**What are Subgroups?**

**Subset of the work-items** in a **work-group** are **executed simultaneously** or **with additional scheduling guarantees**. ***These subset of work-items are called subgroups***. **Leveraging subgroups** will **help to map execution** to **low-level hardware** and **may help in achieving higher performance**.

**Subgroups in ND-Range Kernel Execution**

Text, application

Description automatically generatedDiagram

Description automatically generatedParallel execution with the **ND\_RANGE Kernel** **helps** to **group work items that map to hardware resources**. This helps to tune applications for performance. The **execution range of an ND-range kernel** **is divided into work-groups**, **subgroups** and **work-items**



The picture below shows how work-groups and subgroups map to Intel® Gen11 Graphics Hardware.

Diagram

Description automatically generatedND-Range Hardware Mapping



**Why use Subgroups?**

**Work-items** in a **sub-group** can **communicate directly using shuffle operations**, **without explicit memory operations**.

**Work-items** in a **sub-group** can **synchronize using sub-group** **barriers** and **guarantee memory consistency** ***using sub-group memory fences***.

**Work-items** in a **sub-group** have **access** to **sub-group functions and algorithms**, **providing fast implementations of common parallel patterns**.

**sub\_group class**

The **subgroup handle** can be **obtained from the nd\_item using** the **get\_sub\_group()**

sycl::sub\_group sg = nd\_item.get\_sub\_group();

OR

auto sg = nd\_item.get\_sub\_group();

Once **you have the subgroup handle**, you **can query for more information** **about the subgroup, do shuffle operations** or **use group algorithm**.

**Subgroup info**

The **subgroup handle** **can be queried to get other information like number of work-items in subgroup**, or **number of subgroups in a work-group** which **will be needed for developers to implement kernel code using subgroups**:

* get\_local\_id() = **returns** the **index** **of** the **work-item within its subgroup**
* get\_local\_range() = **returns** the **size** of **sub\_group**
* get\_group\_id() = **returns** the **index** of the **subgroup**
* get\_group\_range() = **returns** the **number of subgroups within** the **parent work-group**

h.parallel\_for(nd\_range<1>(64,64),[=](nd\_item<1> item){

/\* get sub\_group handle \*/

auto sg = item.get\_sub\_group();

/\* query sub\_group and print sub\_group info once per sub\_group \*/

if(sg.get\_local\_id()[0] == 0){

out << "sub\_group id: " << sg.get\_group\_id()[0]

<< " of " << sg.get\_group\_range()[0]

<< ", size=" << sg.get\_local\_range()[0]

<< "\n";

}

});

**Subgroup Size**

For **tuning applications for performance**, **sub-group size** may **have to be set a specific value**. For example, **Intel(R) GPU supports sub-groups sizes of 8, 16 and 32**; by **default the compiler implementation will pick optimal sub-group size**, but **it can also be forced to use a specific value**.The **supported sub-group sizes for a GPU can be queried from device information** as shown below:

auto sg\_sizes = q.get\_device().get\_info<info::device::sub\_group\_sizes>();

**^**

reqd\_sub\_group\_size(S) allows **setting a specific sub-group size** *to* **use for kernel execution**, the **specified value should** be **one of the supported sizes** and **must be a compile time constant value**.

q.parallel\_for(nd\_range<1>(N, B), [=](nd\_item<1> item)[[intel::reqd\_sub\_group\_size(16)]] {

**^**

// Kernel Code

}).wait();

Subgroup Functions and Algorithms

The sub-group functions and algorithms expose functionality tied to work-items within a sub-group.

Providing these implementations as library functions instead increases developer productivity and gives implementations the ability to generate highly optimized code for individual target devices.

Below are some of the group algorithms available for sub-groups, they include useful functionalities to perform shuffles, reductions, scans and votes:

select\_by\_group

shift\_group\_left

shift\_group\_right

permute\_group\_by\_xor

group\_broadcast

reduce\_over\_group

exclusive\_scan\_over\_group

inclusive\_scan\_over\_group

any\_of\_group

all\_of\_group

none\_of\_group

Subgroup Shuffle

One of the most useful features of subgroups is the ability to communicate directly between individual work-items without explicit memory operations.

Shuffle operations enable us to remove work-group local memory usage from our kernels and/or to avoid unnecessary repeated accesses to global memory.

