Introduction to GPU programming II: OpenMP and SYCL

NHR Graduate School 2025, Erlangen

Steffen Christgau

Zuse Institute Berlin

Agenda

Agenda

Native GPU Programming

- GPUs ≠ performance silver bullet
- Vendor-specific C/C++ APIs for GPUs

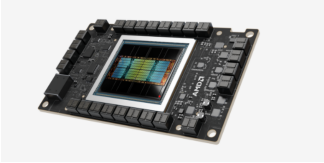
  CUDA Nvidia only, no portability

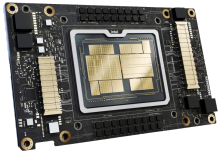
  HIP AMD's API, can be used for Nvidia GPUs too, limited portability
- Proprietary CUDA Fortran extension exists
- Good choice to tune code for device specific capabilities
- Best ecosystem support (Compilers, Debuggers, Profilers)

- **Downside:** Bound to vendor's devices
- **Not recommended for application developers**

Evolving Market

- Nvidia is dominating the market (still)
- But: Major processor vendors offer GPUs for diffrente markets





High-end GPUs: AMD MI300X, Intel GPU Max 1550, and Nvidia H100

- But: Top HPC systems with GPUs from other vendors exist
  - Frontier (AMD GPUs, TOP500 #2)
  - Lumi (AMD GPUs, TOP500 #7)
  - SuperMuc-NG Phase 2 (Intel GPUs, tba)
  - Hunter/Herder (AMD GPUs)
  - PVC-TDS-Partition am ZIB (Intel GPUs)
- What about your Notebook, Gaming PC, Workstation? What GPUs do you have there?

Vendor-Agnostic GPU Programming

Recommendations

- Code and performance portability is possible!
  - More abstract programming → not every platform detail accessible via APIs
- **Standard Language Features**
  - C++17 Execution Policies/Executors (Nvidia, Intel)
  - Fortran 2008 DO/CONCURRENT (Nvidia, Intel, AMD)
- **OpenMP** Offloading Constructs ⚡
  - Well-matured/established API for C/C++ and Fortran
  - Wide/improving GPU support (vendor and Open Source compilers)
- **SYCL** – C++ header-only library; requires compiler support ⚡
  - Currently pushed by Intel (oneAPI), but open standard
  - Compiler ecosystem evolves: 3rd party support for Nvidia/AMD

Vendor-Agnostic GPU Programming (cont'd)

Not so recommendations

- **OpenACC** support outside Nvidia ecosystem improves, but consider OpenMP!
  - supported by Nvidia, Cray, GCC

- **OpenCL** – Nah... Maybe for FPGAs

High-Level Frameworks

- Provide more **abstraction** from platform and architecture + **productivity**
- Express the algorithm, don't care much about the rest (DSL-like approach)
  - Framework cares about parallel execution/mapping to underlying frameworks/architectures
  - Varying levels of control
- Examples (most for C++)
  - Kokkos (Python binding exists)
  - Raja
  - Alpaka / Cupla
  - PETSc
  - ...

Libraries and AI

- Numerical libraries
  - All vendors provide libs for BLAS 1/2/3, FFT, collective communication etc
  - Often similar interfaces → wrappers exists (see oneMKL)
  - Prefer multi-platform libraries, like Magma, Ginkgo

- Existing MPI support for all GPU vendors (varying extent)
  - Both open source (MPICH, Open MPI) and vendor libraries (Intel MPI)

- Accelerated libraries for AI workloads
  - Vendor libraries exists, wrappers also (see oneDNN)
  - Hipified (AMD) PyTorch
  - Intel Extension for PyTorch (IPEX) → not needed anymore (!)

Agenda

Remember Pragmas?

- Compiler directives: steer compiler behavior (may have seen `#pragma once`)
- OpenMP is based on **pragmas** (C/C++): `#pragma omp...`
- Like annotations to source code
- Ideally: Ignore pragmas and you still have valid and working code.

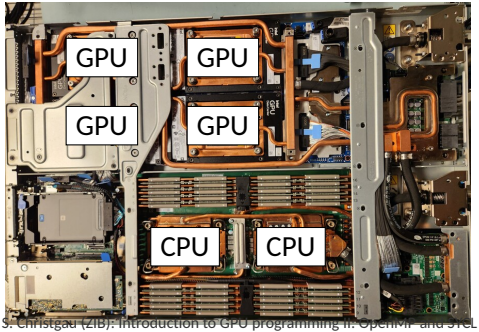
- **Essential one** for this tutorial: offloading pragma → `#pragma omp target ...`

- OpenMP **clauses** – control specifics for pragmas:
  `#pragma omp target [clause, ...]`

OpenMP Offloading Model

Established with OpenMP 4.0 (2013)

- Single **host** (processor/CPU)
- Multiple offload **devices** (GPUs) attached
- **Memory/address spaces not shared** between host and device (relaxations exists)
  → Two **data environments**: Host and Device
- **Data movements required!**



