Expressing Parallelism with SYCL: Data-Parallel Kernels

André Cerqueira

November 25, 2024

This work was funded by Project 10110190 – EUROCC2, with financial support from FCT/MCTES through national funds (PIDDAC).

Session Objectives

- Understand the concepts of **work-items**, **work-groups**, and **ND-ranges**.
- Understand basic kernels using `parallel_for`.
- Explore ND-range-based kernels to express locality and parallelism.
- Study **Matrix Multiplication**: Basic and optimized approaches.

What are Work-Items?

- **Work-Item:** The smallest unit of execution in SYCL, analogous to a thread.
- Executes a single instance of the kernel function.
- Identified uniquely within the computational grid using global IDs.
- Work-items are independent and cannot synchronize or share data directly.

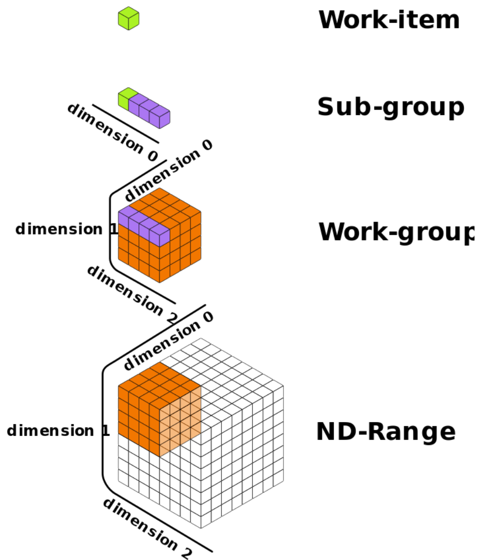
What are Work-Groups?

- **Work-Group:** A collection of work-items that can share data and synchronize.
- Work-items in a group have access to:
  - **Local memory**, shared within the group.
  - **Barriers and fences** for synchronization.
- Identified by a unique group ID within the computational grid.
- Work-items in a group are scheduled concurrently on the same compute unit.

What is an ND-Range?

- **ND-Range:** Defines the total grid of work-items and their division into work-groups.
- Comprised of:
  - **Global Range:** Total number of work-items.
  - **Local Range:** Size of each work-group.

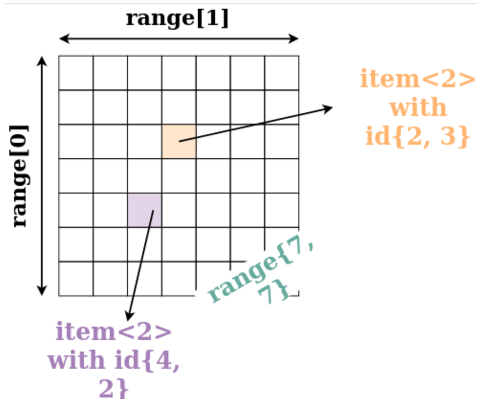
What is an ND-Range ?



Basic Data-Parallel Kernels

- **Execution Range:**
  - Defined using a `range` object (1-, 2-, or 3-dimensional).
  - Each element corresponds to a `work-item`.
  - Work-items are uniquely addressable using:
    - `id`: Lightweight, kernel-instance-specific index.
    - `item`: Kernel-instance index with execution range info.

Basic Data-Parallel Kernels



Basic Kernel Example: Using `id`

```
Q.submit([&](handler &cgh) {
    accessor acc { buf, cgh, write_only };

    cgh.parallel_for(range<2> { n_work_items }, [=](id<2> idx) {
        acc[idx] = 42.0;
    });
});
```

- **Scenario:** Each kernel instance accesses a single element in the buffer.
- **Key Features:**
  - Lightweight and simple for parallel problems.
  - Accessors index buffers directly using `id`.

Basic Kernel Example: Using `item`

```
Q.submit([&](handler &cgh) {
    auto accA = bufA.get_access<access::mode::read>(cgh);
    auto accB = bufB.get_access<access::mode::read>(cgh);
    auto accR = bufR.get_access<access::mode::write>(cgh);

    cgh.parallel_for(range { dataSize }, [=](item<1> itm) {
        auto globalId  = itm.get_id();
        accR[globalId] = accA[globalId] + accB[globalId];
    });
});
```

- **Scenario:** Accessing kernel instance and execution range details.
- **Key Features:**
  - `item` provides global ID and range info.
  - Enables more flexibility for advanced computations.