Below are the different types of shuffle operations available for sub-groups:

select\_by\_group(sg, x, id)

shift\_group\_left(sg, x, delta)

shift\_group\_right(sg, x, delta)

permute\_group\_by\_xor(sg, x, mask)

The code below uses permute\_group\_by\_xor to swap the values of two work-items:

h.parallel\_for(nd\_range<1>(N,B), [=](nd\_item<1> item){

auto sg = item.get\_sub\_group();

auto i = item.get\_global\_id(0);

/\* Shuffles \*/

//data[i] = select\_by\_group(sg, data[i], 2);

//data[i] = shift\_group\_left(sg, data[i], 1);

//data[i] = shift\_group\_right(sg, data[i], 1);

data[i] = permute\_group\_by\_xor(sg, data[i], 1);

});

shuffle\_xor

Code Example: Subgroup - Reduce

The code below uses subgroup reduce\_over\_group function to perform reduction for all items in a subgroup.

h.parallel\_for(nd\_range<1>(N,B), [=](nd\_item<1> item){

auto sg = item.get\_sub\_group();

auto i = item.get\_global\_id(0);

/\* Reduction algorithm on Sub-group \*/

int result = reduce\_over\_group(sg, data[i], plus<>());

//int result = reduce\_over\_group(sg, data[i], maximum<>());

//int result = reduce\_over\_group(sg, data[i], minimum<>());

});

The SYCL code below demonstrates sub-group algorithm: Inspect code, you can change the operator "plus" to "maximum" or "minimum" and check output:

Code Example: Subgroup - Votes

The any\_of\_group, all\_of\_group and none\_of\_group functions (henceforth referred to collectively as “vote” functions) enable work-items to compare the result of a Boolean condition across their group.

The SYCL code below demonstrates sub-group algorithms any\_of\_group, all\_of\_group and none\_of\_group functions: Inspect code, there are no modifications necessary:

**SYCL\* Buffers are powerful and elegant**. ***Use them*** **if the abstraction applies cleanly** in your application, ***and/or if buffers aren’t disruptive to your development***. However, *replacing all pointers and arrays with buffers in a C++ program can be a burden* to programmers so **in this case consider using USM**.

USM provides a familiar pointer-based C++ interface:

* Useful when porting C++ code to SYCL by minimizing changes.
* Use shared allocations when porting code to get functional quickly. Note that shared allocation is not intended to provide peak performance out of box.
* Use explicit USM allocations when controlled data movement is needed.

**Data dependency in USM**

When using unified shared memory, **dependences between tasks must be specified** using **events** since ***tasks execute asynchronously*** and **multiple tasks can execute simultaneousl**y.

Programmers **may explicitly wait on event objects** or use the depends\_on method inside a command group to s**pecify a list of events that must complete before a task may begin.**

In the example below, the **two kernel tasks** are **updating the same data array**, these two **kernels can execute simultaneously and may cause undesired result**. The *first task must be complete before the second can begin*

q.parallel\_for(range<1>(N), [=](id<1> i) {data[i] += 2;});

q.parallel\_for(range<1>(N), [=](id<1> i) {data[i] += 3;});

**Different options to manage data dependency when using USM:**

**wait() on kernel task**

Use **q.wait()** on kernel task to wait before the next dependent task can begin, however it will block execution on host.

q.parallel\_for(range<1>(N), [=](id<1> i) { data[i] += 2; });

q.wait(); *// <--- wait() will make sure that task is complete before continuing*

q.parallel\_for(range<1>(N), [=](id<1> i) { data[i] += 3; });

**Use in\_order queue property**

Use **in\_order** queue *property for the queue*, this will **serialize all the kernel tasks**. Note that **execution will not overlap even if the queues have no data dependency**.

queue q{property::queue::in\_order()}; // <--- this will serialize all kernel tasks

q.parallel\_for(range<1>(N), [=](id<1> i) { data[i] += 2;});

q.parallel\_for(range<1>(N), [=](id<1> i) { data[i] += 3;});

**Use depends\_on queue property**

Use h.depends\_on(e) method in the command group to specify events that must complete before a task may begin.

auto e = q.submit([&](handler &h){ // <--- e is event for kernel task

h.parallel\_for(range<1>(N), [=](id<1> i){

data[i] += 2;

});

});

q.submit([&](handler &h) {

h.depends\_on(e); // <--- waits until event e is complete

h.parallel\_for(range<1>(N), [=](id<1> i){

data[i] += 3;

});

});

You can also use a simplified way of specifying dependencies by passing an extra parameter in parallel\_for

auto e = q.parallel\_for(range<1>(N), [=](id<1> i){

data[i] += 2;

});

q.parallel\_for(range<1>(N), e, [=](id<1> i) { data[i] += 3;});

**^**