Consequences of Device Modell

Goal: Bring two essential building blocks onto the device

1. Execution
2. Data

Achieving First Goal: Code offloading

Make use of `target` construct! → Applied to a structured block (curly braces)

```
#pragma omp target
{
        /* your device code here */
} /* implicit barrier here */
```

1. `target` construct moves execution of structured block to device
   - Execution on host blocked until offloaded code finishes (*synchronous execution*)
   - Nothing is parallel, yet! 🔍
2. Also takes care of *some* data movements 🔍
3. Implicit barrier to wait for kernel to finish (sychronization)

Data Movement: What is done for you

`target` construct handles some data movements

- All scalar variables → **private copies** created on device
  - as `firstprivate` clause would do → values are copied
  - no transfer back to host
- Arrays on the stack are copied to *and* from device

```
double fill_array_on_device_and_get_n_half(...)
{
        const int n = 1000;
        double alpha = 3.14;
        double values[n]; // C/C++ Variable Length Array (VLA) - use with caution

        #pragma omp target
        for (int i = 0; i < n; i++) {
                values[i] = alpha;
        }
        return values[n / 2];
}
```

Data Movement: What you need to do

What about “dynamic arrays”, i.e. **pointers**?

- Pointer itself is a variable, but actual data needs to moved to device
- Your task: Tell compiler how large underlying data actually is.
- Your task: Tell compiler when you want data transfers.
- Compiler’s/Runtime’s task: Allocate data on device, do transfers.

Mapping of host data to device data is established.

Data Movement: Mappings

Tell compiler with mappings of your data with **map clause**:

```
#pragma omp target map(how:what)
```

How can be:

- **to** allocate and copy to device (when block is entered)
- **from** allocation and copy from device (when block is exited)
- **tofrom** combination of to and from
- **alloc** just allocate, no copies made
- **...** there's more

What should be: Your pointer and which range is to be copied:

```
#pragma omp target map(tofrom:my_array[start:num_elements])
```

Data Movement: Example with data mapping

```
float fill_array_on_device(float *buffer, size_t n)
{
        float alpha = 3.14f;

        #pragma omp target map(tofrom:buffer[0:n])
        for (size_t i = 0; i < n; i++) {
                buffer[i] = alpha;
        }
        return buffer[n / 2];
}

int main(void)
{
        const size_t size = 100000;
        float *data = new float[size];
        float check_value = fill_array_on_device(data, size);
        std::cout << check_value << std::endl; // what do you expect be printed?
        delete data;
}
```

Compile and Execute: What you need?

1. A GPU → get one in the NHR center of your choice → Today: Alex at NHR@FAU Nvidia GPUs (A40)
2. A **compiler** and runtime that supports your GPU

AMD	`aocc` for C/C++ or `flang` for Fortran
Intel	`ic[p]x` for C/C++ or `ifx` for Fortran
Nvidia	`nvc[c,++]` for C/C++ or `nvfortran` for Fortran → **Choice for today.**

Compile and Execute: How to do it

For Nvidia GPUs and Nvidia compiler

- **Compilation**: `nvc++ -mp=gpu -Minfo=mp mycode.cpp -o mybinary`
  - Enable support for offloading OpenMP to GPU
  - Get Information about OpenMP compilation → can be helpful
- **Execution**: As usual. Just launch your binary: `./mybinary`
- Helpful **environment variables** and settings
  - `OMP_TARGET_OFFLOAD`, set to `MANDATORY` to force offloading
  - `NVCOMPILER_ACC_NOTIFY`, set to `31` to see what's going on[1]
  - For LLVM-based compilers: `LIBOMPTARGET_INFO`

[1] https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#id27

Try yourself!

1. Launch JupyterHub just like yesterday (choose **Alex 1x A40 / Container** type)
2. Clone git repository for tutorial `https://git.zib.de/bzbchris/nhrgrads2025`
3. Open the OpenMP notebook (openmp.ipynb), work through *Compilation and Execution* and *Data movement* sections.
4. Take a look at the output of compilation and execution!
5. What do you observe?

Bio Break

Better Data Movement Control

`target`-clause not always bound to actual device code

- **Avoid data movements to/from GPU where possible**
- Structured code: Multiple functions operating on the same data on the GPU
- Methods of classes: Constructor creates data, destructor does destoys

Not so good example:

```
while (some_condition) {
        #pragma omp target map(tofrom:buffer[0:n])
        for (i = 0; i < buf_size; i++) {
                buffer[i] = ...
        }
}
```

Beneficial: separate `map` from `target` construct

Better Data Movement control

More constructs to the rescue

- `#pragma target data map(...)` *`structured block`*
  Note: structured block, i.e. code, is required

- `#pragma target enter data map(...)` map data to device
- `#pragma target exit data map(...)` *un*map data from device

Last two are standalone directives → no device code block needed

Example: Standalone Mappings

```
class SomeCalculation {
        private:
                size_t m_n;
                double *m_buffer;
        public:
                SomeCalculation(size_t n) /* constructor */
                        : m_n(n)
                {
                        m_buffer = new double[m_n];

                }

                void run() { ... }

                ~SomeCalculation() /* destructor */
                {

                        delete [] m_buffer;
                }
}
```

