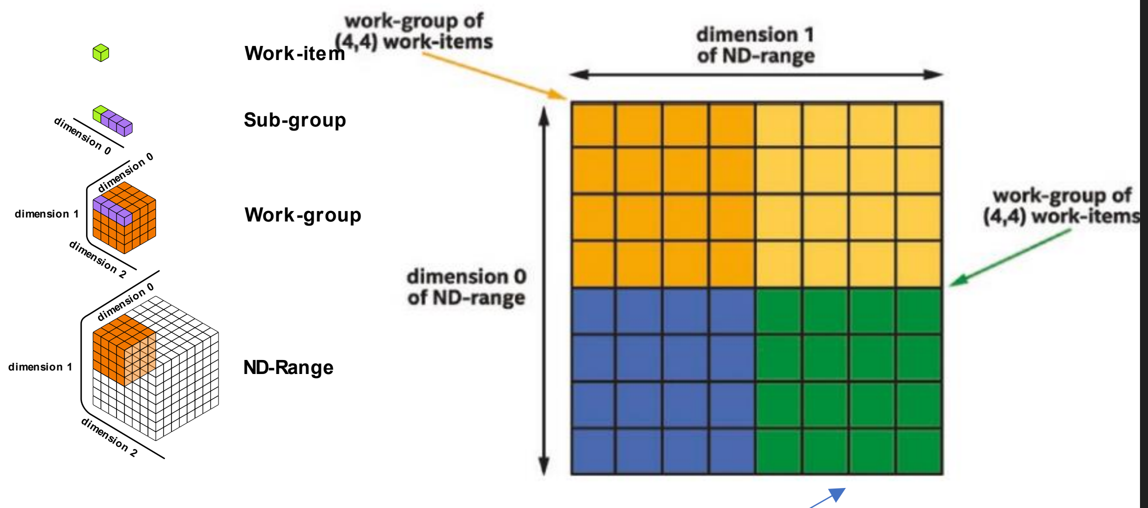
Host code

device code

Host code

ND Range, Workitems, workgroup

- **sycl::nd_range** – class that defines an ND-range which takes as input the sizes of local and global work items.



Two-dimensional ND-range of size (8, 8) divided into four work-groups of size (4,4)

```
// Define global and local sizes for 2D ND-range
sycl::range<2> global_size(8, 8);  // Total number of work-items (8x8)
sycl::range<2> local_size(4, 4);   // Work-group size (4x4)

// Define 2D ND-range with global and local sizes
sycl::nd_range<2> ndRange(global_size, local_size);

cgh.parallel_for(ndRange, [=](sycl::nd_item<2> item) {
    // Get global and local IDs in both dimensions
    size_t global_id_x = item.get_global_id(0); // Global ID in x-dimension
    size_t global_id_y = item.get_global_id(1); // Global ID in y-dimension

    size_t local_id_x = item.get_local_id(0);   // Local ID in x-dimension
    size_t local_id_y = item.get_local_id(1);   // Local ID in y-dimension

    size_t group_id_x = item.get_group(0);      // Work-group ID in x-dimension
    size_t group_id_y = item.get_group(1);      // Work-group ID in y-dimension

    // Storing some values for demonstration
    acc[global_id_x][global_id_y] = global_id_x + global_id_y + local_id_x +
     local_id_y + group_id_x + group_id_y;
});
```

ND Range, Work-items, workgroup

- 1Dimension ND range

```
const size_t global_size = 16;  // Total number of work-items
const size_t local_size = 4;    // Work-items per work-group

// Define ND-range with global and local sizes
sycl::nd_range<1> ndRange(sycl::range<1>(global_size), sycl::range<1>(local_size));
cgh.parallel_for(ndRange, [=](sycl::nd_item<1> item) {
            // Get global and local IDs
            size_t global_id = item.get_global_id(0);   // Global index in the range
            size_t local_id = item.get_local_id(0);     // Local index within the work-group
            size_t group_id = item.get_group(0);        // Work-group ID

            // Storing some values for demonstration
            acc[global_id] = global_id + local_id + group_id;  // Example computation
        });
```