When to Use `id` vs. `item`

- Choosing between `id` and `item` depends on the problem's complexity:

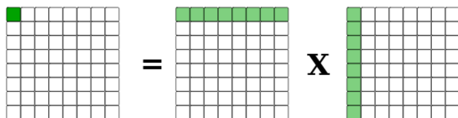
Feature	`id`	`item`
Kernel Instance Awareness	Yes	Yes
Access to Global Range Info	No	Yes
Use Case	Simple operations	Advanced operations
Overhead	Minimal	Slightly higher
Example	`acc[idx] = value;`	`acc[it.get_id()] = sum;`

- `id`: Lightweight and sufficient for simple problems.
- `item`: Adds flexibility for advanced use cases (e.g., vector operations).

Matrix Multiplication

- Compute $C[i,j] = \sum_k A[i,k] \cdot B[k,j]$ using data-parallel kernels.
- Each work-item computes one element of the resulting matrix.
- Challenges:
  - Loading the operands multiple times (inefficient memory usage).
  - Lack of memory reuse without explicit locality handling.

Basic Data-Parallel Kernels



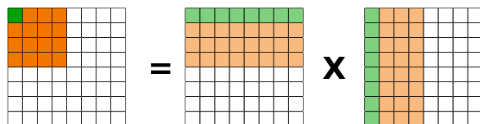
Basic Matrix Multiplication Kernel

```
Q.submit([&](handler &cgh) {
    accessor A { bufA, cgh, read_only };
    accessor B { bufB, cgh, read_only };
    accessor C { bufC, cgh, write_only };

    cgh.parallel_for(range<2> {N, N}, [=](id<2> idx) {
        int row = idx[0];
        int col = idx[1];
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row][k] * B[k][col];
        }
        C[row][col] = sum;
    });
});
```

- **Global Range:** Defines the 2D execution space.
- **Kernel Logic:** Each work-item calculates one matrix element.
- **Challenge:** Inefficient due to repeated data loads.

Optimized Matrix Multiplication with ND-Range



- **Work-Item Role:** Each work-item computes an element in the result matrix by accessing a full row of A and a full column of B.
- **Data Reuse:** Unlike the naive approach, work-items in a work-group can reuse data from shared memory, improving memory access locality and reducing redundant loads.

Optimized Matrix Multiplication with ND-Range: Code

```
Q.submit([&](handler &cgh) {
    accessor A { bufA, cgh, read_only };
    accessor B { bufB, cgh, read_only };
    accessor C { bufC, cgh, write_only };

    range<2> global {N, N};
    range<2> local {B, B};

    cgh.parallel_for(nd_range<2>(global, local), [=](nd_item<2> it)
    ↪  {
        int row = it.get_global_id(0);
        int col = it.get_global_id(1);
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row][k] * B[k][col];
        }
        C[row][col] = sum;
    });
});
```

- **ND-range:** Divides the workload into work-groups.
- **Memory Locality:** Shared data within work-groups allows optimized memory access.
- **Improvement:** Reduces redundant memory loads, leveraging better performance.

Optimized Matrix Multiplication Explained

- **Global Range:** Defines the total execution grid (e.g., $N \times N$).
- **Local Range:** Specifies the size of each work-group (e.g., $B \times B$).
- **Work-Groups:** Allow work-items to share data in local memory.
- **Key Insight:** Data locality reduces memory traffic significantly.
- **Result:** Improved performance due to fewer redundant loads and better utilization of hardware capabilities.

Important Note: Don't Reinvent the Wheel!

Use Optimized Libraries

Matrix Multiplication and similar operations are fundamental but computationally intensive.

- SYCL is great for learning or custom optimizations.
- **For production:** Use optimized BLAS libraries like **oneAPI**.

Why?

- Expert-tuned for specific hardware (e.g., CPUs, GPUs).
- Advanced optimizations like vectorization and memory reuse.
- Orders of magnitude faster and more reliable.

Summary

- **Work-items:** Basic units of execution.
- **Work-groups:** Enable data sharing and synchronization within groups.
- **ND-ranges:** Define computational grids with control over locality.
- **Data-parallel kernels:** Achieved with `parallel_for`.
- **Matrix Multiplication:** Optimized kernels improve memory access patterns and performance.