Example: Standalone Mappings

```
class SomeCalculation {
        private:
                size_t m_n;
                double *m_buffer;
        public:
                SomeCalculation(size_t n) /* constructor */
                        : m_n(n)
                {
                        m_buffer = new double[m_n];
                        #pragma omp target enter data map(to:buffer[0:m_n])
                }

                void run() { ... }

                ~SomeCalculation() /* destructor */
                {
                        #pragma omp target exit data map(delete:buffer[0:m_n]) /* or from?! */
                        delete [] m_buffer;
                }
}
```

Recap

What we have achieved so far:

- Execution gets moved to device (`#pragma target`) ✔
- Data gets allocated and moved between host and device with `map(...)` clause in target region or other `data` constructs ✔

What's missing? Parallelism/Performance! 🚀

Example for this Block: Vector Copy

Sounds simple, but helps us to understand parallelism.

- Goal: Copy data from one array *efficiently* to another one.
- Something like
  ```
  for (size_t i = 0; i < N; ++i) { dst[i] = src[i]; }
  ```
- Goal: Better understanding of OpenMP worksharing constructs on GPU

Benchmark

Task (copy data from A to B) sounds familiar? → **Stream-Benchmark**

- Stream was made for CPU (John McCalpin)
- **BabelStream** for GPU (U Bristol, Tom Deakin et al.)

  Implementation using different programming frameworks

  - OpenMP
  - OpenACC
  - CUDA
  - HIP
  - SYCL
  - ...

What can we expect?

- Educated Guess: Problem is *memory-bound* → much more memory accesses than computation.
- Datasheet: Memory bandwidth **696 GB/s**
- BabelStream for `double` data type:
  - CUDA Copy: 658 GB/s (94.5% of peak)
  - OpenMP Copy: 651 GB/s (93.5% of peak)

Try Yourself

- Go to OpenMP notebook
- Follow steps in section *Stream: A First Try*
  - Add mappings for data needed on device?
  - Achieve code offloading in `stream` function (template)

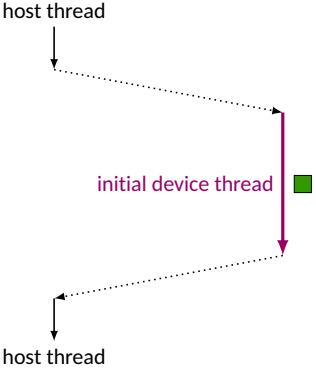
- What bandwidth do you achieve?
  - Use `double` as data type
  - Try with 128 M elements, will take about 2 Minutes
  - Reminder: Datasheet promised 696 GB/s

- Disappointed?!

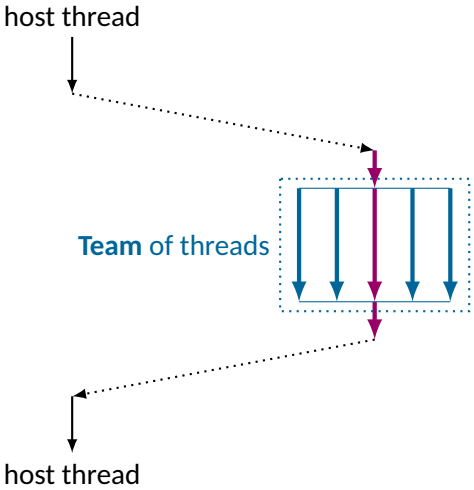
OpenMP offloading in detail

Quote from OpenMP 6.0 Specification, Section 1.2 Execution Model:

> *An initial thread executes the enclosed target region.*
> *The initial thread executes sequentially, [. . .]*



Introduce Parallelism: `parallel` construct



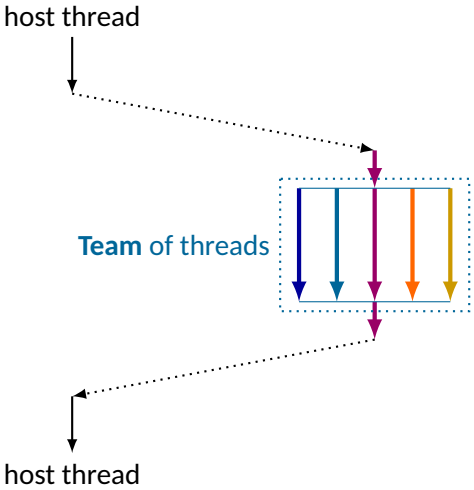
*The parallel construct creates a **team** of OpenMP threads that execute the region.*

All threads in the new team, including the master thread, execute the region

- `#pragma omp parallel` enables parallelism
- **Your task**: Add parallel construct to stream loop!