SYCL memory abstractions

Buffers

- Abstraction managing memory +data transfer
- Implicit copying
- Need accessors to interact with buffers
- Easy to use, less control, less performant
- Have destructors (self cleaning)

```cpp
#include <CL/sycl.hpp>
using namespace sycl;

int main() {
    queue Q;

    int N = 10;
    auto R = range<1>{ N };
    buffer<int> A{ R };

    Q.submit([&](handler& h) {
        accessor A_acc(A, h);

        h.parallel_for(R, [=](auto indx) {
            A_acc[indx] = indx;
        });
    });

    host_accessor result(A);

    //# print output
    for (int i = 0; i < N; i++) std::cout << result[i] << " "; std::cout << "\n";
    return 0;
}

```

SYCL memory abstractions

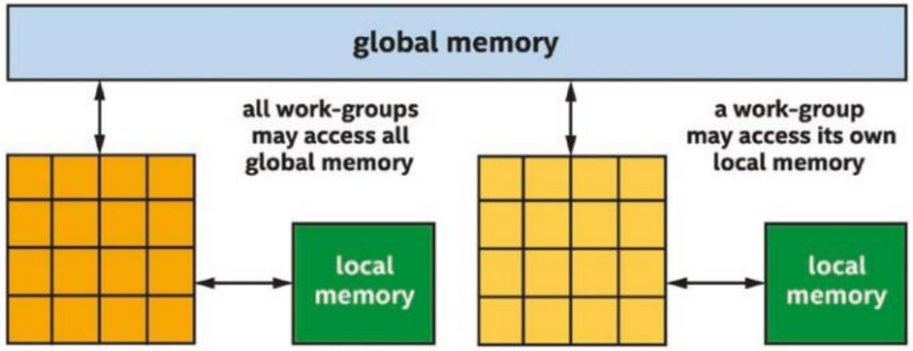
USM (Unified Shared Memory)

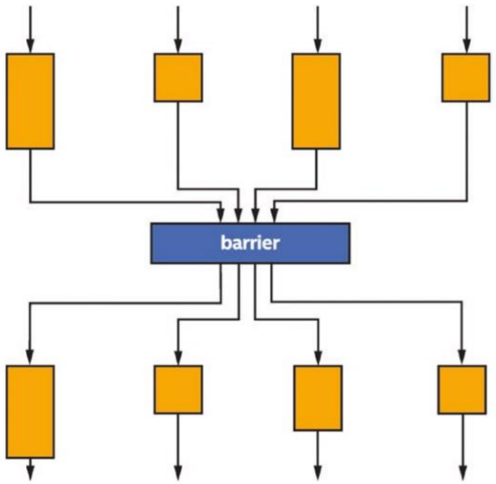
- Requires HW support (Always check!)
- Pointer based approach
- Consistent with C++
- Explicit data movement
- Require explicit cleanup
- More effort, more performant

```cpp
#include <CL/sycl.hpp>

int main() {
    // Create a SYCL queue, specifying the target device and properties
    sycl::queue q;

    // Allocate shared memory for the data
    int N = 10;
    sycl::malloc_device_ptr<int> data(N, q);

    // Initialize the data array
    q.submit([&](sycl::handler& h) {
        h.parallel_for(sycl::range(N), [=](sycl::id idx) {
            data[idx] = idx;
        });
    }).wait();

    // Perform your parallel computations using the data
    // ...

    // Copy the results back to the host if needed
    q.memcpy(sycl::host_ptr<int>(data.get()), data, N).wait();

    // Free the device memory
    data.free();

    return 0;
}
```

Synchronization

- **Work-group and Work-item Synchronization**
  - sycl::group_barrier()
  - sycl::nd_item::barrier()
- **Host-Device Synchronization**
  - sycl::queue::submit()
  - sycl::event::wait()
  - sycl::queue::wait()
- **Memory fences**
  - sycl::memory_fence()





