SC22

Dallas, TX | hpc accelerates.

# Hands-On HPC Application Development Using C++ and SYCL

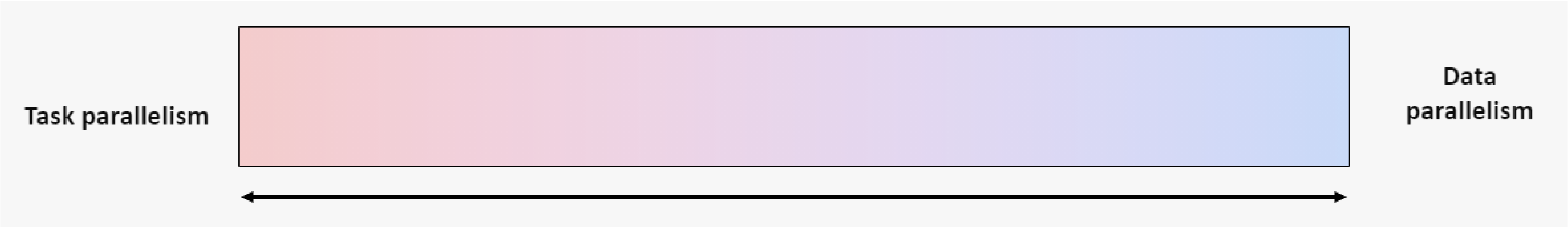
Dounia Khaldi, Hugh Delaney, James Reinders, Michael Wong, Ronan Keryell

# DATA PARALLELISM

# LEARNING OBJECTIVES

- Learn about task parallelism and data parallelism
- Learn about the SPMD model for describing data parallelism
- Learn about SYCL execution and memory models
- Learn about enqueuing kernel functions with `parallel_for`

# TASK VS DATA PARALLELISM



- **Task parallelism** is where you have several, possibly distinct tasks executing in parallel.
  - In task parallelism you optimize for latency.
- **Data parallelism** is where you have the same task being performed on multiple elements of data.
  - In data parallelism you optimize for throughput.

# VECTOR PROCESSORS

- Many processors are vector processors, which means they can naturally perform data parallelism.
    - GPUs are designed to be parallel.
    - CPUs have SIMD instructions which perform the same instruction on a number elements of data.

# SPMD MODEL FOR DESCRIBING DATA PARALLELISM

## Sequential CPU code

```
void calc(const int in[], int out[]) {
  // all iterations are run in the same
  // thread in a loop
  for (int i = 0; i < 1024; i++){
    out[i] = in[i] * in[i];
  }
}

// calc is invoked just once and all
// iterations are performed inline
calc(in, out);
```

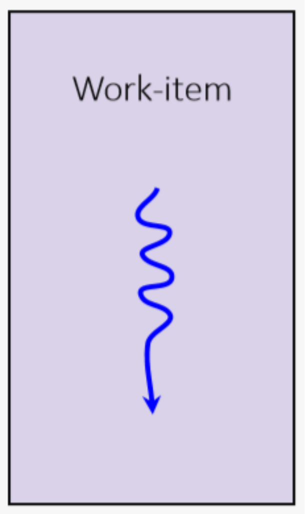
## Parallel SPMD code

```
void calc(const int in[], int out[], int id) {
  // function is described in terms of
  // a single iteration
  out[id] = in[id] * in[id];
}

// parallel_for invokes calc multiple
// times in parallel
parallel_for(calc, in, out, 1024);
```