- Parallel provides **no worksharing**
- work is replicated among team members

Enable Worksharing: `for` construct



Worksharing-Loop Construct: *Iterations are distributed across threads in team*:

```
#pragma omp for
for (i = 0; i < n; i++) { ... }
```

Can be merged with parallel construct

```
#pragma omp parallel for
for (i = 0; i < n; i++) { ... }
```

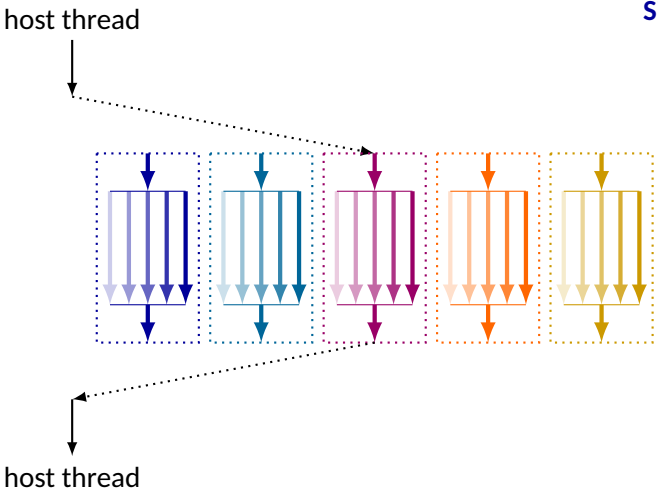
- **Your task**: Add `parallel for` to stream loop
- Does performance increase? Yes.
- Still not close to promised/observed values.

Remember GPU-Architecture

Example of GPU architecture (not A40)

- GPU has multiple levels of parallelism
  - Multiple Streaming Multiprocessors (SMs)
  - Multiple Processing Elements (PEs)
- Numbers for A40
  - 84 SMs
  - Total of 10752 PEs (CUDA cores)
- Multiple levels addressed by OpenMP
  - Parallel – accross PEs inside SM (*Team* of Threads)
  - **Teams** – accross SMs inside GPU (**League** of Teams)
- Teams are created by `teams` construct
- Work must be shared among teams/within league: `distribute` construct

Putting it together



Six constructs

- `target` – brings code to device, creates initial thread on device
- `teams` – create league of teams with individual initial threads
- `distribute` – distribute work among teams
- `parallel` – creates mutiple threads inside existing teams
- `for` – distribute work within team
- `simd` – make use of data parallelism

BUD: The Big Ugly Directive

Six constructs can be combined together

- `#pragma omp target teams distribute parallel for simd` *`for loop`*
- Loop iterations are distributed among teams and within team
- No synchronization between teams → **loop iterations must be independent**

Controlling Parallelism

You can control the degree of parallelism

1. `num_teams` – sets upper limit for number of created teams (note: it's a limit)
2. `thread_limit` –
3. `num_threads` – sets number of threads in parallel region

Try yourself: Multi-Level Parallelism

- Take stream example
- Extent `target parallel for` by BUD constructs
  1. `teams`
  2. `teams distribute`
  3. `simd`

  and measure achieved bandwidth. What values do you achieve?
- Adjust `num_teams` for `teams` and `num_threads` for `parallel` construct.
  - What are the defaults chosen by compiler/runtime?
  - To what do teams and threads map on Nvidia hardware?
  - Can you get better bandwidth than with the default values?
  - Is this portable?

Mapping: CUDA and OpenMP terms

- OpenMP *team* maps to CUDA *thread block* (executed on SMT)
  - Teams work indepedent of each other, no sychronization possible
  - All OpenMP *teams* map to CUDA *grid*
- OpenMP *threads* map to CUDA *threads* within block (executed on CUDA core)
- OpenMP *SIMD* is effectively ignored by Nvidia Compiler (simdlen=1)
  - Can be different for different compilers! (see Cray)
  - Hardware can be different
  - Don't leave `simd` out

The `loop` construct

A loop construct specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.

- Can be used to replace BUD
- `#pragma omp target loop`
- Important: You have a contract with compiler that **loop iterations are independent**!

Try yourself: Loop construct

- Replace BUD with loop construct.
- What bandwidth do you achieve with this version?

Bio Break

Recap

What we have achieved so far:

- Code execution on device (`#pragma target`) ✔
- Data gets allocated and moved between host and device with `map(...)` clause in target region or other `data` constructs ✔
- Achieve parallelism on device with BUD or `loop` construct 🚀

What's missing? Dealing with device memory

Extended Example: Matrix Addition

We want to add two matricies.

- $C_{ij} = A_{ij} + B_{ij}$ (element-wise addition)
- Find sequential code in notebook's `3_matrixadd` directory.

- Again educated guess: Code is still memory bound (three memory accesses plus one floating-point operation)

- How would **you** port this code to the GPU with OpenMP?
  - What data needs to be moved to GPU?
  - Which functions are offloaded?
  - How do you achieve parallelism?

Try yourself: Port code to GPU

- Port `matrix-add.cpp` example to GPU
- What performance do you observe?
- Are you getting close to maximum device bandwidth?

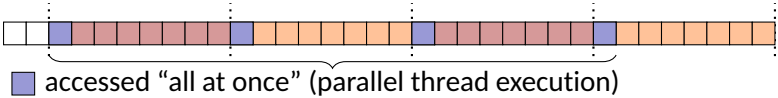
Memory Access Patterns

Assume two nested loops: first/outer one's iterations are distributed accross GPU.