Real world usages

```cpp
template <typename T>
void cpu_add_matrix(std::vector<std::vector<T>>&matrix_A, std::vector<std::vector<T>>&matrix_B, std::vector<std::vector<T>>&matrix_C )
{

    if(!matrix_is_empty(matrix_A) && !matrix_is_empty(matrix_B) && matrix_addition_is_possible(matrix_A,matrix_B))
    {

        auto start = std::chrono::high_resolution_clock::now();
        for(size_t i = 0; i <matrix_A.size(); i++)
        {
            for(size_t j = 0; j < matrix_A[0].size(); j++)
            {
                matrix_C[i][j] = matrix_A[i][j] + matrix_B[i][j];
            }
        }

        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed_time = (std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() ) /1000.0;

        std::cout<<"Elapsed time for CPU matrix addition is : "<<elapsed_time<< " seconds" << std::endl;

    }else{
        std::cout<<"ERROR:  operation aborted! Ensure matrix are non-empty and their dimensions match" <<std::endl;

    }

  return;
}
```

CPU matrix addition

GPU matrix addition

```cpp
void gpu_add_matrix(std::vector<std::vector<T>>&matrix_A, std::vector<std::vector<T>> &matrix_B, std::vector<std::vector<T>>&matrix_C, sycl::queue q)
{
    if(!matrix_is_empty(matrix_A) && !matrix_is_empty(matrix_B) && matrix_addition_is_possible(matrix_A,matrix_B))
    {
        std::cout << "Running on: " << q.get_device().get_info<info::device::name>() << std::endl;
        std::vector<T> flat_matrix_A = flatten_matrix(matrix_A);
        std::vector<T> flat_matrix_B = flatten_matrix(matrix_B);
        std::vector<T> flat_matrix_C = flatten_matrix(matrix_C);
        size_t M = matrix_A.size();
        size_t N = matrix_A[0].size();

        buffer<T,1> buffer_A(flat_matrix_A.data(),range<1>(M*N));
        buffer<T,1> buffer_B(flat_matrix_B.data(), range<1>(M*N));
        buffer<T,1> buffer_C(flat_matrix_C.data(),range<1>(M*N));


        auto start = std::chrono::high_resolution_clock::now();

        q.submit([&] (handler &h ) {
            auto a = buffer_A.template get_access<access::mode::read>(h);
            auto b = buffer_B.template get_access<access::mode::read>(h);
            auto c = buffer_C.template get_access<access::mode::write>(h);

            h.parallel_for(range<1>(M*N), [=] (id<1> idx) {

                c[idx] = a[idx] + b[idx];

            });
        });

        q.wait();
        auto end = std::chrono::high_resolution_clock::now();
        auto elapsed_time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()/1000.0;
         std::cout << "Elapsed time for GPU matrix addition is: " << elapsed_time << " seconds" << std::endl;
         auto host_result_access = buffer_C.template get_host_access();
          std::cout<<"here 3"<<std::endl;
         for(size_t i = 0; i < M; i++)
         {
            for(size_t j = 0; j < N; j++)
            {
                matrix_C[i][j] = host_result_access[i*N + j];
            }
         }
    }else {
          std::cout<<"ERROR:  operation aborted! Ensure input matrix are non-empty and their dimensions match" << std::endl;

    }
}
```

```cpp
template<typename T>
void cpu_multiply_matrix(std::vector<std::vector<T>> &matrix_A, std::vector<std::vector<T>>&matrix_B, std::vector<std::vector<T>>&matrix_C)
{

    // by definition, columns of matrix A must be equal to the rows of matrix B , this means the result matrix will be of dimensions rowsof
    // matrix A by columns of matrix B

    if(matrix_A[0].size() != matrix_B.size())
    {
        std::cout<<"ERROR: matrix_A * matrix_B is undefined because cols(mamtrix_A) is not equal to rows(matrix_B)";
        return;
    }


    size_t rows_A = matrix_A.size();
    size_t rows_B = matrix_B.size();
    size_t cols_B = matrix_B[0].size();

    auto start = std::chrono::high_resolution_clock::now();
    for (size_t  i = 0; i < rows_A; i++)
    {
        for(size_t j = 0; j< cols_B; j++)

        {
            for(size_t k = 0; k < rows_B; k++)
            {
                matrix_C[i][j] += matrix_A[i][k] * matrix_B[k][j];
            }
        }
    }
   auto end = std::chrono::high_resolution_clock::now();
   auto elapsed_time = (std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count() ) /1000.0;

    std::cout<<"Elapsed time for CPU matrix multiplication is : "<<elapsed_time<< " seconds" << std::endl;

}
```