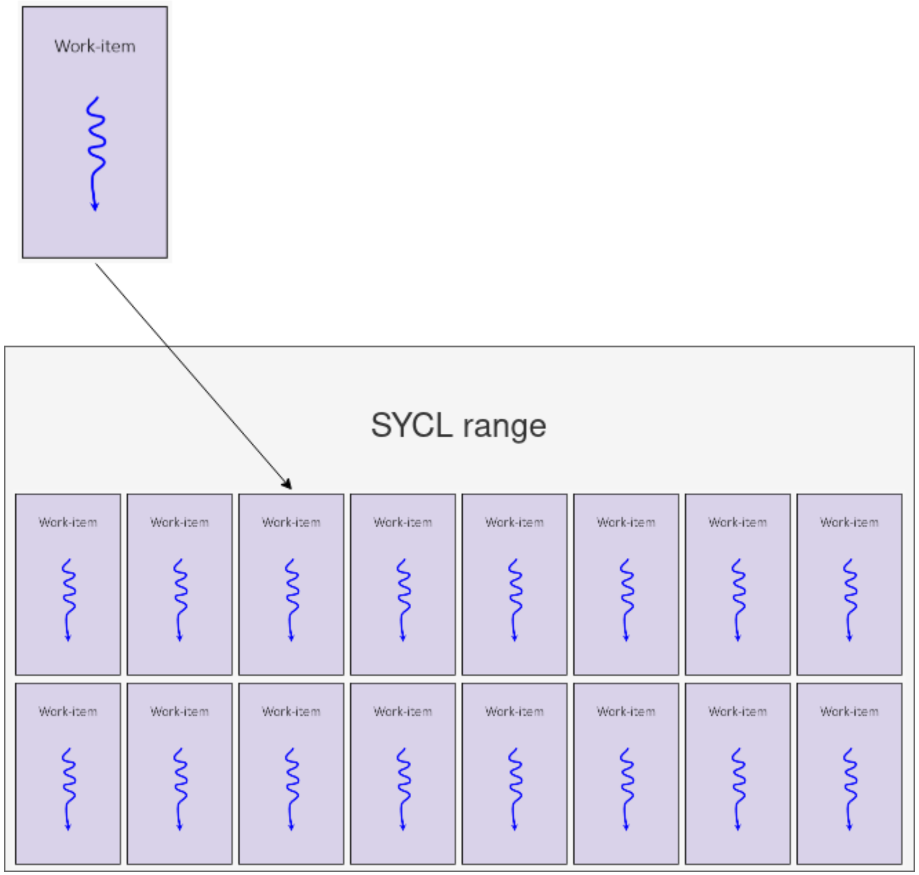
# SYCL EXECUTION MODEL

- In SYCL kernel functions are executed by **work- items**.
- You can think of a work-item as a thread of execution.
- Each work-item will execute a SYCL kernel function from start to end.
- A work-item can run on CPU threads, SIMD lanes, GPU threads, or any other kind of processing element.



# SYCL EXECUTION MODEL

- Work-items are launched in parallel in a `sycl::range`.
- In order to maximize parallelism, the range should correspond to the problem size.



# PARALLEL_FOR

```
cgh.parallel_for<my_kernel>(range{64, 64},
                            [=](id<2> idx){
  // SYCL kernel function is executed
  // on a range of work-items
});
```

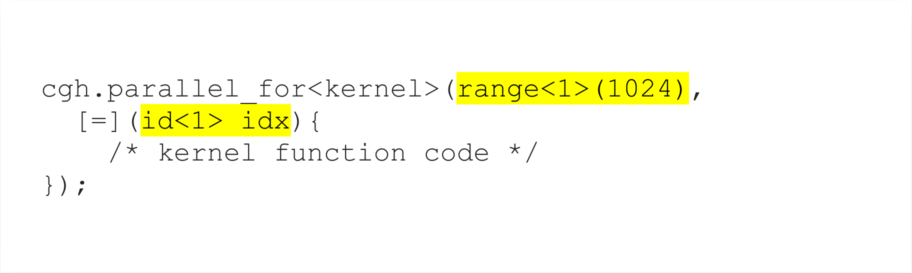
- In SYCL kernel functions can be enqueued to execute over a range of work-items using `parallel_for`.
- When using `parallel_for` you must also pass `range` which describes the number of iteration space to be executed over.

# PARALLEL_FOR

```
cgh.parallel_for<my_kernel>(range{64, 64},
                          [=](id<2> idx){
  // SYCL kernel function is executed
  // on a range of work-items
});
```

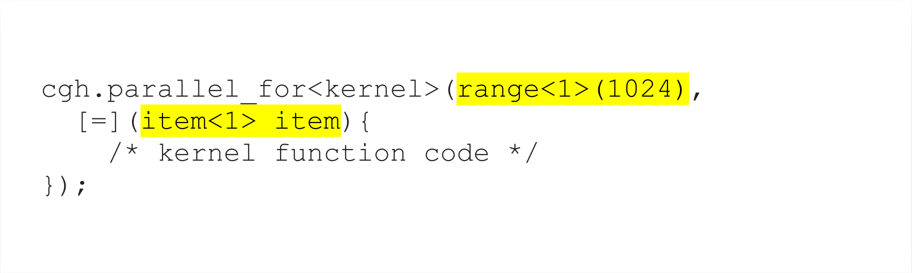
- When using `parallel_for` you must also have the function object which represents the kernel function take an `id`.
- This represents the current work-item being executed and its position within the iteration space.

# EXPRESSING PARALLELISM



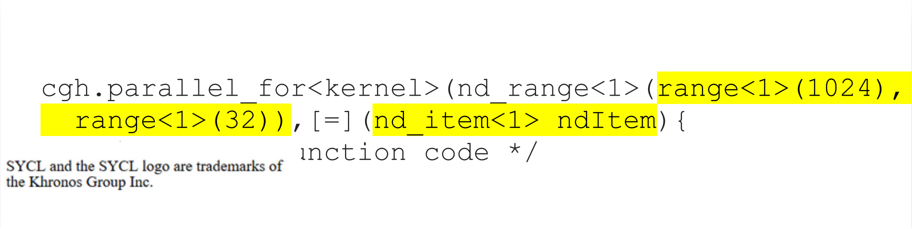
```
cgh.parallel_for<kernel>(range<1>(1024),
  [=](id<1> idx){
    /* kernel function code */
});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **id** parameter represents the index within the global range



```
cgh.parallel_for<kernel>(range<1>(1024),
  [=](item<1> item){
    /* kernel function code */
});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **item** parameter represents the global range and the index within the global range



```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),
  range<1>(32)),[=](nd_item<1> ndItem){
                    unction code */
```

# QUESTIONS

# EXERCISE

Code_Exercises/Exercise_06_Vector_Add/source.cpp

Implement a SYCL application that adds two arrays of values together in parallel using `parallel_for`.