Two possibilities to code this:

1. ij-Loop (row-major) – Iterate over rows, then over columns inside current row
   `idx = i * n_columns + j` ← different `i`'s on threads

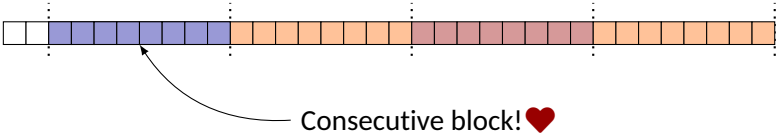
   Main Memory (with matrix data)

2. ji-Loop (column-major) – Iterate over column, then over rows inside current column
   `idx = i * n_rows + j` ← different `j` on threads

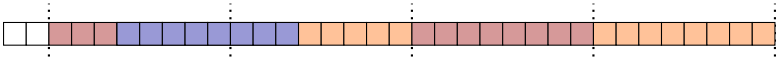
Coalesced Memory Accesses



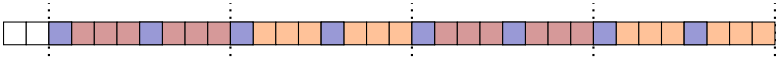
- GPUs are bandwidth-optimized → wide memory interfaces
- Access of consecutive addresses accross threads is preferred access patterns
  Turn accesses from multiple threads into one → **coalesce memory access**
- Ideally aligned to boundaries (for A100/A40: 64 Byte)

More Memory Access Patterns

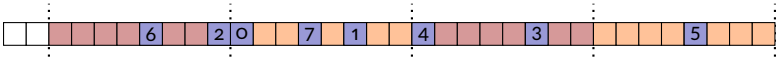
- Constant offset: `x = mem[idx + c]`



- Non-unit stride: `x = mem[idx * stride]`



- Random: `x = mem[random()]`



Try yourself: Memory Access Patterns

- Modify matrix add and implement different patterns
  - Constant offset
  - Non-unit stride
  - Random
- Pay attention not to cross memory boundaries.
  Use modulo operator (`%`) to stay in array bounds.
- Try to keep the number of total accesses the same between all versions

Recap

What we have achieved so far:

- Code execution on device (`#pragma target`) ✔
- Data gets allocated and moved between host and device with `map(...)` clause in target region or other `data` constructs ✔
- Achieve parallelism on device with BUD or `loop` construct 🚀
- Understand working with device memory

Bio Break

More OpenMP: collapse

- Clause for loops
- `#pragma ... for collapse(n)`
- Apply for construct to the n nested loops → think of merged loops
- Allow more work to be done in parallel.

Example:

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < ny; i++) {
        for (int j = 0; j < nx; j++) {
                idx = i * nx + j;
                ...
        }
}
```

More OpenMP: Reductions

- Reduce clause (no pragma) `reduction(op:vars)`)
- Perform reduction of variables on other constructs (`parallel`, `for`, `loop`)
- Built-in reductions: `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`
- Can be tedious/error-prone when doing it efficiently on your own
- Reduction variables are implicitely mapped `tofrom`

Example:

```
#pragma omp target teams distribute parallel for simd reduce(+:sum)
for (int i = 0; i < ny; i++) {
        for (int j = 0; j < nx; j++) {
                int idx = i * nx + j;
                sum += a[idx]
        }
}
std::cout << sum << std::endl;
```

More OpenMP: if

Sometimes you know, it is better not to offload/parallelize

- `if(condition)` clause
- Applies to `target` and `parallel` constructs
- Construct "does not take action" if conditions evals to false
  - `offload` → execution stays on host
  - `parallel` → no threads created

More OpenMP: ...

There's much more to discover:

- Tasks: More flexible worksharing; also allow to define dependencies
- Handling of multiple devices
- Atomic operations/synchronization
- Asynchronous Tasks
- InterOp with "low-level" APIs (CUDA, HIP, ...)
- ...

Get familiar with what you learned today first. Advance afterwards.

Hands-On: BYOC

Hands-On Session: Let's practise some more.

You can choose:

- Re-Iterate examples from Notebook
- Try porting your own code
- Maybe (not so) easy: easyWave
  - github.com/christgau/easywave-sycl
  - CUDA, HIP, SYCL versions exist → GPU code exists
  - Try to port to OpenMP
  - Hint: Start with CUDA. No need to understand everything in detail

Congrats! You made it!

Take-away messages:

- OpenMP makes *portable* GPU programming easy.
- Make use of parallelism on parallel hardware!
- Data movement and memory access patterns are critical (and easily done wrong).

Questions! Discussion!

Further Reading and Doing

- Tom Deakin and Tim Mattson (2023): Programming Your GPU with OpenMP
- Ruud van der Pas, Eric Stotzer and Christian Terboven (2017): Using OpenMP—The Next Step Affinity, Accelerators, Tasking, and SIMD
- The OpenMP Specification

- Attend other Tutorials
- Code, Code, Code

Sources/References

- T. Mattson, T. Deakin: Programming Your GPU with OpenMP – A “hands-on” Introduction. SC’23
- C. Terboven, M. Klemm, B. de Supinski: Advanced OpenMP Tutorial, ISC’22
- S. Pophale: Introduction to OpenMP Device Offload, 2022

- Sebastian Kuckuk SebastianKuckuk/apex
- Xin Wu pc2/OMP-Offloading

Agenda

Recap from Yesterday

Focus was on OpenMP

- Pragmas (compiler steering commands) to instruct compiler
- Developer marks offloaded code and controls data movements
- Compiler support required for device offloading.

Today: SYCL

SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

Let's disect this sentence.

What is SYCL?

*SYCL is a **single source**, high-level, standard C++ programming model, that can target a range of heterogeneous platforms*

You don't need to split your source code for host (CPU) and device (GPU). → Just like in OpenMP.

What is SYCL?

*SYCL is a single source, **high-level**, standard C++ programming model, that can target a range of heterogeneous platforms*

Abstractions for management of low-level tasks, such as device handling, kernel launch, creation of data region etc. → Details are hidden from you, similar to OpenMP

What is SYCL?

*SYCL is a single source, high-level, **standard C++** programming model, that can target a range of heterogeneous platforms*

Unlike CUDA or OpenMP no language extensions or additional compiler pragma are needed.
However: You need a SYCL compiler (like you need an OpenMP compiler)

What is SYCL?

SYCL is a single source, high-level, standard C++ programming model, that can ***target a range of heterogeneous platforms***

You can target different *device types*, not only GPUs.
Device support depends on SYCL implementation

SYCL Implementations

Intel oneAPI Intel's SYCL implementation. The icpx compiler fully supports the Khronos SYCL 2020 specification. Not limited to Intel products with help of plugins (see today).

Inte LLVM Intel's open source LLVM compiler supports SYCL as well (base for oneAPI's icpx)

AdaptiveCpp Independent, community-driven modern platform for C++-based heterogeneous programming.

neoSYCL A SYCL implementation for SX-Aurora TSUBASA.

SimSYCL A single-threaded, library implementation of SYCL 2020 for testing SYCL applications against simulated hardware.

Anatomy of a SYCL Program

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

int main()
{
    // create Queue instance
    sycl::queue defaultQueue;

    // ...
}
```

- SYCL is based on C++17 → Template-based library with compiler/implementation support
- Header `sycl/sycl.hpp` file is required.
- Sometimes `using namespace sycl` is used. Don't do this.
- `sycl::queue` is your access point to an offloading device. ϟ
- Device model is similar to OpenMP's: Multiple devices attached to a host with separate address spaces

Steering Device Selection

- `sycl::queue` constructor accepts a *device selector* as parameter
- SYCL allows more than GPU offloading
- Predefined ones are:

  default_selector_v – Default one
  gpu_selector_v – Selects a GPU of the node the code runs on
  cpu_selector_v – Selects a CPU of the node the code runs on
  accelerator_selector_v – Select an accelerator → FPGAs!
- Available devices depends on SYCL implementation → run `sycl-ls` to see them
- For Intel, environment variable `ONEAPI_DEVICE_SELECTOR` can steer default selector
  - Fine grained filtering possible[2]
  - Example: `ONEAPI_DEVICE_SELECTOR="*:gpu"` → select GPUs only

[2] see https://intel.github.io/llvm/EnvironmentVariables.html

Compile and Execute

- Similar to OpenMP you need a SYCL compiler.
- For Intel oneAPI: `icpx` (remember, SYCL is C++-based)
- To activate SYCL support, provide `-fsycl` flag.

  Example: `icpx -fsycl mycode.cpp -o mybinary`

- Launch the binary as usual: `./binary`

- Important: Device selection does not neccessarily mean your code will work there.
  If you compiled for a certain device type (see later), selecting another device type will likely very cause errors.

Try yourself: Show Name of selected Device

Create your working environment:

- Open JupyterHub as yesterday (use A40 **container** type)
- Clone repository from `https://git.zib.de/bzbchris/nhrgrads2025`
- Open SYCL notebook and follow steps in *Device Selection*

Bio Break

Goals

What we need to use the GPU (not different from OpenMP):

- Execute code on device.
- Get data allocated and moved between host and device `data` constructs.
- Exploit GPU parallelism.

Run Code on Device

- **The queue** is key to device access!
- Actually: Need a device context to talk to device, but provided through queue automatically.
- Queue is used to submit *commands* ("work") to the device.
- Work is essentially a **kernel function** which returns void, i.e. nothing
- Two major kinds of work submission (others exist):

  single_task – sequentially execute kernel on device
  parallel_for – launch kernel code in parallel for given number of *work items*.

- **Kernel execution is asynchronous** w.r.t. the host!
  Submitting a kernel does not wait for completion of execution (Different to OpenMP!)
- Queue is **out of order** → FIFO does not apply!

Submitting a Kernel

single_task submission not covered here → Focus on parallel_for.