CPU matrix multiplication

```
template<typename T>
void gpu_multiply_matrix_naive(std::vector<std::vector<T>> &matrix_A, std::vector<std::vector<T>> &matrix_B, std::vector<std::vector<T>> &matrix_C, sycl::queue q) {

    std::cout << "Running on: " << q.get_device().get_info<info::device::name>() << std::endl;

    auto flat_matrix_a = flatten_matrix(matrix_A);
    auto flat_matrix_b = flatten_matrix(matrix_B);
    auto flat_matrix_c = flatten_matrix(matrix_C);
    size_t M = matrix_A.size();
    size_t N = matrix_A[0].size();

    buffer<T, 1> buffer_A(flat_matrix_a.data(), range<1>(M * N));
    buffer<T, 1> buffer_B(flat_matrix_b.data(), range<1>(M * N));
    buffer<T, 1> buffer_C(flat_matrix_c.data(), range<1>(M * N));

    auto start = std::chrono::high_resolution_clock::now();
    q.submit([&](handler &h) {

        auto a = buffer_A.template get_access<access::mode::read>(h);
        auto b = buffer_B.template get_access<access::mode::read>(h);
        auto c = buffer_C.template get_access<access::mode::write>(h);

        h.parallel_for(range<2>(M, N), [=](id<2> idx) {
            size_t j = idx[0];
            size_t i = idx[1];
            for (size_t k = 0; k < N; ++k) {
                c[j * N + i] += a[j * N + k] * b[k * N + i];
            }
        });

    });

    q.wait();
    auto end = std::chrono::high_resolution_clock::now();
    auto elapsed_time = (std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()) / 1000.0;
    std::cout << "Elapsed time for naive GPU matrix multiplication is: " << elapsed_time << " seconds" << std::endl;

// Read data back to host
    auto host_result_access = buffer_C.template get_host_access();
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            matrix_C[i][j] = host_result_access[i * N + j];  // Copy data back
        }
    }
}
```

GPU matrix multiplication
call it naïve for now ☺

Profiling tools

- **AMD**: Radeon GPU Profiler, uProf, CodeXL (Legacy).
- **Intel**: VTune Profiler, GPA.
- **Cross-Vendor/Open-Source**: Perfetto, Apitrace, RenderDoc, Vulkan GPU-Assisted Validation.
- **Apple**: Xcode GPU Frame Debugger.**Other**: SYCL Profiler, TAU Performance System.
- **NVIDIA**: Nsight Compute, Nsight Systems, Visual Profiler, cuBLAS/cuDNN Profilers.

vTune Profiler (Launch vTune with naïve matrix multiplication workload)

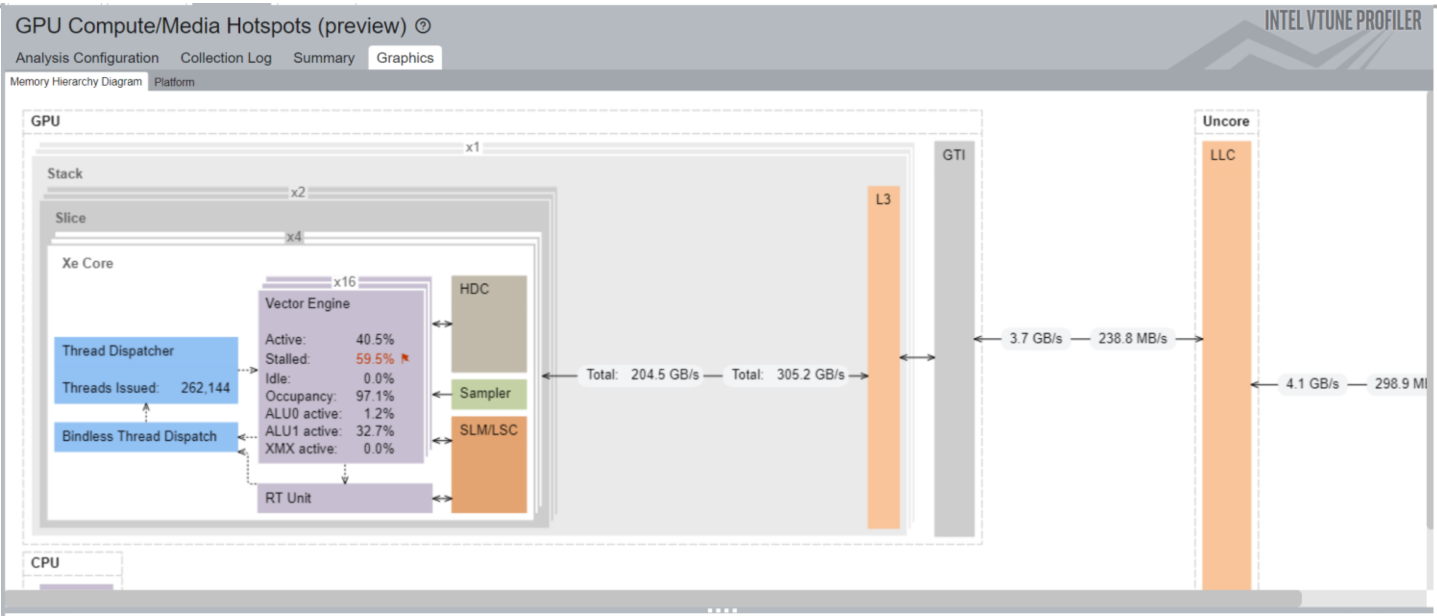
Hottest GPU Computing Tasks

A significant portion of GPU time is lost due to stalls. Use GPU Compute/Media Hotspots (preview) analysis to analyze HW usage efficiency.

U, sorted by the Total Time. Focus on the computing tasks flagged as performance-critical.

	Total Time	Execution Time	% of Total Time	SIMD Width	Peak XVE Threads Occupancy	XVE Threads Occupancy	SIMD Utilization
gpu_multiply_matrix_naive<float>(void, std::vector<std::vector<float, std::allocator<float>>, std::allocator<std::vector<float, std::allocator<float>>>>&, std::vector<std::vector<float, std::allocator<float>>, std::allocator<std::vector<float, std::allocator<float>>>>&, std::vector<std::vector<float, std::allocator<float>>, std::allocator<std::vector<float, std::allocator<float>>>>&, sycl::_V1::queue)::{lambda(sycl::_V1::handler&)#1}::operator()(sycl::_V1::handler&) const::{lambda(sycl::_V1::id<(int)2>)#1}	0.472s	0.469s	99.2%	8	100.0%	96.5%	100.0%
[Outside any task]	0.001s	0s	0.0%				

*N/A is applied to non-summable metrics.



Grouping: Computing Task

Computing Task	Work Size: Global ▼	Work Size: Local	Computing Task: Total Time	Computing Task: Average Time	Computing Task: Instance Count	Computing Task: SIMD Width	Computing Task: SVM Usage Type	Data Transferred: Size	Data Transferred: Total, GB/sec	XVE Array: Active	XVE Array: Stalled	XVE Array: Idle
gpu_multiply_matrix_naive<float>(void, std::vector<st	2048 x 2048	256 x 1	9.258s	9.258s	1	8		0 B	0.000	40.5%	59.5%	0.0%
[Outside any task]			0s	0s	0			0 B	0.000	1.1%	17.4%	81.5%

Optimization tips

- Maximize GPU occupancy
- Minimize Global Memory Access
- Minimize Divergence
- Minimize Data Transfer Between Host and Device (batch transfers + USM)
- Utilize shared memory
- Combine multiple kernels if possible, to avoid kernel invocation overhead
- Use tools Vtune to identify bottlenecks
- Optimize memory access patterns (access contiguous memory locations)
- Loop unrolling

```
auto start = std::chrono::high_resolution_clock::now();
q.submit([&](handler &h) {

    auto a = buffer_A.template get_access<access::mode::read>(h);
    auto b = buffer_B.template get_access<access::mode::read>(h);
    auto c = buffer_C.template get_access<access::mode::write>(h);

    h.parallel_for(range<2>(M, N), [=](id<2> idx) {
        size_t j = idx[0];
        size_t i = idx[1];
        size_t c_index = j * N + i;
        size_t a_index_base = j * N;
        T res = 0;
        for (size_t k = 0; k < N; ++k) {
            res += a[a_index_base + k] * b[k * N + i];  // change 1 use a local variable in private memory
        }

        c[c_index] = res;
    });

});

q.wait();
auto end = std::chrono::high_resolution_clock::now();
```