Example:

```
sycl::range<2> domain_size(1024, 1024);
queue.parallel_for<class mykernel>(domain_size, [=](sycl::item<2> workitem) {
        /* your code here */
});
```

Things to notice:

- **No loops.** You just state what is done per item in your compute domain
  Data parallel programming (see loops in OpenMP)
- *Iteration space* provided by multidimensional *sycl::range* (not via loop boundaries)
- Kernel function will be instantiated/executed for each item in iteration space.

Submitting a Kernel

Example:

```
sycl::range<2> domain_size(1024, 1024);
queue.parallel_for<class mykernel>(domain_size, [=](sycl::item<2> workitem) {
        /* your code here */
});
```

Further things to notice:

- Kernel function is a C++ **lambda expression**/unnamed function (or class)
- Kernel can be named with template parameter to submitting function → Can be beneficial when using tools like profilers.
- **Variables captured by value** (`[=]`) → copies are created (see OpenMP)
  Restriction: Data **type must be device-copyable** (trivially copyable)

Handle Queue Properties

Remember: Kernel execution is asynchronous to the host

- If need to wait for completion of work: `queue.wait()`
- Can be used to maintain order of submissions
- `wait()` may cause lots of overhead (series of submit+wait)
- Alternatives:
  1. Create queue with **in-order** semantics:
     `sycl::queue q(sycl::property::queue::in_order{});`
  2. Use `sycl::event` to **define dependencies**
     Submitting work to queue returns event and accepts those as dependencies.
  3. Use **SYCL Graphs** for repetitive work submissions (experimental extension)
- Will stick to #1 for this tutorial

Remark: We're using shortcuts!

Actually ...

- ...we need to submit a **command group**
- ...there's a **handler** object that allows to add **requisites** (at most) one **kernel** function to the submitted command group

```
queue.submit([&](sycl::handler &cgh) {
        // add requisites

        // add work the command group
        cgh.parallel_for<class mykernel>(range, [=](...) {
        });
});
queue.wait();
```

Buffers and Accessors

SYCL's Original Way to Handle Memory

Separation of storage and data access

- Storage – `sycl::buffer`
- Access – `sycl::accessor`

Consequences of this model:

- Buffer takes ownership of data for it's lifetime and manages transfers upon *access*.
- Accessor are created (within command group) on buffer with access type
- Access to data in kernel must be done through accessor
- Dependencies detectable by runtime
- Accessor represents requisite

Example with Buffer and Accessors

```
std::vector<double> vec(n);
{
        sycl::buffer vec_buf(vec.data(), sycl::range{n});

        queue.submit([&](sycl::handler& cgh) {
                sycl::accessor acc_write_vec = vec_buf.get_access<sycl::write_only>(cgh);
                cgh.parallel_for(n, [=](sycl::id<1> id) {
                        acc_write_vec[id] = id;
                });
        });

        queue.submit([&](sycl::handler& cgh) {
                sycl::accessor acc_read_vec = vec_buf.get_access<sycl::read_only>(cgh);
                cgh.parallel_for(n, [=](sycl::id<1> id) {
                        ... = acc_read_vec[id] ...;
                });
        });
        // buffer get's destroyed here (but not vec).
}
```

Uff.

Unified Shared Memory (USM)

If you like pointers more than buffers/accessors.

- USM allows allocations visible to both host and device(s)
- Three different allocation types available:

host	in host memory
device	in device memory, not accessible by host
shared	in shared memory, accessible by host and device

- Three allocations function (plus overloads)
  - `sycl::malloc_host`
  - `sycl::malloc_device`
  - `sycl::malloc_shared`
- Templated versions exist → `sycl::malloc_host<mytype>(n_elems)`
- Free with `sycl::free`

Data Movement with USM

Two options:

1. Let runtime automatically manage data transfers→ only works for shared allocations.
2. Explicit data transfers.

Explicit data transfer/utility functions

- `queue.memcpy(dst, src, num_bytes)` – copy data to/from device memory.
- `queue.copy<T>(src, dst, count)` – templated version (watch for argument order!)
- `memset` and `fill` (templated version)
-

Example with USM

```
double* d_vec = sycl::malloc_device<double>(n, queue);

// init on device, assume queue is created with in_order property
queue.parallel_for(n, [=](sycl::id<1> id) {
        d_vec[id] = id;
});

queue.parallel_for(n, [=](sycl::id<1> id) {
        // compute somethin on d_vec
});

double* h_vec = sycl::malloc_host<double>(n, queue);
queue.memcpy<double>(d_vec, h_vec, n).wait();

// queue.wait();
```

Time to Practise

- Try to implement stream benchmark using SYCL
- You already know the incredients

```
double* d_vec = sycl::malloc_device<double>(n, queue);

// init on device, assume queue is created with in_order property
queue.parallel_for(n, [=](sycl::id<1> id) {
        d_vec[id] = id;
});

queue.parallel_for(n, [=](sycl::id<1> id) {
        // compute somethin on d_vec
});

double* h_vec = sycl::malloc_host<double>(n, queue);
queue.memcpy<double>(d_vec, h_vec, n).wait(); // or queue.wait();
```

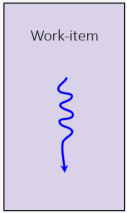
Goals

What we need to use the GPU (not different from OpenMP):