Optimized GPU matrix multiplication: Optimizations:

- Reduce global memory access by using a placeholder local variable to store accumulated result
- Precompute indices for elements
- There is room for more optimization (local memory)

Do it yourself at home

1. Download cmake, Intel one API toolkit, Ninja, clone my repo

    https://cmake.org/download/

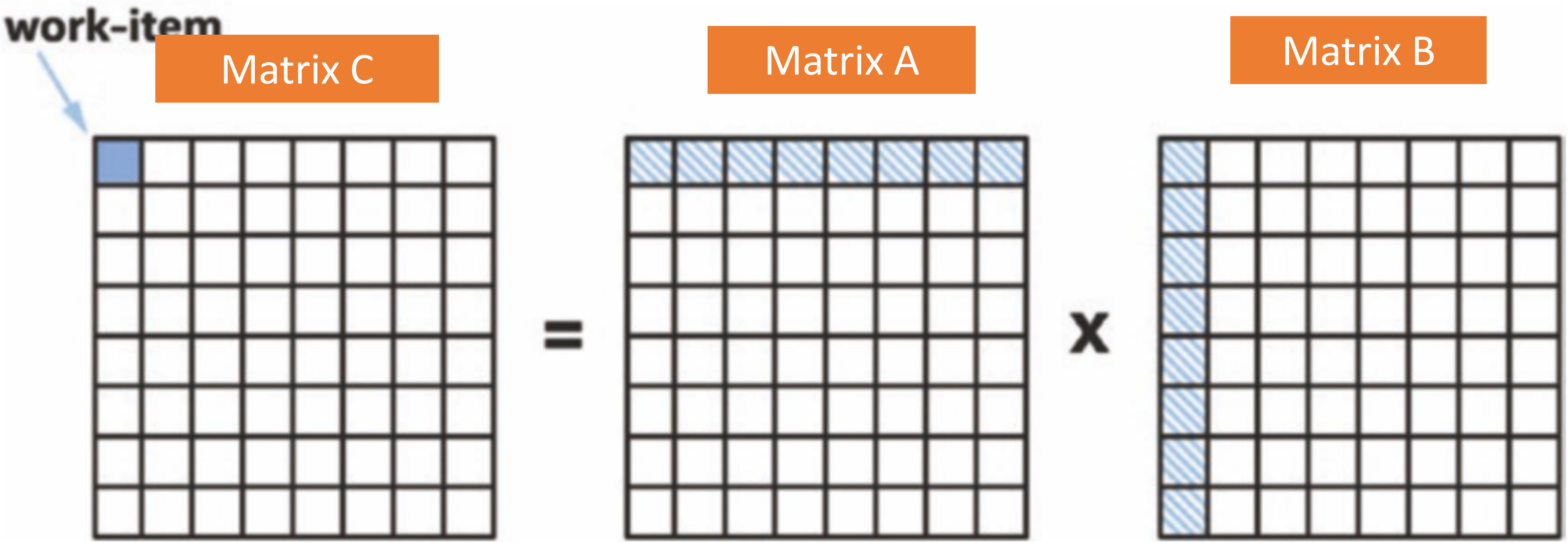
    https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit-download.html

    https://ninja-build.org/

    https://github.com/fredrickomondi/cpp_under_sea.git

2. Download Intel One API toolkit and Install (with admin priviliges)
3. Initialize oneAPI environment variables - this sets DPC++ compiler **C:\Program Files (x86)\Intel\oneAPI\setupvars.bat**
4. **Clone repository**
4. Go to the root of tutorial_app folder
5. **mkdir build && cd build**
6. Execute the following commands
    - **Cmake –G Ninja ..**
    - **Ninja**
    - **My_app.exe**

Homework ☺



Hint : exploit coalesced memory locations

And above all- Don’t overdo it ☺

Reference

- https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html