- Get code executed on device. ✔
- Get data allocated and moved between host and device. ✔
- Exploit GPUs' parallelism. ✔

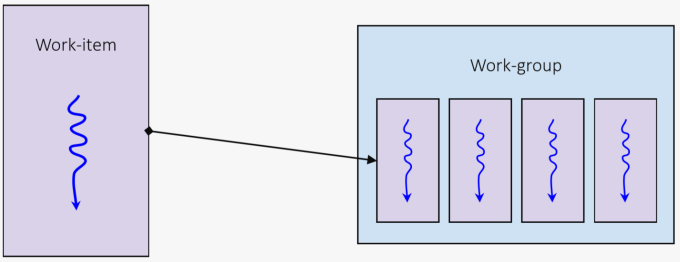
Execution Model: Work-Item

- SYCL kernel functions are executed by **work-items**
- Think of a work-item as a thread of execution
- Work-item can run on CPU threads, SIMD lanes, GPU threads, ...



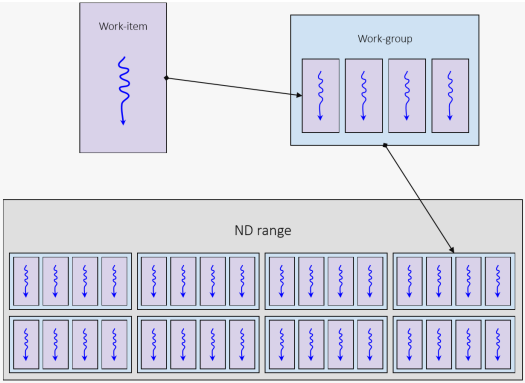
Execution Model: Work-Group

- Work-items grouped in . . . **work-groups**
- Work-group size is adjustable



Execution Model with Work-Groups

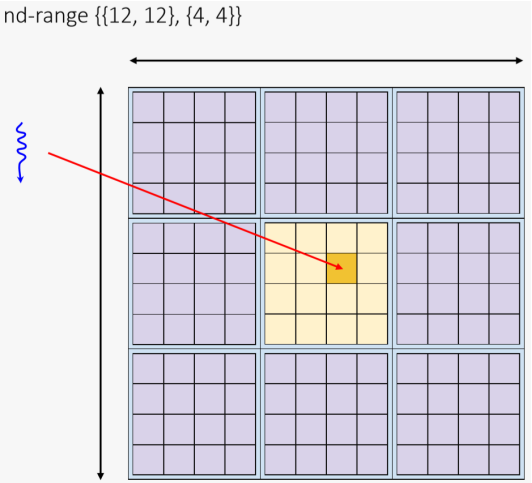
- `sycl::nd_range` can be used to specify both global range, i.e. problem size, plus work-group size



A Closer Look on the Iteration Space

- Each invocation in the iteration space of an nd-range is a work-item
- Each work-item has the following:
  - Global range: {12, 12}
  - Global id: {6, 5}
  - Group range: {3, 3}
  - Group id: {1, 1}
  - Local range: {4, 4}
  - Local id: {2, 1}
- convenience function

```
nd_item::get_global_linear_id()
```



Iterating through the Iteration Space

Three ways for index retrieval/iteration space model:

1. Have `sycl::id` as parameter to kernel function:
   ```
   parallel_for(sycl::range<1>(gs), [=](sycl::id<1> id) {...} )
   ```
   Local Range is decided by runtime.

2. Have `sycl::item` as parameter to kernel function:
   ```
   parallel_for(sycl::range<1>(gs), [=](sycl::item<1> id) {...} )
   ```
   Local Range is decided by runtime.
   Difference to `id`: `item` also contains global range

3. Provide `sycl::nd_range` and `sycl::nd_item`
   ```
   parallel_for(sycl::nd_range<1>(gs, ls), [=](sycl::nd_item<1> id) {...}
   ```

Try Yourself

- Take Stream benchmark code
- Play around with work group size and check obtained performance

Things not Covered

- Using events for synchronization/dependencies
- Reduction variables
- Working with different memory types (shared, constant, ……)
- Image and (experimental) bindless images
- Sychronization
- Handling of multiple devices and subdevices
- Specialization Constants
- Programming for other device types (CPUs, FPGAs[?])
- (Using SYCLomatic)
- InterOp with vendor tools, like profiler etc.

Further Reading

- James Reinders, James Brodman, John Pennycook et al: Data Parallel C++ – Programming Accelerated Systems Using C++ and SYCL. PDF freely available
- SYCLAcademy with step-by-steps tutorials
- CodePlay SYCL Guide – Product discontinued, but documentation still good.
- SYCL Specification (Khronos Group)
- SYCL tutorials

SYCL Summary

- More explicit, yet high-level GPU programming
- Cross platform (not demonstrated for OpenMP)
- Performance can be portable as well.

Overall Summary

- Took a look at OpenMP and SYCL
- Different approaches for GPU programming
- GPUs are powerful devices → have problems that can make use of the power
- It takes time to get familar with (efficient) GPU programming

Questions? Discussion!